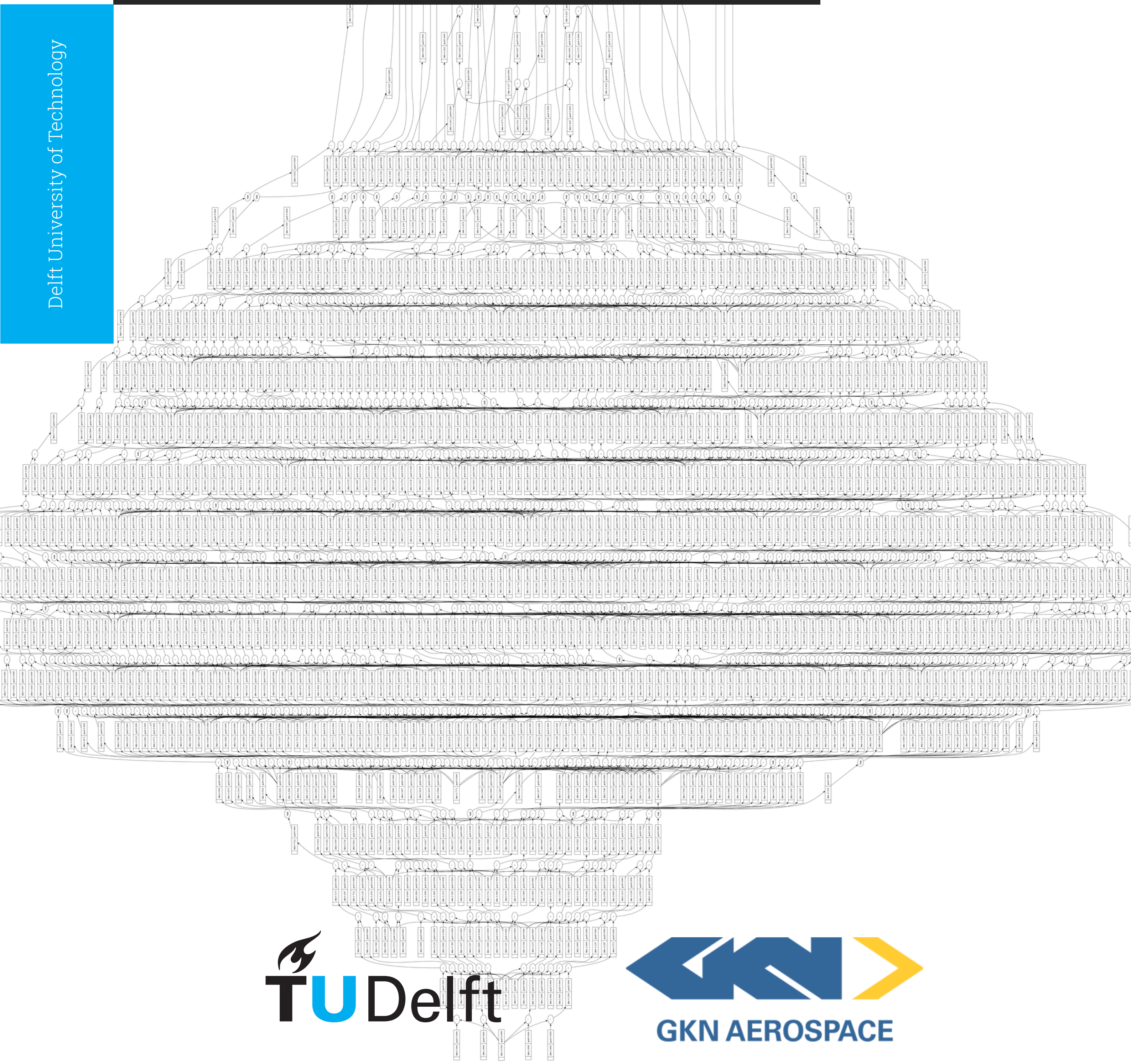
Toward an End-to-End Pipeline from Requirements to Code

Large Language Model Supported Coding Assistant for Knowlegde Based Engineering Application Development

E. Hof

Delft University of Technology



Toward an End-to-End Pipeline from Requirements to Code

Large Language Model Supported Coding Assistant for Knowlegde Based Engineering Application Development

by

E. Hof

in partial fulfilment of the requirements for the degree of

Master of Science in Aerospace Engineering

at the Delft University of Technology,

to be defended publicly on December 12, 2025 at 14:00.

Thesis committee:	
Chair:	Prof. dr. ir. L.L.M. Veldhuis
Supervisors:	dr. ir. G. La Rocca
	drs. ir. A.P. Gómez
External examiner:	dr. ir. P. Proesmans
Place:	Faculty of Aerospace Engineering, Delft
Project duration:	January, 2025 – December, 2025
Student name:	E. Hof
Student number:	4852680

Cover:	*Dependency graph of a simple multi-layer perceptron (own work)*
Style:	*TU Delft Report Style, with modifications by Daan Zwaneveld*

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

Word count: 35771

Abstract

Developing Knowledge-Based Engineering (KBE) applications remains a significant challenge in high-tech industries like aerospace, where front-loaded product development demands extensive automation. The manual code completion phase of these applications is particularly problematic: it consumes considerable time, requires specialized expertise in proprietary frameworks, and creates steep learning curves that limit broader adoption. While recent advances in Artificial Intelligence (AI) have revolutionized software development assistance, commercial AI systems consistently fail when working with specialized KBE frameworks, like ParaPy. In the absence of sufficient training data on proprietary frameworks, these systems produce code that appears correct but is actually non-functional.

This research validates that retrieval-augmented approaches offer a practical alternative to retraining models for specialized domains with limited data, such as ParaPy. In these approaches, AI systems dynamically access domain-specific knowledge during operation rather than relying solely on their training. This has significant implications for industries or institutions using proprietary tools where comprehensive model retraining is economically infeasible.

The research implements this approach by developing and evaluating a dual-agent framework for AI-assisted KBE application development that operates within industrial privacy and security constraints. The framework comprises a Developer Agent optimized for code generation and debugging with the ParaPy SDK, and an Educational Agent focused on ParaPy learning support and documentation. Both agents access a knowledge infrastructure that uses semantic search over indexed ParaPy documentation, curated examples, and technical references. Additionally, the Developer Agent employs verification mechanisms that progressively check code at two core levels: syntax correctness (ensuring the code follows programming language rules) and successful execution (confirming the code runs without errors).

User testing revealed how different skill levels benefit from AI assistance. Intermediate users benefited most, showing dramatic improvements in productivity and performance. Novice users achieved substantial productivity gains and reduced framework-specific (ParaPy) errors significantly, with task completion rates approaching expert baseline performance. Expert users, however, experienced slight performance degradation due to reduced code review under time pressure. The framework successfully reduced knowledge barriers for novice and intermediate users, broadening access to specialized engineering tools.

Despite these successes, the framework has persistent limitations in understanding three-dimensional spatial relationships. This is critical for KBE applications where code must define the precise position, orientation, and assembly of physical components. The framework struggles to correctly place components in space or apply proper rotational transformations. This produces code that may be syntactically correct but results in misaligned parts or incorrectly oriented features. These geometric errors require iterative refinement with human guidance, representing a fundamental limitation of the current approach and language model architectures.

While geometric reasoning limitations suggest fundamental boundaries of current AI capabilities, the demonstrated productivity improvements and reduced knowledge requirements establish a foundation for broader AI adoption in knowledge-intensive engineering domains. The framework contributes a validated operational prototype that addresses critical gaps in AI-assisted KBE development: reducing the manual coding bottleneck, lowering knowledge barriers for new users, and providing privacy-compliant deployment options. The framework functions most effectively as a development accelerator requiring expert oversight, supporting human engineers rather than replacing them.

Keywords: Knowledge-Based Engineering, Large Language Models, Code Generation, ParaPy, Retrieval-Augmented Generation, AI-Assisted Development, Aerospace Engineering, Multi-Agent Systems

Acknowledgements

My first and foremost gratitude goes out to my parents — *Mama en Papa* — as well as my family and friends for their unwavering support, for being a place of comfort, reflection, and meaningful discussion, and for allowing me to vent when needed. I am incredibly fortunate to have so many people close by to rely on and talk to.

A sincere thanks goes to my university supervisor, Dr. ir. Gianfranco la Rocca, for setting me on the path of Knowledge-Based Engineering and software development years ago, when I first attended the Knowledge-Based Engineering course at Aerospace Engineering. Thank you for your continuous guidance and valuable feedback throughout this thesis.

To my industry supervisor, drs. ir. Alejandro Pradas Gómez, thank you for your many insightful feedback sessions and for offering me the opportunity to complete this thesis in collaboration with GKN Aerospace Sweden. You made the experience in Sweden truly worthwhile, and I am certain that this work would be of significantly lower quality without your support.

I would also like to thank ParaPy, and in particular Reinier van Dijk and Max Baan, for giving me the opportunity to learn and grow during my time as a part-time software engineer at ParaPy. Special thanks to Levi van Aanholt and San Kilkis for their dedicated feedback on the implementation of AI within the ParaPy environment.

A final and heartfelt thank you to all other employees and colleagues at both ParaPy and GKN Aerospace who took the time to provide feedback, and who made it possible for me to conduct interviews and user testing sessions — your contributions greatly strengthened the validity of this research.

Finally, I would like to thank all others, named and unnamed, who have directly or indirectly supported me throughout this year in pursuit of my engineering degree.

Preface

Since the start of this project, it has been both a joy and a concern to dive deeper into the world of Artificial Intelligence. As AI capabilities continue to grow and increasingly shape our daily lives, it is easy to feel overwhelmed by the vast amount of information available. It is truly a double-edged sword. With services like ChatGPT becoming ever more human-like, information appears more accessible and trustworthy than ever before. However, many people fail to understand the underlying mechanisms of these systems. At their core, models such as ChatGPT are enormous algebraic networks, fine-tuned to predict human-like text based on natural language prompts.

While experts continue to debate whether true thinking or intelligence has been achieved—or ever will be achieved—with current large language model architectures, the outputs of these systems remain what they fundamentally are: natural language predictions designed to sound as plausible as possible given a user's input. Although companies such as OpenAI, Anthropic, and DeepSeek are extending their services by enabling models to access external tools and up-to-date information, I often worry about the lack of fact-checking among users who rely on AI in their daily lives.

Despite these concerns, I confidently embrace AI in my own work and have witnessed numerous positive impacts. These include the enormous and emerging AI community spearheading open-source models, solutions, and frameworks, supported by market leaders and professionals from industry and academia alike. The benefits span multiple sectors: faster cancer diagnoses in healthcare, improved efficiency and cost reduction in business, personalized education approaches, and enhanced cybersecurity measures. At the same time, we must remain aware of the negative consequences and risks: biased models (intentional or not), ethical and privacy issues, military applications reminiscent of "Terminator-like" scenarios, and the substantial environmental footprint of large-scale data centres.

In a world where progress continues to accelerate exponentially, we should embrace innovation with both optimism and caution. It will be fascinating to see how quickly the technologies and approaches I explore and discuss here may appear archaic to future readers. Above all, it has been a joy and a privilege to investigate how I, as a software engineer, might one day be replaced—or perhaps complemented—by the very technologies I study.

Ernesto Hof
Delft, August 2025

Voorwoord

Sinds de start van dit project is het zowel een bron van enthousiasme als van zorg geweest om dieper in de wereld van kunstmatige intelligentie te duiken. Naarmate KI-systemen krachtiger worden en een steeds grotere rol spelen in ons dagelijks leven, kan de hoeveelheid aan ogenschijnlijke informatie al snel overweldigend zijn. Het voelt soms als een mes dat aan twee kanten snijdt. Diensten zoals ChatGPT ogen steeds menselijker, waardoor kennis toegankelijker en betrouwbaarder lijkt dan ooit. Toch realiseren veel mensen zich nauwelijks hoe zulke systemen daadwerkelijk werken. In de kern zijn modellen als ChatGPT niets meer dan gigantische wiskundige netwerken, fijn afgestemd om mensachtige tekst te voorspellen op basis van natuurlijke taal.

Terwijl wetenschappers onderling blijven discussiëren of we met de huidige generatie grote taalmodellen al echt van 'denken' of 'intelligentie' kunnen spreken – en of dat punt ooit bereikt zal worden – blijft één feit overeind: de uitkomsten zijn en blijven statistische voorspellingen, ontworpen om zo aannemelijk mogelijk te klinken. Bedrijven als OpenAI, Anthropic en DeepSeek breiden hun modellen weliswaar uit met hulpmiddelen die toegang geven tot actuele informatie, maar ik zie nog te vaak dat gebruikers klakkeloos aannemen wat KI-systemen produceren, zonder enige vorm van factchecking.

Dat neemt niet weg dat ik KI zelf met overtuiging omarm in mijn werk en de positieve impact ervan dagelijks ervaar. Deze omvat de enorme en groeiende KI-gemeenschap die opensource-modellen, oplossingen en frameworks ontwikkelt, gesteund door marktleiders en professionals uit zowel de industrie als de academische wereld. De voordelen strekken zich uit over meerdere sectoren: snellere kankerdiagnoses in de zorg, verbeterde efficiëntie en kostenbesparing in het bedrijfsleven, gepersonaliseerde onderwijsbenaderingen en versterkte cybersecurity. Tegelijkertijd mogen we de schaduwzijde niet uit het oog verliezen: modellen die – bewust of onbewust – bevooroordeeld zijn, ethische en privacyvraagstukken, militaire toepassingen die doen denken aan sciencefictionfilms en de enorme ecologische voetafdruk van megadatacenters.

In een wereld waarin technologische vooruitgang in hoog tempo blijft toenemen, is het zaak om innovatie met open armen te verwelkomen, maar wel met een gezonde dosis waakzaamheid. Met fascinatie (en een lichte dosis spanning) kijk ik naar de toekomst en ben ik benieuwd hoe snel de technologieën en benaderingen die ik hier bespreek, antiek zullen lijken voor de aanstaande lezer. Voor mij persoonlijk was het vooral een voorrecht om te onderzoeken hoe ik als software engineer ooit misschien (deels) kan worden vervangen – of juist aangevuld – door de technologieën die ik nu met zoveel nieuwsgierigheid bestudeer.

Ernesto Hof
Delft, Augustus 2025

Contents

List of Figures

List of Tables

Nomenclature

Symbol	Definition	Unit
k	Number of results in top-k sampling or semantic search	[-]
n	Number of dimensions in vector space or number of parameters	[-]
p	Probability threshold in nucleus sampling	[-]
T	Temperature parameter controlling randomness in token selection	[-]
θ	Angle between vectors in cosine similarity calculation	[rad]
$\vec{A}, \vec{B}$	Vector representations in embedding space	[-]

Part I

Thesis

1
Introduction

Humankind stands at the forefront of—some argue, already within—a fourth industrial revolution, poised to reshape our way of life [1]. Human creativity and the relentless pursuit of progress seem deeply ingrained in our nature, driving technological innovation [2]. Amidst growing socioeconomic challenges [3, 4, 5] and intensifying global competition, high-tech industries face constant pressure to accelerate time to market and enhance cost efficiency of products. Aerospace Engineering is no exception; in fact, it exemplifies a highly complex, inherently multidisciplinary, and heavily regulated sector that stands to benefit significantly from advancements in technology and engineering methodologies.

A well-established approach to achieving these improvements is *front-loaded* product development, or "a strategy that seeks to improve development performance by shifting the identification and solving of [design] problems to earlier phases of a product development process" [6]. Front-loaded development builds upon *concurrent engineering*, which enhances traditional, often highly sequential, design processes by executing multiple design tasks in parallel. This approach aims to boost productivity and profitability while reducing lead times and last-minute (re)work [7]. Although concurrent engineering improves upon traditional methods, it introduces its own challenges. Engineering design is inherently multidisciplinary and cannot be fully parallelized. Interdependent systems and design processes must rely on "advance information" —assumptions about initially unavailable dependent data— throughout much of the design process [8]. Consequently, concurrent engineering often creates time pressure on upstream activities, limiting the depth of engineering work, encouraging conservative designs, and ultimately increasing costs [8, 9].

The enhanced front-loading approach [6, 10] aims to identify and address critical bottlenecks early in the design process by implementing automated design workflows and developing *Knowledge Bases*. These tools enable rapid product design at the start of new projects by leveraging:

1. **Project-to-project knowledge transfer**: Standardized, centralized Knowledge Bases facilitate the transfer of problem-solving insights from one project to the next. This reduces the time required to resolve recurring issues and allows engineers to address potential challenges early, sometimes even before a project begins [8].
2. **Rapid problem-solving**: Advanced technologies and methodologies, such as Computer-Aided Engineering (CAE) in place of physical prototyping, accelerate problem resolution. Additionally, iterative approaches like Design of Experiments (DOE) and optimization enable systematic exploration and refinement of design solutions.

Figure 1.1 qualitatively illustrates the impact of front-loaded engineering on time-to-market and its relationship with concurrent and traditional engineering design.

While other methods exist to streamline product development, the front-loaded approach enables extensive simulations during the design phase, with the resulting designs stored in a dedicated database. This database allows engineers to query a wide range of design variations with accurate performance predictions at the outset of new projects. High-tech suppliers, in particular, may benefit from this approach. Large-scale design explorations aimed at populating the database can reveal valuable patterns or novel solutions, which could be proactively presented as proposals to potential OEMs (Original Equipment Manufacturer) [11].

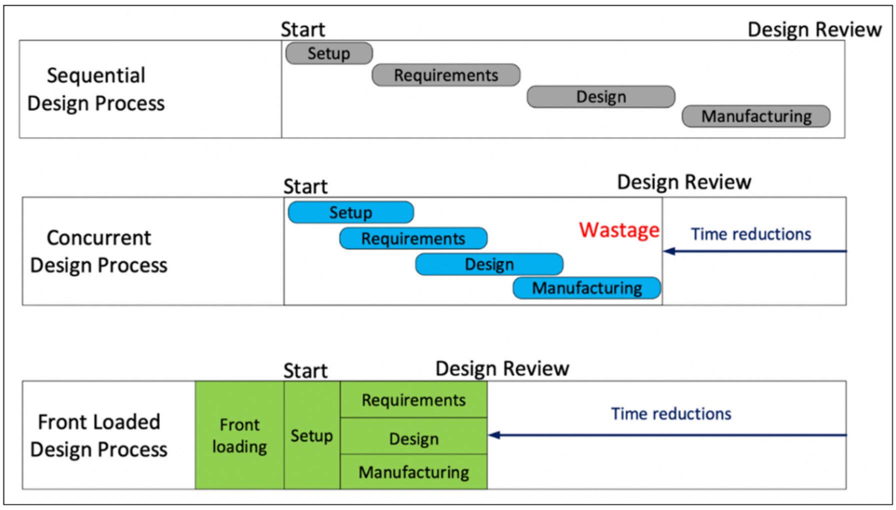


Figure 1.1: The impact of front-loaded engineering on time-to-market relative to traditional and concurrent engineering (adapted version taken from [11], originally published in [12]).

The integration of Knowledge Based Systems and *Knowledge Based Engineering* (KBE) applications into the design process is a key enabler of the front-loaded product development approach. KBE is a technology that combines Computer-Aided Design (CAD) with Expert Systems AI to automate and enhance engineering workflows. KBE platforms facilitate the generation and manipulation of product geometries while embedding engineering knowledge and reasoning through specialized programming languages. By leveraging object-oriented programming, caching, and dependency tracking, KBE enables powerful automation for tasks such as product configuration, design space exploration, and multidisciplinary design optimization. Originating in highly competitive industries like aerospace, automotive, and manufacturing, KBE has become a widely adopted approach for automating workflows, optimizing designs, and ensuring consistency across engineering tasks. [13]

At its core, developing KBE applications involves transforming 'raw' expert knowledge from industry into functional code using the programming language provided by the chosen KBE system. Examples of such systems include ParaPy[1], Viktor[2], Siemens Knowledge Fusion[3], and the Adaptive Modeling Framework from Technosoft[4]. This development process is typically case-based, multi-actor, and iterative, requiring expertise across multiple domains, including -but not limited to- knowledge acquisition and modeling, software engineering, design optimization, and domain-specific engineering knowledge.

This complexity of KBE application development has led to various efforts to formalize and improve the process over the years. One of the earliest and most well-known methodologies, MOKA (Methodology and tools Oriented to Knowledge-based engineering Applications), systematically captures, organizes, and implements engineering knowledge through formal and informal knowledge models, as well as a neutral language knowledge model [14]. Formal models consist of structured, machine-readable representations such as diagrams, ontologies, and specifications, while informal models capture unstructured knowledge like design rationale, expert insights, and documentation. Other methodologies, such as CommonKADS (Common Knowledge Acquisition and Documentation Structuring) [15] —on which MOKA is based— or KNOMAD (Knowledge Nurture for Optimal Multidisciplinary Analysis and Design) [16], have also been developed. However, each comes with its own limitations. At their core, these methodologies touch (at least one of) the fundamental challenges of KBE application development, some of which have already been implicitly mentioned [17]:

1. **Case-based, ad hoc development**: KBE applications are often developed on a case-by-case basis without a structured framework or adherence to existing methodologies. A review of case studies revealed that 81% did not explicitly adhere to a specific methodology, leading to potential

[1] https://parapy.nl/
[2] https://www.viktor.ai/
[3] https://plm.sw.siemens.com/en-US/nx/cad-online/
[4] https://www.technosoft.com/

knowledge loss, misuse, under-utilization, and higher maintenance costs.

2. **Tendency toward "black-box" applications**: Current KBE development leans towards applications where captured knowledge is represented as context-less data and formulas, lacking explication and provisions for capturing design intent. A "round-trip" automatic feature is needed to link application code back to formal and informal models for validation, update, and reuse of underlying knowledge. The work of [18] present a first step in solving this problem.
3. **Lack of knowledge re-use**: The difficulty of reusing knowledge in KBE systems is closely tied to the previous point. Higher-level knowledge, such as project constraint reasoning, problem resolution methods, design intent, and supply chain knowledge, is often not captured or reused.
4. **Limited quantitative frameworks for KBE assessment and justification**: Literature reviews indicate that KBE research has historically lacked comprehensive quantitative methods for both evaluating implemented systems and determining the suitability of tasks for KBE representation. Many case studies do not report time or cost benefits, and systematic frameworks to assess whether a design task warrants KBE development —incorporating both quantifiable metrics and qualitative considerations— remain underdeveloped. While industry practitioners may have developed internal assessment methods, such approaches are often proprietary and absent from published research.

To address these challenges, the recently concluded DEFAINE (Design Exploration Framework based on AI for froNt-loaded Engineering) project proposed and partially developed a novel methodology for KBE application development based on *Model-Based Systems Engineering* (MBSE) [19]. This approach provides a structured framework for systematically capturing, developing, and maintaining KBE applications [18], with the ultimate goal of achieving automated round-trip engineering. Figure 1.2 presents a high-level visual representation of the MBSE approach, accompanied by a brief description below. For a detailed discussion of the methodology and its individual steps, the reader is referred to the dedicated literature.

1. **Knowledge Model**: Domain experts and KBE developers collaborate to create a knowledge model using SysML and MSoSA (Magic Systems of Systems Architect, now CATIA Magic[5]). This model includes a requirements package, a process package, and a product package. SysML diagrams like Block Definition Diagrams (BDDs) and Activity Diagrams (ADs) are used to capture the software structure and engineering workflows [20]. More detailed information on the knowledge model can be found in [21, 11], for a practical guide on SysML the reader is referred to [22].
2. **Automatic code generation**: The skeleton code of the KBE application is automatically generated using a Translation Engine [23]. This engine processes the knowledge model to create the KBE application architecture with all necessary classes, inputs, attributes, and part declarations.
3. **Manual code completion**: KBE developers *manually* complete the auto-generated code by adding domain-specific knowledge and operational logic. This step turns the skeleton code into a fully functional KBE application.
4. **Model reverse engineering**: The knowledge model is updated to reflect changes made during manual code completion. *Software Reverse Engineering* (SRE) methods are employed to automatically generate updated product and process models, with code-to-model (c2m) tools extracting model information directly from the KBE application. The work of [18] represents a first step toward this automation by demonstrating process model extraction from existing KBE applications, though complete knowledge model updates remain an ongoing challenge.

Recently, automation tools have been introduced to streamline the KBE development workflow. The Translation Engine [23] automates step (2) by generating skeleton code from knowledge models, while initial reverse engineering capabilities [18] enable partial automation of step (4) through process model extraction from existing applications. However, step (3) —manual code completion— remains entirely unautomated and represents the critical bottleneck in the development process.

Manual code completion is both the most time-intensive phase and the most dependent on specialized expertise for several interconnected reasons. First, developers must possess deep knowledge of the proprietary KBE modelling language, including its syntax, semantics, and idiomatic patterns. Unlike mainstream programming languages with extensive documentation and community support,

[5] https://www.3ds.com/products/catia/catia-magic

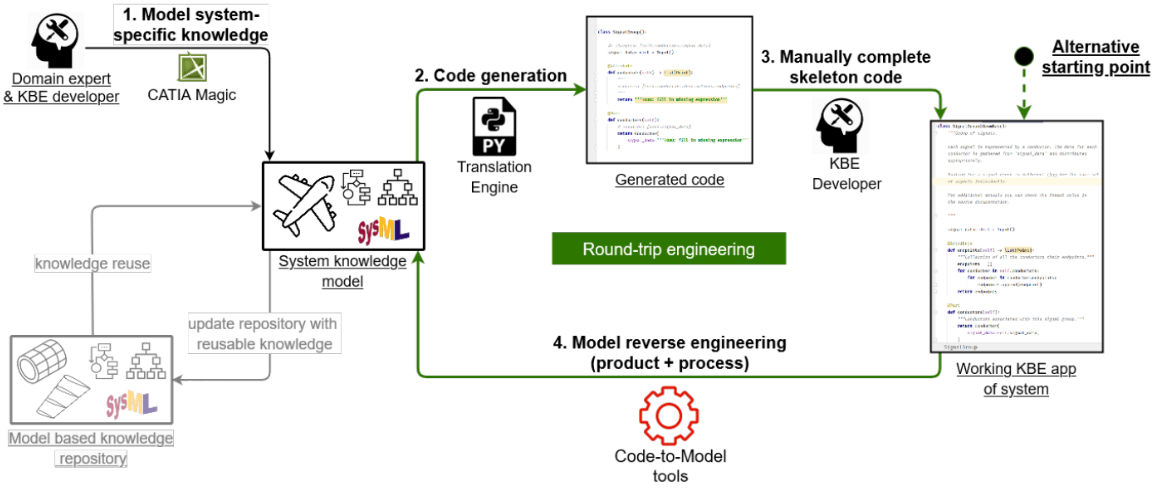


Figure 1.2: The envisioned MBSE driven approach developed during DEFAINE: the development of a new KBE application is divided into four mains steps, highlighted in bold text (taken from [18] which was adapted from [21]).

KBE frameworks like ParaPy have limited publicly available resources, steep learning curves, and framework-specific conventions that can only be mastered through extensive hands-on experience. Second, developers must simultaneously understand the domain-specific engineering knowledge—such as aerodynamic principles, structural constraints, or manufacturing requirements—and translate this knowledge into computational logic. This translation is non-trivial: engineering concepts expressed in natural language, equations, or heuristics must be reinterpreted as object-oriented code structures, dependency graphs, and parametric relationships within the KBE framework. Third, the iterative nature of KBE development compounds these challenges. As requirements evolve or edge cases emerge during testing, developers must refactor code while maintaining architectural coherence, managing dependencies, and preserving the semantic relationships between engineering concepts and their computational representations. This combination of framework-specific technical expertise and domain knowledge creates a significant barrier that limits development speed, increases costs, and restricts the scalability and broader adoption of KBE solutions.

While the MBSE-driven DEFAINE approach successfully addresses knowledge modelling and partial automation through structured methodologies, its inherently rigid, rule-based nature cannot accommodate the flexibility required for handling the variability and context-sensitivity of manual code completion. MBSE excels at ensuring adherence to requirements, standards, and formalized workflows—critical for validation and traceability—but lacks the adaptive capabilities needed to interpret diverse engineering contexts, generate context-appropriate code variations, and assist developers with varied expertise levels. For decades, this gap persisted because automating such complex and variable tasks was considered technically unfeasible [24].

Recent advancements in *Generative Artificial Intelligence* (GenAI) have fundamentally shifted this landscape. *Large Language Models* (LLMs), in particular, have demonstrated capabilities that directly address the limitations of structured approaches: pattern recognition across vast codebases, natural language understanding for translating engineering intent into code, and flexible generation that adapts to varying contexts and expertise levels [25]. These capabilities position LLMs as a complementary technology to MBSE, one that can maintain the rigour and traceability of structured methodologies while providing the adaptive flexibility required for effective code completion assistance in KBE development.

This research aims to bridge the identified gap by developing a GenAI-based virtual coding assistant, or "copilot", specifically tailored for KBE application development using the ParaPy modelling language. The proposed approach addresses the manual code completion bottleneck while tackling several interconnected challenges that complicate both KBE development and the integration of GenAI into this domain:

1. **Closed-source environments**: Many KBE modelling languages are proprietary, resulting in limited publicly available data for training open models (e.g., ChatGPT, Claude, Mistral, DeepSeek). This scarcity often leads to hallucinations[6] in KBE-specific coding suggestions, rendering general-purpose AI assistants ineffective for framework-specific development tasks.
2. **Domain-specific requirements**: KBE applications require developers to simultaneously navigate framework-specific programming constructs and domain engineering principles. General-purpose AI models lack access to both the proprietary framework documentation and the engineering design rationale necessary to generate contextually appropriate code.
3. **Steep learning curve and knowledge acquisition barriers**: The limited availability of learning resources, tutorial materials, and community support for proprietary KBE frameworks creates significant onboarding challenges for new developers. The absence of interactive, context-aware educational tools compounds these challenges, as developers must simultaneously master programming concepts, framework-specific patterns, and domain engineering principles without guided support.
4. **Local execution and privacy**: Protecting both the closed-source software used in KBE development and the proprietary engineering data involved is a critical concern. Intellectual property considerations, non-disclosure agreements, and competitive sensitivity often prohibit the use of external API-based AI services, necessitating locally deployable solutions that maintain full data sovereignty.

These challenges collectively highlight the need for specialized AI-assisted development tools that combine framework-specific knowledge, domain engineering expertise, and privacy-preserving deployment capabilities. Recent literature confirms that GenAI-based code generation is an active and evolving area of research. A comprehensive overview of current advancements is provided in [26]. Notable developments include the coordination of large language models (LLMs) by AI agents with specialized roles for code generation tasks [27], as well as Meta's proprietary AI-assisted coding toolchain [28]. These developments, along with the broader evolution of Generative Artificial Intelligence and language modelling in particular, will be discussed in detail in the formal literature review (Chapter 3). At the outset of this thesis project (March 18, 2025), no publicly available application was identified that integrates generative AI approaches with KBE[7].

The author explicitly notes the date of this literature review, as the field of (Generative) Artificial Intelligence is evolving at an unprecedented pace (see Chapter 3), making it increasingly challenging to stay up to date. Nevertheless, this research is expected to provide lasting value, particularly in the domains of Knowledge-Based Engineering (KBE) and Aerospace Engineering. High-tech industries, despite their reputation for producing high-quality products, often face challenges in adopting new technologies promptly [29]. This lag is not only due to the large scale of their operations but also the highly regulated nature of their environments. The proposed copilot tool seeks to bridge this gap by providing a valuable solution tailored specifically for industry adoption in said environments.

This research, alongside recent works by [23, 18], aspires to pave the way for deeper AI integration in KBE, potentially evolving toward a fully automated, human-in-the-loop, round-trip engineering approach. Similar efforts are already emerging in industry [25]. However, this work will remain focused on assisting developers, aligning with the prevailing consensus in academia and government that GenAI should support rather than replace human expertise [30, 31, 32, 33]. Although this approach may seem cautious, the vast amounts of data available in aerospace and high-tech engineering suggest immense potential for data-driven methodologies. In particular, Generative Artificial Intelligence holds promise for reshaping the field, an idea explored in detail in Chapter 3.

[6]Hallucination refers to the generation of false, misleading, or nonsensical information that appears plausible but does not represent real data or facts. This occurs when the model confidently produces incorrect responses due to gaps in its training data or the way it generalizes information

[7]Based on a Scopus search on 17-03-2025 using the following search query: "((large AND language AND models OR gen*ai) AND *engineering AND ((coding AND assistant) OR copilot OR parapy))"

2
Research Framework

The present chapter builds upon the research gap outlined in the Introduction by consolidating it into tangible research objectives and questions, while also defining the scope of the study. Based on the established objectives and corresponding research questions, a set of requirements is formulated to guide the subsequent phases of the research—namely, the literature review, system design, and evaluation.

To clarify the direction of this research, this chapter introduces the research objectives in section 2.1, formulates the main research questions in section 2.2, and defines the requirements for the intended support in section 2.4, alongside a delineation of the study's scope. The research follows four stages inspired by the Design Research Methodology (DRM) as described by [34], which are further detailed in section 2.5. Finally, key definitions essential for understanding the requirements and the remainder of this thesis are provided in section 2.6.

2.1. Objectives

This AI-assisted framework directly addresses the manual code completion bottleneck in step (3) of the MBSE-driven KBE development process. This bottleneck arises from several interconnected challenges: the proprietary nature of the ParaPy SDK limits access to public learning resources and onboarding support; developers must simultaneously grasp both framework-specific constructs and domain-specific engineering knowledge, often without sufficient guidance; and the translation of engineering concepts into executable code is time-consuming and technically demanding.

To mitigate these challenges, the framework integrates intelligent code completion and generation with contextual explanations and ParaPy-specific guidance. It is designed to lower the entry barrier for developers of varying expertise, accelerate development workflows, and support long-term skill acquisition—while ensuring code quality and functional correctness are preserved.

The key objectives of this research project are:

1. To develop a locally-run, Generative AI-powered coding assistant tailored for Knowledge-Based Engineering (KBE) application development using the ParaPy SDK.
2. To ensure the AI-generated code meets a minimum quality threshold, defined as syntactically correct, readily executable, and functionally aligned with user intent.
3. To define and measure key performance indicators (e.g. development time, code quality, and knowledge requirements) relevant to KBE development productivity.
4. To evaluate the assistant's effectiveness across users with varying levels of expertise (novices vs. experts).

2.2. Main Questions

To tackle each research objective, the following main research questions have been formulated:

1. How can a (locally-run) Large Language Model be employed to support ParaPy KBE application development?
2. What are the minimum quality standards that AI-generated ParaPy code must meet to be considered usable and production-ready?
3. What metrics can be used to rigorously evaluate performance improvements in KBE application development with AI assistance?
4. How effective is the GenAI coding assistant in improving KBE application development performance, and how does this effectiveness vary between novice and expert users?

2.3. Scope

This research focuses on developing and evaluating an AI-assisted coding framework specifically for ParaPy SDK development, operating within the constraints of industrial privacy requirements and proprietary framework limitations. The primary aim for the framework is to develop a KBE application starting from skeleton code—partially completed source code structures that outline high-level components without full implementation (as defined in section 2.6). This approach leverages auto-generated skeleton code produced by the SysML-to-Python Translation Engine [23] as a starting point, with the extended capability to generate complete class definitions in the absence of such skeleton code.

Explicitly excluded from the current research scope are:

- Creation of novel LLM architectures or comprehensive training of models from scratch
- Fine-tuning of language models on ParaPy-specific datasets
- Development of sophisticated graphical user interfaces or visual programming environments beyond the implemented CLI application
- IDE integration or inline code suggestion mechanisms
- Web-based deployment platforms or cloud infrastructure beyond API provider integration

2.4. Requirements

Based on the research questions, the following preliminary requirements have been defined. These requirements serve as the basis for evaluating whether the developed support contributes effectively to solving the core problem—namely, the time- and knowledge-intensive nature of KBE application development. The requirements will be used during the verification and validation phases (Chapter 7) to assess whether the proposed solution meets its intended goals, either fully or partially. All key terms are underlined and formally defined in section 2.6. The structure used for requirement coding is as follows:

$$\text{REQ}^{i} - X^{ii} - Y^{iii} \quad \begin{cases} i: & \textit{Denotes a requirement} \\ ii: & \textit{Refers to the primary associated research question (subclass)} \\ iii: & \textit{Indicates the requirement number within the subclass} \end{cases}$$

> **REQ-1-1** : The copilot tool shall support both local LLM deployment (via Ollama or similar frameworks) and remote LLM inference through privacy-compliant API services (e.g., OpenAI, Anthropic, or other approved providers).

> **REQ-1-2** : The copilot tool shall be capable of completing auto-generated skeleton code produced by the SysML-to-Python Translation Engine [23] *[minimum]*;

> **REQ-1-3** : Building upon **REQ-1-2**, the copilot tool shall be capable of generating class definitions even in the absence of skeleton code *[extended]*;

> **REQ-2-1** : AI-generated ParaPy code shall exhibit syntactic correctness;

> **REQ-2-2** : AI-generated ParaPy code shall exhibit runtime correctness;

> **REQ-2-3** : AI-generated ParaPy code shall exhibit functional correctness;

> **REQ-3-1** : Development time shall be the primary metric used to assess performance improvements introduced by the copilot tool in KBE application development;

> **REQ-3-2** : AI-generated code shall be 100% functionally identical to manually completed, correct code;

> **REQ-3-3** : AI-generated code shall receive an expert-assigned grade reflecting its semantic correctness, maintainability, and adherence to PEP-8 standards [35];

> **REQ-3-4** : A composite score, incorporating development time, functional correctness, and the expert-assigned quality grade, shall serve as a secondary metric to assess the overall performance improvement delivered by the copilot tool;

> **REQ-4-1** : The use of the copilot tool shall reduce the total development time of a KBE application for both novice and expert users;

> **REQ-4-2** : The use of the copilot tool shall increase the composite score (as outlined in **REQ-3-4** with respect to manually completed code, for both novice and expert users;

> **REQ-4-3** : The use of the copilot tool by novice users shall reduce the knowledge required for KBE application development.

2.5. Structure

This research follows a systematic approach inspired by Design Research Methodology [34], progressing through four main stages that structure the thesis chapters:

1. The first stage establishes the research foundation. This chapter, together with Chapter 1, defines the research goals, questions, and scope, while formulating preliminary success criteria that guide subsequent phases.
2. The second stage deepens understanding of the problem domain through literature review and empirical analysis. Chapter 3 and Chapter 4 examine existing knowledge and current practices, identifying key factors that inform the design solution and refining the evaluation criteria.
3. The third stage translates this understanding into a practical solution. Chapter 5 and Chapter 6 present the systematic design and implementation of the AI-assisted coding framework, including its architecture, components, and deployment considerations.
4. Finally, the fourth stage evaluates the developed solution. Chapter 7 assesses both the framework's usability and its effectiveness in achieving the intended improvements, while Chapter 8 and Chapter 9 synthesize findings and propose directions for future work.

The next chapter, Chapter 3, provides the literature review that maps relevant domains, existing knowledge, and theoretical foundations supporting this research.

2.6. Definitions

1. **Skeleton Code** refers to a partially completed source code structure that outlines the high-level components of a program, module, or class without implementing full functionality. It typically includes declarations of functions, classes, parameters, and control flow placeholders (e.g., `pass` statements or `TODO` comments) that guide further development. Skeleton code serves as a template or framework for developers to fill in with detailed logic and implementation.

   An example of skeleton code from the Translation Engine [23]:

```
class Aircraft(GeomBase):
    #: :targets: [self.wing_span]
    max_range: float = Input()

    @Attribute
    def wing_span(self) -> float:
        """

```

```
        :sources: [self.max_range]
        """
        return '''TODO fill in missing expression'''

    @Part
    def wing(self) -> ExtrudedSolid:
        """

        :sources: [self.root_airfoil.position]
        """
        return '''TODO fill in missing expression'''
```

2. A **Class Definition** is a syntactic construct in object-oriented programming that declares a new class, specifying its name, attributes (also called fields or properties), methods (also called functions), and potentially other class-level elements such as inheritance, decorators, and documentation. It serves as a blueprint for creating instances (objects) of that class.

   In Python, using the ParaPy SDK, a class definition typically follows this structure:

```
class ClassName(Base):
    some_number: float = Input()
    some_text: str = Input()
    some_points: list[Point] = Input()

    @Attribute
    def some_attribute(self) -> dict[Point, float]:
        ...
        return a_dictionary

    @Part
    def some_geometric_part(self) -> GeomBase:
        ...
        return a_solid

    def do_something(self) -> None:
        """Regular python method."""
        ...
        return None
```

3. **Syntactic correctness** refers to whether the code adheres to the formal grammar rules of the programming language (in this case, Python and the ParaPy SDK). A syntactically correct code snippet can be parsed by the interpreter without producing syntax errors, and includes valid use of language constructs such as indentation, brackets, colons, and statement structure.

   Example failure:

```
def foo() print("hello")
```

   Which raises:

```
Traceback (most recent call last):
  File "path\to\file", line 1
      def foo()
               ^
SyntaxError: expected ':'
```

4. **Runtime correctness** indicates that the code, once syntactically valid, executes without encountering unhandled errors during its run. This includes the successful resolution of variable names, API calls (e.g., ParaPy-specific constructs), and handling of runtime conditions like division by zero, file access, or invalid input.

   Example failure:

```
from parapy.core import Input

class Foo(Base):
    bar: int = Input()
```

   Which raises:

```
Traceback (most recent call last):
  File "path\to\file", line 3, in <module>
    class Foo(Base):
              ^^^^
NameError: name 'Base' is not defined.
```

5. **Functional correctness** refers to whether the code performs the task or behavior intended by the user or defined in the task specification. This includes correct use of logic, accurate integration of API elements (ParaPy SDK), and the production of expected outputs or model behaviors.

   Example failure: the user prompt specifies the generation of a parametric model of an aircraft, while the AI-generated code represent the model of a ship.

6. **Semantic correctness** refers to whether the code is logically sound and type-consistent beyond mere syntactic validity. This encompasses two key dimensions: (1) type correctness—whether variables, function arguments, and return values maintain consistent types throughout the code, and (2) logical correctness—whether the code is free from logical flaws such as unreachable code blocks, undefined variables, incorrect control flow, or improper API usage patterns. Semantically correct code may execute without errors but still contains logical inconsistencies or type mismatches that could lead to unexpected behaviour or maintenance challenges. Semantic correctness can be assessed through static analysis techniques that detect type errors and logical inconsistencies.

7. **Maintainability** refers to the ease with which code can be understood, modified, extended, and debugged by developers over time. Maintainable code exhibits clear structure, manageable complexity, readable logic, and adherence to established coding conventions. High maintainability reduces long-term development costs and facilitates collaboration among team members. Maintainability can be quantitatively measured using complementary metrics that assess code complexity and structure [36, 37]:

   (a) **Maintainability Index (MI)**: A composite metric ranging from 0 to 100 that combines measures of computational complexity, control flow complexity, and code volume. Higher MI values indicate more maintainable code.

   (b) **Cyclomatic Complexity (CC)**: A metric that quantifies the number of linearly independent paths through a program's source code by counting decision points. Lower CC values indicate simpler, more maintainable code.

3

Literature Review

This chapter examines the theoretical foundations and contemporary developments in Artificial Intelligence (AI) and Large Language Models (LLM) that are relevant to Knowledge-Based Engineering (KBE) applications. The review establishes the technical context necessary for understanding the industrial investigation in Chapter 4 and informs the design decisions presented in Chapter 5 and Chapter 6.

The chapter is organized into four main sections. Section 3.1 traces Artificial Intelligence from symbolic expert systems through neural networks to Transformer-based language models. Section 3.2 examines code generation with language models, focusing on challenges in domain-specific frameworks with limited training data. Section 3.3 surveys four design paradigms for LLM applications: prompt engineering, retrieval-augmented generation, fine-tuning, and agentic systems, analysing their strengths, limitations, and use cases. Finally, section 3.4 synthesizes these findings and discusses implications for AI-assisted KBE tools.

By the end of this chapter, readers should understand the fundamental concepts underlying modern AI systems, the specific considerations for applying language models to code generation tasks, and the design paradigms available for building LLM-powered applications. This foundation enables the refinement of preliminary requirements from Chapter 2 into measurable success criteria and informs the architectural decisions in subsequent chapters. Readers seeking a more comprehensive treatment of language modelling foundations, including statistical approaches, sequence models, and other neural architectures, are referred to Chapter 10.

Notion on the Use of ArXiv

The author acknowledges that articles published on arXiv are not peer-reviewed by independent sources. However, arXiv remains highly relevant in the field of Generative AI, serving as a primary publication platform for cutting-edge research. Major industry research laboratories frequently release preliminary findings on arXiv, particularly for foundational model architectures and training methodologies, though publication patterns vary significantly among organizations. While some companies like Meta and Google DeepMind continue to publish extensively on arXiv, others increasingly utilize proprietary publication channels or selective academic venues for their most significant developments.

3.1. Foundations of Artificial Intelligence

Artificial Intelligence (AI) is broadly defined as the science and engineering of creating machines that exhibit intelligent behavior (e.g., performing tasks that typically require human intelligence) [38]. The field's formal inception is often traced to the Dartmouth Workshop of 1956, where the term "Artificial Intelligence" was coined and AI was established as an academic discipline [39]. Early AI research in the subsequent decades was dominated by symbolic reasoning and rule-based expert systems that encoded domain knowledge explicitly. This era produced Expert Systems in the 1970s–1980s: software that emulated the decision-making of human experts in narrow domains using hand-crafted if-then rules and logical inference. A general timeline of computer technology and associated advances in AI is illustrated in Figure 3.1.

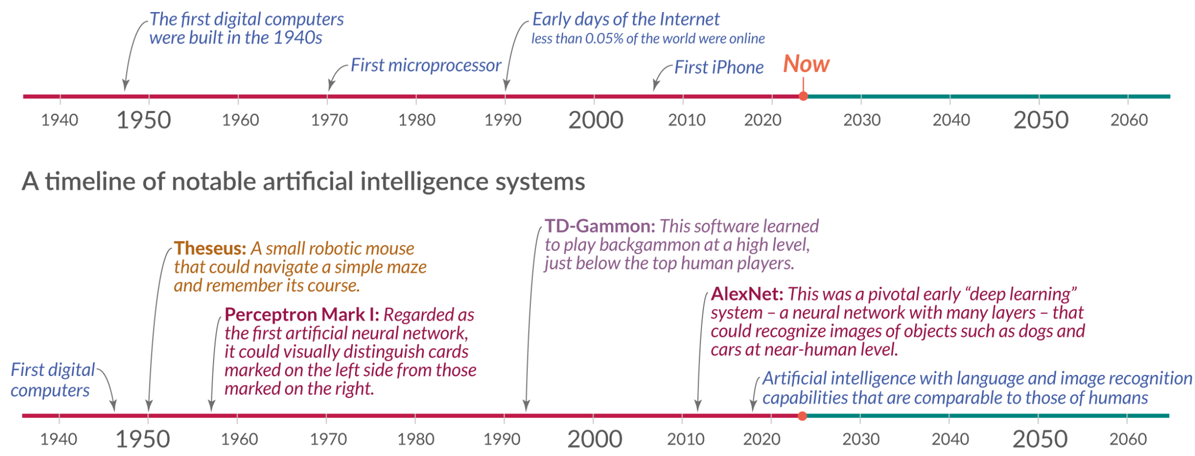


Figure 3.1: A timeline of the evolution of computer technology and the associated advances in AI [40].

Knowledge-Based Engineering (KBE) emerged directly from this expert systems paradigm as an application of AI techniques to engineering design problems. Rather than developing as an independent methodology, KBE represents the adaptation and specialization of AI's rule-based reasoning approaches to capture and automate engineering knowledge [13]. As such, KBE can be understood as a domain-specific instantiation of symbolic AI, inheriting both its architectural principles and philosophical approach. Both KBE and classical expert systems rely fundamentally on explicit knowledge representation – encoding domain expertise as rules, objects, facts, and logical relationships – to automate reasoning and decision-making in specialized domains [13]. In KBE specifically, these AI techniques are employed to capture design rationale, engineering rules, and geometric relationships, enabling the automation and reuse of engineering design knowledge through knowledge models and rule-based systems.

This shared heritage with rule-based AI explains both KBE's strengths and its limitations. Such rule-based AI approaches proved powerful within their scope, but they suffered from the notorious "knowledge acquisition bottleneck" [41, 42]: it was difficult and labour-intensive to extract, formalize, and maintain the vast amount of tacit expertise that humans use, making these systems brittle and hard to scale [43]. They also struggled to adapt to novel scenarios outside their encoded knowledge. KBE systems, being grounded in these same AI principles, face identical challenges in knowledge elicitation, maintenance, and generalization [17].

By the 1990s and 2000s, AI began shifting from purely rule-based systems toward *machine learning* (ML) approaches. ML is a subfield of AI that gives computers the ability to learn from data and improve through experience, rather than relying on explicit programming of every rule [38]. Instead of engineers hard-coding knowledge, ML algorithms automatically infer patterns and decision logic from large datasets. This paradigm shift helped overcome the knowledge bottleneck by letting models learn complex behaviors from examples. Machine learning encompasses a variety of methods (e.g. decision trees, support vector machines, neural networks) and has become the dominant approach for AI in practice. In fact, within AI, data-driven learning techniques emerged as the method of choice for tasks like computer vision, speech recognition, natural language processing, and robotics [44]. Key subdomains of AI include:

- knowledge-based systems (expert systems, KBE techniques),
- machine learning (data-driven statistical learning algorithms), and within ML, deep learning using neural networks,
- other specialized areas such as computer vision, natural language processing, and robotics.

3.1.1. Artificial Neural Networks

One of the most influential approaches in modern AI is the use of artificial neural networks (ANNs): computational systems inspired by the structure and function of biological neural networks. At their core, ANNs aim to learn the underlying patterns or structures that govern complex, often non-linear problems, particularly those for which explicit rules are difficult to define or encode manually (natural language as an example). An ANN is composed of many simple, connected processing units known as artificial neurons, organized into layers. Each neuron receives one or more input values, applies a learnable weight to each of them, adds a bias term, and passes the result through a non-linear activation function. This output then serves as input to the next layer of neurons. The use of weights and biases allows the network to represent and adjust the importance of different input features, while the activation function introduces non-linearity—crucial for learning complex patterns beyond what linear models can represent. [44]

Neurons are typically structured into an input layer, one or more hidden layers, and an output layer. A classic example of this architecture is the Multilayer Perceptron (MLP), as seen in Figure 3.2: a type of feed-forward neural network in which information flows strictly in one direction: from input to output, via the hidden layers [44]. In most MLPs, each neuron in a layer is fully connected to all neurons in the subsequent layer. Despite its relative simplicity, the MLP forms the foundation for many more advanced neural architectures, such as the Transformer architecture.

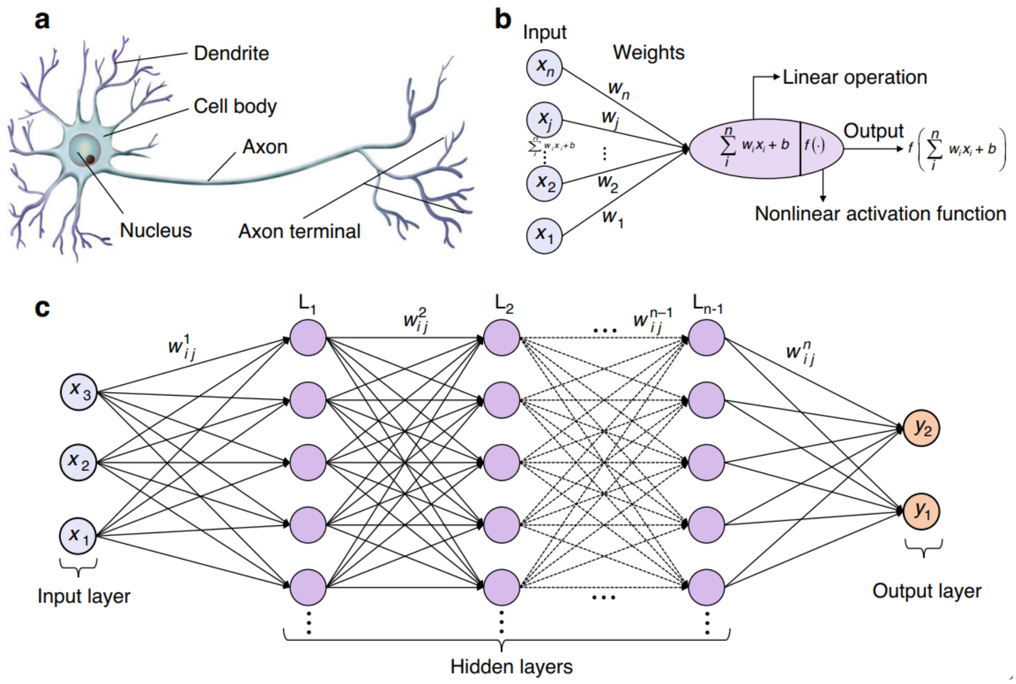


Figure 3.2: Overview and biological analogy of an artificial neuron and structure of an artificial neural network: (a) structure of biological neuron; (b) overview of an artificial neuron within a MLP with its input, weights, activation function and output; (c) structure of an Multilayer Perceptron. [45]

Training a neural network such as an MLP involves adjusting its internal weights and biases to minimize the discrepancy between the network's predicted outputs and the true values (i.e., the error), a typical optimization problem encountered in engineering as well. This is most commonly achieved using the backpropagation algorithm, which efficiently computes the gradient of the loss function with respect to each weight and bias. These gradients are then used in an optimization algorithm—typically stochastic gradient descent (SGD) or one of its variants—to iteratively update the parameters in a way that reduces the overall error [46]. Through repeated exposure to large datasets, the network gradually "learns" to approximate the function that maps inputs to desired outputs. When a neural network contains multiple hidden layers, it is referred to as a deep neural network, and the corresponding approach is known as deep learning. The depth of such networks enables them to extract increasingly abstract features at each layer, ranging from simple edges in early layers of an image recognition network to complex patterns such as objects or semantics in later layers.

The remarkable flexibility and power of deep learning stem from this ability to automatically discover hierarchical representations of data. This has led to significant breakthroughs in areas such as image classification, speech recognition, and natural language processing. However, this power comes at a cost: deep networks often operate as "black boxes", where knowledge is distributed across millions of parameters rather than expressed in interpretable, symbolic rules. This opacity can be problematic in domains that require explainability or verifiability. Nevertheless, the strength of neural networks, particularly deep ones, lies in their ability to uncover subtle, high-dimensional relationships in vast, unstructured datasets, a capability that traditional rule-based approaches struggle to match [44].

3.1.2. Recent Developments: Transformers and Large Language Models

The last decade has seen rapid advancements in AI, marked by the rise of *Generative AI* (GenAI) and extremely large-scale models. Generative AI refers to systems that can generate novel content (text, images, code, etc.) by learning patterns from vast amounts of data. A pivotal breakthrough enabling this shift was the introduction of the *Transformer architecture* in 2017 [47]. The Transformer uses a self-attention mechanism that allows each token in a sequence to directly attend to all other tokens, regardless of distance, enabling the model to efficiently capture long-range dependencies in data. Unlike previous recurrent architectures that processed sequences step-by-step, Transformers can process all tokens in parallel, making them vastly more efficient to train. However, Transformers operate within a fixed context window and maintain no persistent memory across separate interactions, each inference call is stateless. This architectural characteristic makes prompt engineering essential: all relevant information, instructions, and context must be explicitly included in the input prompt to guide the model toward correct outputs [48, 49, 50]. The Transformer rapidly became the foundation for modern language models, with adaptations like BERT[1] (for understanding tasks) and GPT[2] (for generation tasks) emerging within a year of the original architecture's introduction [51, 52]. A detailed treatment of the evolution from statistical language models through recurrent architectures to Transformers is provided in Chapter 10 for readers interested in the technical foundations.

Since the transformer's debut, progress in AI has accelerated dramatically. Model sizes have grown exponentially. For instance, OpenAI's GPT-3 language model contains 175 billion parameters and demonstrated unprecedented capabilities, such as fluent paragraph-length text generation and simple reasoning, that were not observed in smaller predecessors [48]. Notably, GPT-3 showed emergent abilities like few-shot learning. In just a few years, the field moved from tens of millions of parameters to hundreds of billions (and, as of 2023, trillion-parameter models), accompanied by a corresponding leap in performance and versatility. The pace of AI development has become increasingly rapid: new state-of-the-art *foundation models* are now released on a monthly cadence [53]. These models are trained on enormous datasets using massive computational resources; in fact, the compute (the required computational resources) required for training frontier models has been doubling roughly every 5–6 months in recent years [53]. This combination of algorithmic innovation, big data, and big compute has yielded AI systems with strikingly advanced capabilities, as seen in Figure 3.3.

Today's generative AI systems are capable of complex language understanding and generation, creative content production, and non-trivial reasoning and problem-solving. In software engineering, LLMs have demonstrated strong performance as coding assistants. OpenAI's Codex, a descendant of GPT-3 trained on programming data, translates natural language descriptions into working code across various programming languages [54]. It powers GitHub Copilot, which assists developers by autocompleting functions and suggesting context-aware snippets. DeepMind's AlphaCode pushed these capabilities further by generating and evaluating code solutions for competitive programming problems, achieving mid-level human performance on Codeforces challenges [55]. Similarly, open-source models such as Salesforce's CodeGen and Meta's InCoder have expanded the accessibility of these tools beyond proprietary platforms.

As noted in Chapter 1, recent developments suggest a complementary role for LLMs within traditional Knowledge-Based Engineering workflows. Instead of relying exclusively on upfront modelling of

[1]Bidirectional Encoder Representations from Transformers
[2]Generative Pre-trained Transformer

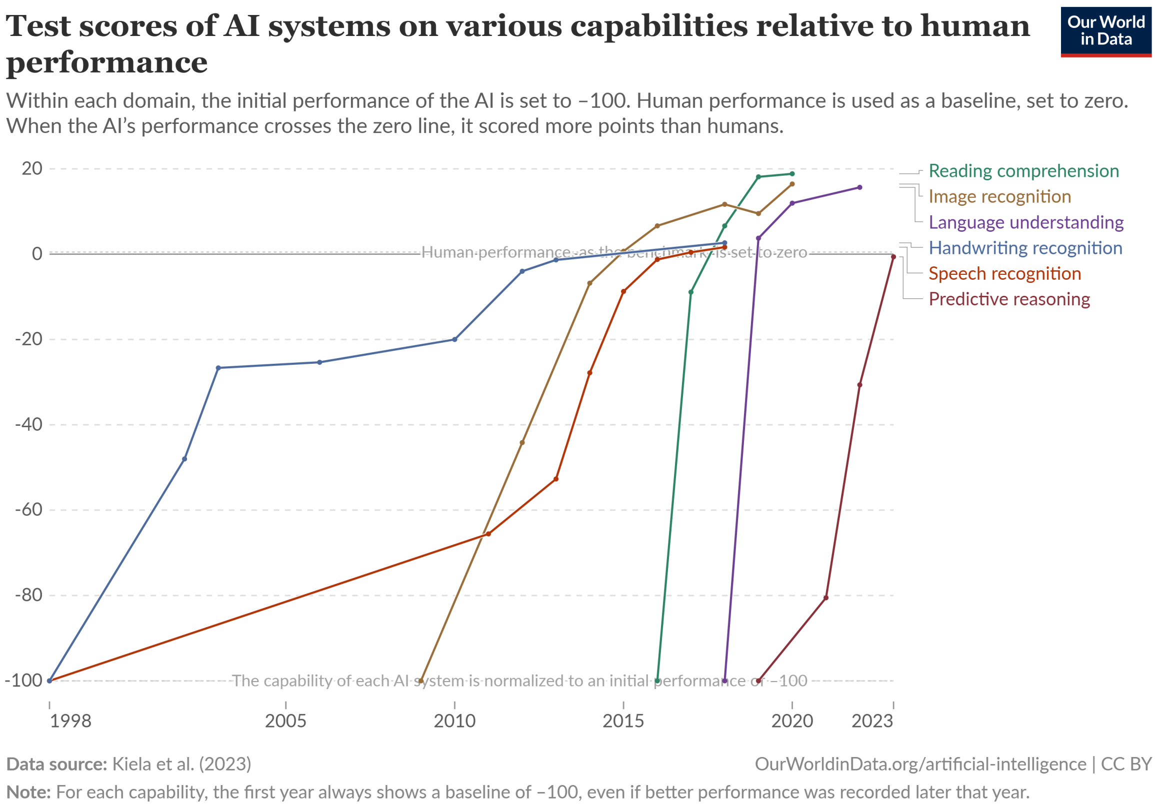


Figure 3.3: AI systems performance on various capabilities relative to human performance, from 1998 to 2025 [40].

domain rules, engineers can increasingly use LLMs to generate or refactor code and constraints, retrieve and explain prior design cases [18], and integrate up-to-date standards or design rationale into their development process. However, rule-based systems remain indispensable wherever determinism, explainability, and traceability are critical, such as in the MBSE contexts [53, 50]. As generative models gain wider adoption across engineering disciplines, a deeper understanding of their underlying mechanisms—particularly the language modelling objectives that shape their behaviour—will be essential to fully harness their potential.

3.2. From Natural Language to Programming

Programming languages are formal languages, but they share many characteristics with human language: a vocabulary (keywords, identifiers), syntax rules (grammar), and even stylistic conventions [35]. It is therefore natural to apply language modelling techniques to source code. Indeed, code can be treated as just another form of text, and many LLMs have been trained or fine-tuned on large code corpora (such as open-source GitHub repositories) in addition to natural language data [54]. By learning from millions of code examples, an LLM can statistically infer how to write syntactically correct and plausible code to accomplish a given task described in natural language. For instance, an LLM-based system can take a prompt like "Implement a Python function to check if a number is prime" and generate a complete function in Python to do so. This ability has huge practical implications: it can automate boilerplate coding, assist in software development, and enable conversational interfaces for programming.

One key difference between natural language and programming is that code execution is unforgiving to errors. A grammatical mistake in an English sentence might still be understood by a reader, but a syntax error in code will prevent it from running. Likewise, a slightly off factual statement in prose might go unnoticed, but a single incorrect API call in code can cause a program to crash. Thus, *adherence to syntax and semantics* is paramount in code generation. LLMs generally have excellent syntax adherence for languages seen frequently in training (such as Python, JavaScript, etc.), often producing code that compiles or runs on first try. This is evidenced by high scores on competitive coding benchmarks achieved by state-of-the-art models like GPT-4 or Anthropic's Claude [54]. However, ensuring semantic correctness (that the code does what it's supposed to do) is a harder challenge. Models may produce logically flawed or inefficient code that superficially looks right. There is ongoing research on evaluating functional correctness via test suites and on making LLMs not just syntactically but semantically aligned with user intent.

A particularly challenging scenario for code LLMs is dealing with unfamiliar or under-represented libraries and frameworks [56, 57]. LLMs do not possess true "knowledge" of APIs but rather learn statistical patterns from training data—public libraries with abundant training examples yield better results, while models struggle with rare or proprietary code unless fine-tuned on it [28]. Without retrieval mechanisms at generation time (unless augmented with external tools, see subsection 3.3.2), LLMs are prone to *hallucinate* when encountering unfamiliar libraries: fabricating plausible but incorrect function names, class definitions, or usage patterns [58, 59, 56, 60, 61]. This problem is particularly acute for closed-source, domain-specific frameworks like ParaPy: because such code is largely absent from public repositories, the model may invent non-existent SDK components that sound reasonable but fail to compile or execute [57, 56]. Studies confirm that LLMs frequently produce answers rather than abstain when uncertain, and that hallucinations increase significantly without codebase-specific knowledge [62, 63, 64, 61, 28]. Mitigating this requires techniques beyond standard training, such as retrieval-augmented generation (providing relevant documentation or code snippets in the prompt) or fine-tuning on the specific library [58, 65, 28, 56].

3.3. Design Paradigms for LLM-Based Applications

The practical deployment of Large Language Models in real-world applications requires careful consideration of how to optimize their performance for specific tasks and domains. While pre-trained LLMs demonstrate remarkable general capabilities, translating these capabilities into effective, reliable, and scalable applications presents distinct engineering challenges that can be addressed through four complementary paradigms:

1. **Prompt engineering and orchestration techniques** operate at inference time, leveraging existing capabilities through carefully designed inputs and multi-step workflows.
2. **Retrieval-augmented generation and tool integration** extend these capabilities by connecting models to external knowledge sources and computational resources.
3. **Fine-tuning approaches** modify model parameters to specialize behaviour for particular domains or tasks.
4. **Agentic and multi-agent systems** coordinate multiple LLM instances to handle complex, autonomous problem-solving scenarios.

These paradigms are not mutually exclusive but rather represent a toolkit of strategies that can be combined based on specific application requirements, resource constraints, and performance objectives. Understanding their respective strengths, limitations, and optimal use cases is essential for practitioners seeking to build robust LLM-powered systems.

3.3.1. Prompt Engineering and Orchestration

Prompt engineering uses carefully crafted input instructions to guide a pre-trained LLM's behaviour without altering its weights [66]. This approach leverages the model's latent knowledge and few-shot learning abilities: for example, chain-of-thought prompts can induce step-by-step reasoning, enabling complex problem solving without fine-tuning [67]. Orchestration extends simple prompting into multi-step workflows where applications decompose tasks into sequential prompts or role-based dialogues, maintaining conversational context across turns to "plan and execute" sub-tasks in stages [68]. These strategies require no model retraining and are highly flexible, developers can rapidly prototype new behaviours by adjusting prompts or workflow steps [67].

The major advantage of prompt engineering lies in its accessibility: it operates at inference-time, avoiding the cost and data requirements of model training, works with both closed APIs and open models, and easily combines with human-in-the-loop iteration [67]. However, it faces inherent limitations. LLMs have fixed context windows that cannot accommodate arbitrarily large instructions or knowledge, causing complex tasks to exceed these limits [69]. Prompting alone can lead to hallucinations or inconsistent outputs since the model remains a generalist rather than a domain specialist, and result quality is highly sensitive to prompt wording and ordering [69]. For example, in clinical text classification, GPT-4 prompted with clear instructions (including reasoning steps) matched specialist models without additional fine-tuning [70], demonstrating prompt engineering's effectiveness for bounded tasks. However, very domain-specific or large-scale applications may require additional techniques.

In practice, orchestration frameworks implement patterns like REACT (Reasoning and Acting) [71], where the model first reasons about what action to take, then outputs a formatted action that the application executes, and finally receives the result to continue the dialogue. This approach is common in conversational assistants and applications where interpretability (seeing the model's intermediate reasoning) is valuable. Software engineering assistants, for instance, use prompts to adopt specific personas and structured output formats for code suggestions, enabling quick adaptation of general LLMs to specialized query styles.

3.3.2. Retrieval-Augmented Generation and Tool Use

While prompt engineering provides a foundation for LLM interaction, many applications require access to information beyond what can be included in the model's context window. This limitation motivates the second paradigm: retrieval-augmented generation and tool integration.

Retrieval-Augmented Generation (RAG) addresses fundamental LLM limitations - knowledge cut-offs, hallucinations, and fixed context windows – by connecting models to external information sources [72]. The approach is straightforward: when a user submits a query, the application searches an external knowledge base (such as documentation, code repositories, or specialized databases) for relevant information. These retrieved passages are then inserted into the prompt as additional context, allowing the LLM to generate responses grounded in current, domain-specific information rather than relying solely on its training data [72]. For instance, a coding assistant might search a project's documentation when generating code, dynamically providing the model with up-to-date API references and usage examples. By shifting knowledge storage from model parameters to searchable external sources, RAG enables continuous updates without retraining and reduces factual errors by grounding outputs in retrieved evidence [69].

Beyond retrieving information, LLMs can actively invoke tools to extend their capabilities, calling APIs, executing code, performing calculations, or interacting with external systems [59]. Function-calling frameworks enable models to output structured commands (typically JSON) that trigger external operations, with results returned to continue generation. For example, OpenAI's function-calling API allows developers to register tools that GPT-4 can autonomously invoke by generating appropriate JSON specifications [71]. Standardization efforts like Anthropic's Model Context Protocol (MCP) aim to provide universal interfaces for tool integration, reducing the custom integration work required for each new capability [73]. This tool-use paradigm allows language models to offload specialized computations (mathematics, database queries, code execution) to dedicated systems while focusing on natural language reasoning and coordination.

Retrieval and tool augmentation are inference-time strategies that enhance LLM performance on knowledge-intensive and interactive tasks [72]. They excel when prompts alone are insufficient: question-answering over proprietary documents, maintaining current world knowledge, providing up-to-date API references for code generation, or solving tasks requiring external actions. By grounding outputs in retrieved facts, RAG reduces hallucinations and improves response credibility [72]. By delegating deterministic operations to tools, the approach lets models focus on reasoning while offloading computational tasks to appropriate external modules [59]. However, these benefits come with added complexity – managing knowledge bases, implementing retrieval pipelines, handling tool execution errors – and dependency on retrieval quality: incomplete knowledge bases or failed searches can still produce incorrect answers. When combined with robust prompting, RAG and tool-use have nonetheless proven effective across diverse practical applications [72].

3.3.3. Fine-Tuning and Customization

Inference-time approaches, such as prompting and retrieval, offer flexibility and enable rapid deployment, but certain applications demand deeper specialization that can only be achieved through model parameter adaptation. This consideration motivates an exploration of fine-tuning and other customization techniques.

Finetuning involves taking a pre-trained LLM and further training it on domain-specific or task-specific data. Unlike prompting or RAG (which operate at inference time), finetuning updates the model's weights during an additional training phase so that the model internalizes new knowledge or behaviour. This can be full fine-tuning (all parameters updated) or parameter-efficient methods (updating only small adapters like LoRA[3] layers). Fine-tuning was the traditional approach to get a language model to perform a new task: e.g. training GPT-3 on medical texts to create a specialized clinical assistant. It remains a powerful paradigm: fine-tuned LLMs can outperform prompt-based use of a generic model, especially when the task requires nuanced domain expertise or formatting that the base model was never exposed to [70]. For instance, open-source models like LLaMA-2 (Meta) have been fine-tuned on instruction-following datasets to align them with user instructions to produce LLaMA-2-Instruct, and further fine-tuned variants exist for coding (CodeLLaMA), medicine, law, etc., achieving high accuracy on those specialized benchmarks.

The strength of fine-tuning lies in specialization: by updating model parameters on domain-specific data, the model encodes task-specific features that often yield more accurate and consistent outputs than zero-shot prompting [70]. Fine-tuning is particularly valuable for structured output formats, policy adherence, and improving smaller models that lack the emergent capabilities of frontier systems. However, it carries significant costs: labelled datasets, computational resources, and ongoing maintenance when base models or requirements change. Fine-tuned models risk overfitting to narrow data, may lose general knowledge, and lack the task-switching flexibility of prompt-based approaches. Additionally, maintaining multiple domain-specific models creates MLOps[4] complexity, and even specialized fine-tuned models can underperform larger general models in reasoning tasks [70].

Fine-tuning is typically employed when high accuracy on specific tasks is paramount and training data is available, common in enterprise and academic contexts. In software engineering, models are often fine-tuned on code corpora or internal codebases to align with proprietary APIs and coding standards, yielding improvements in code completion and compile correctness. Modern practice increasingly blends paradigms: an instruction-tuned base model might be further augmented with retrieval during inference, exploiting the complementary strengths of parameter specialization and dynamic knowledge access. Fine-tuning is favoured when knowledge or skills cannot be easily injected via prompts or external tools, particularly when efficiency or latency constraints require capabilities to reside within model parameters (such as mobile deployments without retrieval infrastructure). While techniques like low-rank adaptation have reduced costs, fine-tuning remains resource-intensive compared to inference-time approaches, leading practitioners to typically attempt prompt engineering and RAG first, resorting to fine-tuning only when base model limitations cannot be overcome at inference-time [70].

3.3.4. Agentic Systems and Multi-Agent Frameworks

The paradigms discussed thus far primarily address single-model applications, but complex engineering tasks often benefit from decomposition into specialized subtasks and autonomous operation over extended workflows. This observation motivates the development of agentic and multi-agent frameworks.

The term *agent* is used inconsistently across industry and research, but the distinguishing characteristic is *autonomous, goal-directed iteration* rather than simple tool invocation. While tool-calling systems respond reactively to prompts and terminate after function execution, agents pursue objectives through recursive planning, action execution, outcome observation, and strategy adaptation [69]. This iterative, self-directed behaviour, where an LLM operates over multiple reasoning cycles to achieve high-level goals

[3] *Low-Rank Adaptation* freezes the original model weights and trains only small additional matrices, reducing trainable parameters by up to 10,000× while maintaining performance.

[4] *Machine Learning Operations* is the engineering discipline of deploying, versioning, and maintaining machine learning models in production, similar to DevOps for traditional software. From cloud.google.com/architecture/mlops [accessed 21-08-2025].

without explicit human instruction at each step, differentiates agentic systems from the inference-time augmentation techniques discussed previously.

Multi-agent systems extend this paradigm by coordinating multiple specialized LLM instances. Research demonstrates that LLM-based systems have "rapidly evolved from single-agent planning to operating as multi-agent systems" [69], reflecting a shift toward distributed problem-solving architectures where agents assume distinct roles – planner, executor, critic, domain specialist – and exchange information through structured interactions. Each agent can leverage core paradigms (RAG for information retrieval, fine-tuned models for specialized subtasks, orchestrated prompts for inter-agent communication) while contributing specialized capabilities to collective problem-solving. Frameworks like CAMEL (Communicative Agents for Mind Exploration of Large Language Models) demonstrate this through role-playing setups where agents collaborate without direct human intervention [68].

The value proposition of agentic systems is their capacity for long-horizon, open-ended tasks that resist decomposition into fixed workflows. By enabling recursive planning and self-coordination, these systems can autonomously manage complex software engineering tasks, workflow automation, and scenarios requiring parallel solution space exploration [69, 74]. However, increased autonomy introduces complexity: agents may produce inconsistent results, enter infinite loops, or fail to converge on valid solutions. Communication overhead scales with agent count, and architectural choices (hierarchical coordination versus peer-to-peer interaction) critically determine system effectiveness [74]. Evaluation and debugging present particular challenges when agents dynamically generate plans, as tracing errors through emergent multi-agent behaviour remains non-trivial. Despite these limitations, multi-agent architectures represent a significant evolution in LLM application design, particularly for domains requiring modular specialization and sophisticated orchestration [69].

3.3.5. Frameworks, Platforms, and Service Ecosystems

The rise of LLM applications has been paralleled by a rich ecosystem of development tools that fall into two broad categories: *application frameworks and libraries* that provide building blocks for LLM-powered systems, and *model hosting services* that provide access to pre-trained models, whether through cloud APIs or local deployment.

Application frameworks and libraries address the engineering challenges of building with LLMs by providing abstractions for common patterns and workflows. On the open-source side, Hugging Face's Transformers library has become foundational for accessing and fine-tuning models [75]. It provides a unified API to hundreds of pre-trained models (GPT-style, T5, etc.), abstracting architectural differences and enabling easy model switching [75]. The accompanying Model Hub encourages community-driven sharing of fine-tuned checkpoints and provides utilities for deployment from cloud to on-device scenarios. Specialized libraries target specific paradigms: LlamaIndex (GPT-Index) offers tools for building RAG pipelines (indexing documents, querying them, composing retrieved chunks with prompts), while LangChain provides components for prompt chaining, tool integration, and agent construction. These frameworks handle common patterns (conversing with vector databases, formatting outputs, managing conversation state) allowing developers to focus on application logic rather than infrastructure. Newer libraries like Pydantic-AI integrate LLM calls with type validation using Pydantic schemas to parse model outputs, enforcing reliable output formats. These frameworks emphasize developer experience by abstracting prompt templating, context management, and multi-step control flow into reusable components, enabling software engineers to integrate LLM capabilities like conventional software libraries.

Model hosting services provide access to pre-trained models through two deployment paradigms: cloud-based APIs and local serving infrastructure. Cloud services like OpenAI's ChatGPT/GPT-4, Anthropic's Claude, and Google's PaLM 2 and Gemini offer high-performance models accessible through remote endpoints. These services provide immediate access to frontier capabilities trained on massive proprietary datasets, adaptable to diverse tasks through prompt engineering or API-provided fine-tuning interfaces. The value proposition is operational simplicity: developers leverage cutting-edge models without managing training infrastructure, model serving, or hardware scaling. For instance, Claude Sonnet 4's strong performance on reasoning and coding tasks enabled rapid development of applications like Cursor (an AI-integrated IDE) through straightforward API integration. However, this convenience introduces trade-offs: vendor dependency with associated pricing structures and rate limits, data privacy considerations when transmitting prompts to external servers [70], and limited customization. For applications requiring data sovereignty or offline operation, local hosting solutions like Ollama provide streamlined deployment of open models (GPT, DeepSeek, LLaMA) on local hardware, particularly optimized for personal workstations and on-premises environments. While lacking the raw capability of frontier proprietary models, local hosting eliminates data transmission concerns and vendor dependency. This dichotomy characterizes the ecosystem: cloud API services prioritize raw capability and convenience at the cost of vendor dependency, while local hosting and open models emphasize privacy, control, and deployment flexibility, often paired with application frameworks to build custom solutions on self-hosted infrastructure.

3.4. Synthesis and Implications for KBE

The examination of design paradigms for LLM-based applications reveals that modern implementations demand careful consideration of multiple technical and operational factors. These insights directly inform the system design decisions discussed in Chapter 5 and Chapter 6, and guide the formulation of measurable success criteria in the next section. Practitioners today can choose from four primary design paradigms, each offering distinct advantages and trade-offs along a spectrum ranging from lightweight inference-time modifications to full-scale model retraining. These paradigms vary in their resource demands, flexibility, and performance characteristics, as summarized in Table 3.1.

Table 3.1: Comparison of LLM Application Design Paradigms.

Paradigm	Primary Benefits	Key Drawbacks	Resource Requirements	Optimal Use Cases	Key Sources
Prompt Engineering & Orchestration	Rapid prototyping; Model-agnostic; No training required; High interpretability	Context window limitations; Sensitivity to prompt wording; Limited specialization	Low (inference-time only)	Quick adaptation; Conversational interfaces; Bounded tasks	[66, 67, 68, 71, 70]
Retrieval-Augmented Generation & Tools	Reduces hallucinations; Up-to-date information; External capability integration; Scalable knowledge base	Added system complexity; Retrieval quality dependency; Latency overhead	Medium (infrastructure required)	Knowledge-intensive tasks; Dynamic information needs; Multi-modal applications	[72, 59, 71, 73, 69]
Fine-Tuning & Customization	Domain specialization; Consistent outputs; Format adherence; Parameter efficiency options	Resource intensive; Risk of overfitting; Reduced flexibility; Model maintenance	High (training required)	Domain-specific accuracy; Structured outputs; Private deployment	[70, 75]
Agentic & Multi-Agent Systems	Complex task decomposition; Autonomous operation; Role specialization; Collaborative problem-solving	Increased complexity; Communication overhead; Evaluation challenges; Potential failure cascades	Variable (depends on agents)	Long-horizon tasks; Workflow automation; Parallel processing needs	[69, 68, 76]

Crucially, these paradigms are not mutually exclusive. Modern LLM systems often integrate multiple approaches to capitalize on their respective strengths while mitigating individual limitations. For example, a system may employ a fine-tuned foundation model, enrich it with retrieval-augmented capabilities for dynamic knowledge access, and coordinate multiple specialized agents for complex task orchestration.

These findings have direct implications for Knowledge-Based Engineering applications. LLMs can enhance traditional KBE systems through natural language interfaces for knowledge capture and querying, automated rule implementation via code generation, and intelligent assistance in developing and maintaining knowledge models. However, effective integration requires a careful balance between the deterministic, transparent behavior of rule-based systems and the probabilistic, flexible nature of neural models.

3.5. Measurable Success Criteria

The presented analysis supports several key conclusions that inform the development of measurable success criteria (MSC), outlined below, and the subsequent industrial case study in Chapter 4. These criteria will be used to evaluate the developed framework in Chapter 7, enabling a quantifiable assessment of the research goals defined in Chapter 2.

First, the selection of an LLM application paradigm should be guided by specific requirements, such as accuracy, interpretability, resource constraints, and deployment context, rather than technological novelty. Second, hybrid approaches that combine multiple paradigms are likely to yield superior performance in complex, real-world applications. Third, the rapid evolution of the LLM ecosystem demands flexible system architectures that can accommodate emerging capabilities and evolving standards. Finally, effective integration of LLMs in engineering contexts requires rigorous attention to validation, verification, and quality assurance processes to ensure reliability and trustworthiness in critical applications.

REQ-1-1 (Framework Hosting)

The copilot tool shall support both local LLM deployment and remote LLM inference through privacy-compliant API services.

> **MSC-1-1:** The copilot tool shall support local LLM deployment via the Ollama service and remote LLM access through API services offered by approved providers (Anthropic & Groq).

REQ-1-2 (Code Completion Capability)

The developed copilot tool shall be capable of completing auto-generated skeleton code produced by the SysML-to-Python Translation Engine.

> **MSC-1-2:** The copilot tool shall achieve a minimum 85% successful completion rate for skeleton code structures, measured against a standardized test suite of ParaPy code skeletons with varying complexity levels.

REQ-1-3 (Code Generation Capability)

Building upon REQ-1-2, the copilot tool shall be capable of generating class definitions even in the absence of skeleton code.

> **MSC-1-3** The copilot tool shall achieve a minimum 75% successful completion rate for generating class definitions from natural language specifications, measured against a standardized test suite of ParaPy development cases with varying complexity levels.

REQ-2-1 (Syntactic Correctness)

AI-generated ParaPy code shall exhibit syntactic correctness.

> **MSC-2-1** Generated code shall achieve 95% syntactic correctness as measured by automated parsing and compilation checks.

REQ-2-2 (Runtime Correctness)

AI-generated ParaPy code shall exhibit runtime correctness.

> **MSC-2-2** Generated code shall execute without runtime errors in 80% of test cases, measured through automated execution testing within controlled ParaPy environments using standardized input parameters.

REQ-2-3 (Functional Correctness)

AI-generated ParaPy code shall exhibit functional correctness. The paradigm analysis indicates that functional correctness represents the most challenging aspect of code generation, particularly for domain-specific frameworks where training data may be limited.

> **MSC-2-3** Generated code shall achieve intended functionality in 70% of test scenarios, validated through automated testing suites that compare output behaviour against reference implementations and expert-defined acceptance criteria.

REQ-3-1 (Development Time Metric)

Development time shall be the primary metric used to assess performance improvements introduced by the copilot tool in KBE application development.

> **MSC-3-1** The primary metric to assess performance improvements introduced by the copilot tool shall be development time reduction through controlled user studies using standardized KBE development scenarios, with measurements captured for task completion time and development velocity metrics (e.g., lines per minute, features per minute).

REQ-3-2 (Functional Equivalence)

AI-generated code shall be 100% functionally identical to manually completed, correct code. The paradigm analysis reveals that achieving perfect functional equivalence may be unrealistic given the probabilistic nature of LLM outputs, suggesting a need for refined expectations.

> **MSC-3-2** AI-generated code shall achieve functional equivalence to manually completed reference solutions in 85% of test cases, measured through comprehensive output comparison and behaviour verification testing protocols.

REQ-3-3 (Quality Assessment)

AI-generated code shall receive an expert-assigned grade reflecting its semantic correctness (defined in section 2.6), maintainability (using the MI and CC as defined in section 2.6), and adherence to PEP-8 standards.

> **MSC-3-3** Generated code shall receive an average expert quality score of 7.5 out of 10, using a standardized rubric covering semantic correctness (40%), maintainability (35%), and PEP-8 compliance (25%), evaluated by qualified ParaPy domain experts.

REQ-3-4 (Composite Performance Score)

A composite score, incorporating development time, functional correctness, and expert-assigned quality grade, shall serve as a secondary metric to assess overall performance improvement.

> **MSC-3-4** The composite performance score shall be calculated as a weighted average of normalized development time improvement (40%), functional correctness rate (35%), and expert quality score (25%), with the copilot tool achieving a minimum composite score of 75%.

REQ-4-1 (Development Time Improvement)

The use of the copilot tool shall reduce the total development time of KBE applications for both novice and expert users. Time reduction benefits may vary significantly between user groups due to different baseline capabilities and adaptation patterns.

> **MSC-4-1** The copilot tool shall achieve minimum development time reductions of 40% for

novice users and 25% for expert users, measured through controlled comparative studies using similar -but mutually exclusive- KBE development tasks and the primary metrics introduced in [MSC-3-1].

REQ-4-2 (Composite Score Improvement)

The use of the copilot tool shall increase the composite score outlined in REQ-3-4 with respect to manually completed code, for both novice and expert users. This requirement acknowledges that overall performance improvement encompasses multiple quality dimensions beyond time reduction alone.

> **MSC-4-2** The copilot tool shall achieve composite score improvements of minimum 30% for novice users and 15% for expert users compared to their baseline manual development performance.

REQ-4-3 (Knowledge Barrier Reduction)

The use of the copilot tool by novice users shall reduce the knowledge required for KBE application development. The paradigm analysis suggests that knowledge barrier reduction can be measured through task completion rates and error reduction patterns.

> **MSC-4-3** Novice users utilizing the copilot tool shall achieve task completion rates equivalent to 80% of expert baseline performance and demonstrate 50% reduction in domain-specific knowledge errors compared to unassisted novice performance.

4

Industrial Study: GKN Aerospace

This chapter presents the set of interviews conducted at GKN Aerospace Sweden, located in Trollhättan, between May 4th and May 16th, 2025. It serves as an extension of the theoretical foundations laid out in Chapter 3, providing empirical substantiation of the research objectives and identified knowledge gap from both academic and industrial perspectives. The primary objective of these interviews was to uncover current challenges and requirements surrounding the further implementation of Knowledge-Based Engineering (KBE) within GKN's industrial context and to explore how these findings inform the design of the AI-supported copilot described in Chapter 6.

Following a brief introduction to GKN Aerospace, the rationale for conducting the interviews is outlined in section 4.1. Thematic analysis [77] is used to extract recurring themes, which are presented in section 4.2 along with relevant cross-thematic relationships. This is followed by a technical assessment of existing AI tools in section 4.3, and a summary of key findings in section 4.4.

For brevity, only findings directly relevant to the current research are presented in this chapter. A complete analysis, including all identified themes, relationships, and the methodology used for interview processing, is provided in Appendix A.

Company Introduction

GKN Aerospace Sweden AB, established in October 2012 following the incorporation of Volvo Aero into the GKN group and owned by Melrose Industries since 2018, is the parent company of GKN Aerospace Engine Systems. Operating from Trollhättan, Sweden with approximately 1,700 employees,[1] the company collaborates with major aircraft engine manufacturers including Rolls-Royce, Pratt & Whitney, Safran, and General Electric, with components currently integrated into over 90 percent of all new large civil aircraft worldwide.[2] [3] In addition to its prominent role in commercial aviation engine programs, the company manufactures components for rocket engines used in ESA's Ariane 5 launch vehicle and is recognized by the European Space Agency as a Center of Excellence for rocket engine nozzles and turbines, with particular expertise in subsystems for the Vulcain main rocket engine.[4]

4.1. Study Purpose

This industrial study, conducted at GKN Aerospace Sweden, substantiates the research gaps and claims established in Chapter 1 and Chapter 3. Specifically, the study aims to verify from an industrial perspective that the adoption and application of KBE is indeed time-intensive and knowledge-intensive, and to identify the specific challenges users encounter during this process.

The primary objective of the interviews was to uncover the current barriers and requirements associated with the implementation of KBE within GKN's industrial context, and subsequently to explore how AI can help overcome these barriers. Understanding these challenges is essential for developing effective

[1] From https://www.allabolag.se/bokslut/gkn-aerospace-sweden-ab/trollh%C3%A4ttan/flygplan-utrustningar-tillverkning/2JYGBHNI63IG4 [accessed 25-09-2025]
[2] From https://soff.se/en/medlem/gkn-aerospace-ab/ [accessed 25-09-2025]
[3] From https://www.gknaerospace.com/ [accessed 25-09-2025]
[4] From https://www.gknaerospace.com/markets-solutions/engines/space-launcher-engines/ [accessed 25-09-2025]

support systems that address real-world adoption obstacles. As outlined in requirements REQ-4-1, REQ-4-2, and REQ-4-3 in Chapter 2, the proposed copilot must improve KBE application development performance for both novice and expert users. To achieve this, it is critical to understand what barriers each user group faces when engaging with KBE technology. This motivated the deliberate inclusion of participants with varying levels of KBE experience, ranging from complete novices to seasoned practitioners.

Participants were first introduced to the concepts of KBE and a demonstrator application (a Turbine Rear Structure model, developed using the ParaPy SDK), followed by a guided live coding session. This hands-on approach enabled participants to gain practical exposure to the ParaPy SDK and its application development workflow. Subsequently, they were asked to reflect on their experience with the demonstration, their existing knowledge of KBE principles and automation practices, and their perceptions of how Generative AI or Large Language Model (LLM) based tools might support similar development tasks in the future. No AI-based development tools were demonstrated or used; participants reflected solely on their potential applicability based on their own experiences with LLMs in other contexts.

The inclusion of novice users (individuals with limited or no prior exposure to KBE or ParaPy) was a strategic methodological choice. These participants provide critical insight into the initial barriers to adoption: the steepness of the learning curve, the clarity of documentation, the intuitiveness of the framework, and the cognitive load associated with translating domain knowledge into formal KBE rules. For instance, when a novice user reports difficulty navigating a codebase or understanding framework conventions, this reflects not solely a shortcoming of the participant but rather provides valuable diagnostic information about the entry barriers that must be addressed if KBE adoption is to scale within the organization. By contrast, more experienced participants offered complementary insights into integration challenges, toolchain heterogeneity, and organizational factors that affect sustained KBE use. Together, these perspectives enable a holistic understanding of the factors influencing KBE adoption across different user profiles.

4.2. Findings

The reflexive thematic analysis of interview transcripts identified eight common themes related to KBE adoption challenges at GKN Aerospace. Table A.2 presents an overview of these themes and the number of participants who raised each concern. While all eight themes provide valuable insights into the organizational and technical landscape surrounding KBE implementation, the present discussion focuses on the two themes most directly relevant to the proposed copilot: *Skill Gap & Learning Curve* (Theme 3) and *AI/LLM Assistance – Potential & Trust Concerns* (Theme 4). These themes were universally mentioned by all five participants and exhibit strong interdependencies that inform the design requirements for AI-assisted KBE development tools. An overview of all themes, and their discussion, is available for inspection in section A.2.

Table 4.1: Overview of common themes identified in the interviews and their relative urgency, based on the number of Domain Experts (DEs) referencing each theme.

Nr	**Theme**	**DE responses** *(n = 5)*
1	Inadequate knowledge capture & formalisation	5
2	Integration & tool-chain heterogeneity	5
3	Skill gap & learning curve	5
4	AI/LLM assistance – potential & trust concerns	5
5	Perceived value & benefit-effort trade-off of KBE	5
6	Usability & documentation shortcomings	4
7	Organizational & cultural barriers	5
8	Trust, reliability & validation of automated/AI solutions	3

Successful KBE application development requires dual competencies: deep domain knowledge in aerospace engineering and proficiency in software development, particularly in Python, ParaPy, and CAD APIs. This dual requirement creates a steep learning curve that emerged as a central barrier to adoption across all five interviews. The nature and perceived severity of the skill gap varied significantly across participants. Interviewee 1, who possessed strong programming skills, expressed concern about colleagues without coding backgrounds. Interviewees 2 and 3 emphasized the need for broader organizational training programs. Interviewee 4 admitted to a "lack of deep understanding" of ParaPy's architectural patterns yet demonstrated the ability to quickly adapt and modify existing code during the live coding session, suggesting that with appropriate guidance, novices can become productive relatively quickly. Poor documentation and unintuitive user interfaces compound this learning curve. The difficulty in navigating codebases and understanding framework conventions represents a systemic entry barrier that must be addressed for KBE adoption to scale.

Given the pervasiveness of the skill gap, participants naturally discussed how AI and LLMs might help bridge this divide. All five interviewees reported active use of LLMs for code-related tasks, with expectations ranging from generic code completion to domain-specific chatbots capable of explaining ParaPy APIs. Interviewee 3 envisioned broad applicability: "A chatbot could help everywhere... explain what is happening... mostly from a ParaPy documentation perspective." Interviewee 4 offered a particularly clear statement of need: "I would like a chatbot with a deep knowledge of both the API and a wealth of examples."

However, enthusiasm for AI-driven support was consistently tempered by concerns about reliability and transparency. This reveals a fundamental paradox: AI chatbots are envisioned as learning aids to bridge the skill gap, yet relying on AI assistance may prevent engineers from developing the deep understanding necessary to validate and maintain KBE systems. Interviewee 2 captured this tension: "If you are a non-programmer... asking an AI... would have been probably faster, *if* it would have been a reliable answer." The interviews revealed striking contradictions in how participants conceptualize AI's role. Interviewee 1, despite "exclusive" daily ChatGPT use, still expressed a need for a dedicated ParaPy-specific chatbot, while Interviewee 4, despite extensive Copilot usage, downplayed generic code-completion features in favour of a knowledge-rich conversational agent. These positions suggest that current AI tools, while helpful, do not fully meet the needs of KBE developers working within specialized frameworks.

Trust concerns manifested differently across participants. Interviewee 2 warned against "blind reliance on AI without domain-specific training," emphasizing that without explicit model fine-tuning on ParaPy-specific knowledge, such tools would remain unreliable. Interviewee 3 took a more optimistic view that LLMs could help explain code behaviour, acknowledging that "the technology is not yet sufficiently mature" but seeing clear potential. Interviewee 5 found AI helpful only for proofreading tasks, dismissing its utility for deep CAD scripting.

Two overarching insights emerged from this synthesis. First, a persistent skill gap in Python and ParaPy programming drives both the learning curve barrier and the demand for AI-driven learning support, yet trust in AI outputs remains low without demonstrated reliability and domain specificity. This creates a tension: AI tools are needed to accelerate learning, but engineers are reluctant to adopt them without confidence in their capabilities. Second, documentation quality and usability are decisive factors for adoption. The difficulty of navigating documentation and understanding framework conventions compounds the technical challenges of KBE implementation. Addressing these interconnected challenges requires a support system that not only generates code but also educates users, explains design decisions, and operates transparently.

The technical assessment that follows evaluates the current capabilities of commercial LLMs for ParaPy code generation, providing empirical evidence of the limitations that underpin the trust concerns raised during interviews.

4.3. Technical Assessment of Current AI Tools

To empirically validate the trust concerns and reliability questions raised during the interviews, a technical assessment of current state-of-the-art LLMs was conducted. Five major AI platforms were tested: ChatGPT (GPT-4), Claude (Anthropic), DeepSeek, Mistral, and GitHub Copilot. This pragmatic sampling of readily accessible platforms represents the tools participants reported using in practice. Each platform was presented with identical task descriptions requesting ParaPy code generation for typical aerospace engineering components, representative of interactions encountered in production [78], based on the following prompt:

```
>>> "In Python, I want to generate a parametric wing model representing the wing outer body
    and inner structure. Can you write a Wing class for me in Python using the ParaPy SDK?"
```

The generated code was subsequently analysed for syntactic correctness, semantic accuracy, and adherence to ParaPy SDK conventions. While the same prompt was submitted multiple times to each model, resulting variations were minor and primarily lexical, with no meaningful impact on the syntax or semantics of the generated code. Due to the extensive nature of the model outputs, full responses from all platforms are available in section A.6 for reference. The technical assessment revealed six recurring error categories, highlighting systematic gaps in current LLM capabilities when applied to domain-specific frameworks. These patterns validate practitioner concerns initially raised in Chapter 1, reinforced through the theoretical context in Chapter 3, and now further substantiated by the interview findings regarding the complexity of ParaPy-based development. The most prevalent issues included:

1. **Incorrect import practices**

```
from parapy.core import *
#   OR
from airfoils import AirfoilCurve  # Non existing (ParaPy) library
...
...
ModuleNotFoundError: No module named 'airfoils'
```

```
import parapy.geom as ppg


@Input
def airfoil_root(self):
    return ppg.Airfoil(...)
...
...
AttributeError: module 'parapy.geom' has no attribute 'Airfoil'
```

2. **Improper usage of ParaPy's input declaration syntax**

```
span = Input(15.0, validate = positive)
...
...
NameError: name 'positive' is not defined
```

3. **Fundamental misunderstanding of the framework's type system**

```
@Part
def ribs(self):
    # Define ribs along the span of the wing
    ribs = []  # <--violation of Part grammar in ParaPy < 2.0.0
    num_ribs = []  # <--violation of Part grammar in ParaPy < 2.0.0
    return ...
```

```
RuntimeError: First statement in body should either be a docstring or ‘‘return‘‘, found:
     ...
```

4. **Inappropriate argument passing to constructors and methods**

```
@Part
def wing_surface(self):
    return LoftedSolid(
        profiles=[...],
        rulings=[...],  # <-- argument does not exist
    )
```

```
parapy.core.exceptions.InvalidArgs: Invalid inputs found for LoftedSolid: ['rulings'].
```

5. **Violations of framework organizational principles**

   This includes -but is not limited to- the violations of e.g. `Part` grammar but also the highly nested nature of generated primitives and usage of non-existing variables.

6. **Omission of essential import dependencies**

   This entails using hallucinated primitives that are not imported, causing name errors:

```
NameError: name 'Sweep' is not defined
```

Furthermore, models consistently attempted to apply general Python programming patterns or interpolate from other geometric libraries rather than following ParaPy-specific conventions, hallucinating primitives such as `Translated`, `Scaled` or `Rotated` for geometrical manipulations. These error patterns indicate that current language models rely on statistical interpolation from general programming knowledge rather than accurate understanding of domain-specific API requirements (see Chapter 10). The consistency of these failures across different model architectures suggests fundamental limitations in how current AI systems handle specialized frameworks with limited public training data.

4.3.1. Implications for KBE Development

This gap between general programming capabilities and specialized framework expertise partially explains why organizations continue to face significant barriers in KBE adoption, even as AI-assisted coding tools become ubiquitous in software development more broadly.

The poor performance of LLMs on ParaPy code generation can be attributed to the closed-source nature of the framework. This fundamental data scarcity cannot be overcome by simply using larger or more advanced general-purpose models. As emphasized during the interviews, this problem is compounded in industrial contexts where cloud-based AI services are often prohibited due to data security and confidentiality requirements. Employees in such environments are typically restricted to air-gapped or local infrastructures that rely on open-source or smaller models with significantly fewer parameters. These restrictions not only reduce access to cutting-edge capabilities but also exacerbate the performance limitations observed in this assessment when applying AI to domain-specific tasks requiring deep contextual understanding. Interviewee 1, despite being a frequent ChatGPT user, specifically expressed a desire for a tailored, ParaPy-specific tool that could operate within local constraints: a requirement that current general-purpose models cannot satisfy.

The identified error patterns translate into concrete design requirements for the proposed support system. While it is reasonable to assume that these deficiencies would be less pronounced if public language models had access to substantial ParaPy training data—or if fine-tuned specifically on ParaPy code and documentation—such approaches fall outside the scope of the current research. Instead, the developed system must compensate for these systematic gaps through retrieval-augmented generation, explicit knowledge injection, and structured validation workflows. The technical assessment thus reinforces the conclusion that effective AI assistance for KBE development necessitates purpose-built solutions that bridge the gap between general programming capabilities and domain-specific framework expertise.

4.4. Summary and Implications

The thematic analysis of interviews at GKN Aerospace, combined with the technical assessment of current AI tools, reveals two critical barriers to KBE adoption that directly inform the design of an AI-assisted support system. First, the steep learning curve associated with ParaPy requires simultaneous mastery of domain knowledge, Python programming, and framework-specific conventions, deterring adoption across all experience levels. Second, while participants recognize the potential of AI assistance to flatten this learning curve, they express significant concerns about reliability, transparency, and the risk of dependency on tools they do not fully understand. The technical assessment validates these concerns by demonstrating that even state-of-the-art language models systematically fail to generate functional ParaPy code due to the framework's closed-source nature and resulting absence from training corpora.

These findings motivate a refinement to the support framework outlined in Chapter 2. Requirements REQ-4-1, REQ-4-2, and REQ-4-3 mandate improvements in KBE application development performance for both novice and expert users. To address the skill gap while simultaneously building trust through transparency, the proposed system introduces a dual-agent architecture comprising an *Educational Agent* and a *Developer Agent*. This separation addresses a key tension identified in the interviews: the need for AI assistance must be balanced against the imperative that engineers develop genuine understanding rather than blind reliance on generated code.

An Educational Agent supports users by clarifying ParaPy's architectural patterns, explaining framework conventions, and assisting in navigating documentation to identify appropriate primitives. This pedagogical role directly addresses widespread concerns that insufficient documentation increases cognitive load, while also responding to user demands for tools with deep API knowledge and curated examples. By emphasizing explanation over pure code generation, the Educational Agent fosters the foundational understanding engineers need to critically evaluate outputs, thereby mitigating trust concerns. In parallel, the Developer Agent offers programmatic assistance—functionally similar to tools like GitHub Copilot but tailored to ParaPy development—serving users who already possess a basic understanding of the framework.

Together, these agents address the two central themes from the interview analysis. The skill gap can be significantly flattened through the Educational Agent's guidance combined with the Coding Agent's productivity support, while concerns about AI reliability and trust are mitigated by prioritizing transparency, explanation, and education over opaque code generation. The interview findings, supported by the technical assessment, provide a grounded understanding of the conditions influencing AI-driven KBE adoption at GKN Aerospace. Building on these foundations, the following chapters detail the implementation and evaluation of this dual-agent approach, collectively called the *copilot*.

5
Methodology

This chapter presents the methodological approach for developing and evaluating a generative AI-powered coding assistant tailored for Knowledge-Based Engineering application development using the ParaPy SDK. The framework development operates under two core constraints identified in Chapter 2 and verified through the literature review (Chapter 3): (1) the scarcity of ParaPy-specific training data in publicly available language models, leading to extensive hallucinations in code generation; and (2) the necessity for local execution and privacy due to use of closed-source software and proprietary data.

The research design follows a three-pronged strategy rooted in the industrial case study findings. Interview analysis revealed two distinct user needs: rapid code generation and educational support for understanding ParaPy patterns. The methodology addresses these through a dual-agent architecture comprising a Developer Agent optimized for code generation and an Educational Agent focused on pedagogical support.

First, the methodology establishes a systematic pipeline for AI agent creation and deployment that leverages retrieval-augmented generation and prompt engineering to compensate for limited ParaPy-specific training data, maintaining compatibility with both local execution via Ollama and privacy-compliant API services. Second, this pipeline is applied to develop the dual-agent architecture, with each agent specialized to address one of the two distinct user needs identified through practitioner interviews. Third, an evaluation framework is designed to measure the developed system's performance through automated assessment of syntactic and runtime correctness, manual case studies examining functional accuracy and code quality, and user testing protocols that evaluate development time impact and effectiveness across expertise levels.

5.1. Legal & Ethical Considerations

The development and evaluation of this framework involve handling proprietary ParaPy SDK materials and potentially sensitive industrial application code, necessitating careful consideration of confidentiality and data protection requirements. The knowledge base indexed for semantic search comprises exclusively public-facing ParaPy documentation, API references, and approved usage examples that would be accessible to licensed ParaPy developers through official channels. When utilizing remote API services, only public ParaPy API documentation and syntactic patterns are transmitted within prompts. Exact implementation details of ParaPy's internal mechanisms remain strictly excluded from all model interactions, preserving the distinction between understanding how to use the ParaPy API versus how ParaPy implements its functionality. These privacy constraints limit model selection to either locally deployed open-source models via Ollama for complete air-gapped operation, or external APIs from Anthropic[1] and Groq[2] that ensure temporary or zero data retention and compliance with security

[1]Anthropic is an AI safety-focused company founded in 2021 by former members of OpenAI. It is the creator of Claude, a family of large language models designed with an emphasis on safety, helpfulness, and honesty. Claude is accessible via a web interface, API, and a suite of developer tools. From `https://www.anthropic.com/` [accessed 02-10-2025].

[2]Groq is a semiconductor company that develops specialized AI inference processors known as Language Processing Units (LPUs). The company offers both hardware solutions and cloud-based inference services, primarily focused on hosting open-source models on industry-grade infrastructure to enable high-speed, low-latency inference. From `https://groq.com/` [accessed 02-10-2025].

requirements.

During user testing and case study evaluation, participants' build directories and generated code may be collected for analysis under explicit informed consent, with participants retaining full ownership of their intellectual contributions. Collected artifacts are anonymized prior to analysis, with any organization-specific or proprietary design elements removed or generalized. Aggregated results and illustrative code examples used in thesis documentation undergo review to ensure no competitive or confidential information is disclosed, balancing the research need for authentic industrial examples with the legitimate confidentiality concerns of participating organizations.

5.2. Agent Creation Pipeline

The successful deployment of an AI agent, whether as a single agent or within a multi-agent framework, involves several essential steps. Whilst these steps may vary depending on the specific programming framework and level of abstraction employed, the main steps identified in this research are as follows:

1. LLM deployment, selection and customisation;
2. System prompt development;
3. Runtime dependency configuration;
4. Tool development and deployment.

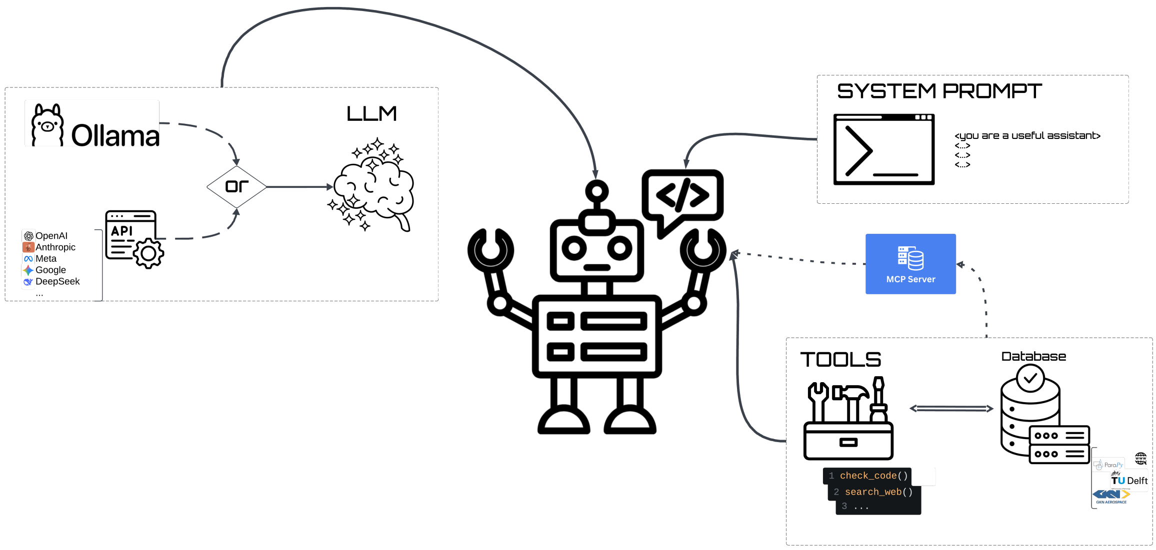


Figure 5.1: Top-level overview of an AI agent architecture, illustrating (1) the role of the large language model (LLM) as the agent's cognitive core, (2) the incorporation of tailored system instructions to guide behaviour, and (3) the integration of external tools and databases, possibly through MCP services.

5.2.1. LLM Deployment, Selection & Customisation

Model deployment is the first consideration in the pipeline. The framework supports both remote API-based deployment and local execution to balance accessibility with privacy requirements. Remote deployment leverages commercial provider infrastructure (like Claude and Groq), while local deployment enables self-hosted solutions through platforms like Ollama. This approach provides deployment flexibility, accommodating cutting-edge commercial models, smaller open-source alternatives, or proprietary in-house models.

Model selection follows deployment, driven by the fundamental trade-off between computational requirements and output quality. Model size directly impacts memory consumption and inference speed, with local deployment environments typically imposing stricter resource constraints. Selection balances parameter count, required capabilities, and task complexity. Well-defined tasks benefit from smaller, specialized models offering faster inference, while complex reasoning tasks require larger models despite higher computational demands. Within multi-agent architectures, this enables task-specific model assignment, where each agent operates with a model appropriate to its functional requirements.

Model configuration further optimizes performance for specific use cases. The current research standardizes temperature settings to zero across all agents, minimizing stochastic sampling to maximize reproducibility in evaluation protocols. A more elaborate list on LLM settings and their effects is presented in section 11.1 but is not relevant for a further understanding of the framework.

5.2.2. Prompt Engineering

System prompt development constitutes the second critical step in agent deployment. As newer models such as Claude 4 and GPT-5 follow instructions more precisely, prompt engineering has become increasingly important. These models often require more elaborate and explanatory prompts but offer users greater control over model behaviour in return. This research adopts Anthropic's systematic approach to prompt engineering, as outlined in [79] and more extensively reported in section 11.2.

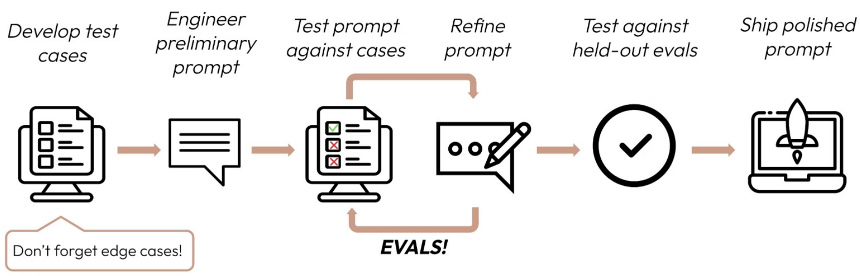


Figure 5.2: Schematic overview of prompt engineering design process, taken from prompt engineering course by Anthropic. From docs.anthropic.com [accessed 03-09-2025].

As shown in Figure 5.2, prompt engineering follows a structured design process that includes establishing success criteria, developing test cases, and conducting iterative refinement prior to deployment. An initial prompt is created, evaluated against test cases, and iteratively improved to arrive at a system prompt that delivers satisfactory performance.

For the two agents in the framework, the strategies from [79] are employed to generate draft prompts that include ParaPy SDK principles such as grammatical rules, best practices, small examples enabling pattern recognition for difficult tasks, response guidelines, and guardrails for queries falling outside the agent's scope. While ParaPy information remains consistent across both agents from a broad perspective, specific best practices, guidelines, and guardrails are tailored to each agent's role. The draft prompts undergo evaluation and iteration until deemed sufficient for testing in Chapter 7. Major model providers such as Anthropic offer prompt generator and improver tools to facilitate this process, including resources for defining success criteria, generating test cases, and designing evaluation procedures. However, caution is advised when reusing auto-generated prompts directly, as they are often optimized for the provider's own models. Nevertheless, such templates serve as valuable starting points in the prompt engineering process. For full documentation, the reader is referred to Anthropic's official documentation.

5.2.3. Runtime Dependencies

With the deployment infrastructure and system prompts established, the next step involves configuring the runtime dependencies that support agent operations. Runtime dependencies are framework components required by agent tools during execution, distinct from the deployment and configuration steps described previously. In this framework, the primary runtime dependency is a semantic search engine that provides dynamic access to ParaPy-specific knowledge.

The framework addresses sparse ParaPy-specific training data in commercial language models through a local knowledge base containing ParaPy SDK documentation, usage examples, and API references. Rather than relying solely on pre-trained knowledge, agents query this curated knowledge base through a custom-built semantic search engine based on cosine similarity, as detailed in section 6.4. The search mechanism transforms both indexed documentation and user queries into high-dimensional vector representations, enabling similarity-based retrieval of relevant information.

The knowledge infrastructure maintains three distinct searchable collections, each optimized for different information needs. The *documentation index* contains the complete ParaPy SDK documentation as published on the official website (version `1.14.1`), including explanations, tutorials, guides, and conceptual descriptions.The *example index* contains isolated code snippets extracted from this same documentation. Rather than searching through full documentation pages with surrounding text, agents can directly search for relevant code patterns and implementations to adapt for their specific tasks. The *API reference* index provides technical specifications for ParaPy functions and primitives, including what inputs they accept, what outputs they produce, and how to use them correctly. By separating these three types of information, agents can search more efficiently and retrieve only the most relevant content for their specific needs.

The search engines are initialized when the framework starts and remain available throughout the agent's operation. Importantly, the system does not load all documentation into the agent's working memory at startup. Instead, agents actively request specific information only when needed during their reasoning process. This on-demand approach reduces computational overhead (agents only process the documentation they actually need) and enables agents to perform multiple searches, refining their queries based on what they learn from previous search results. For example, an agent might first search broadly for positioning concepts, then perform a more targeted search for specific positioning functions after understanding the general approach.

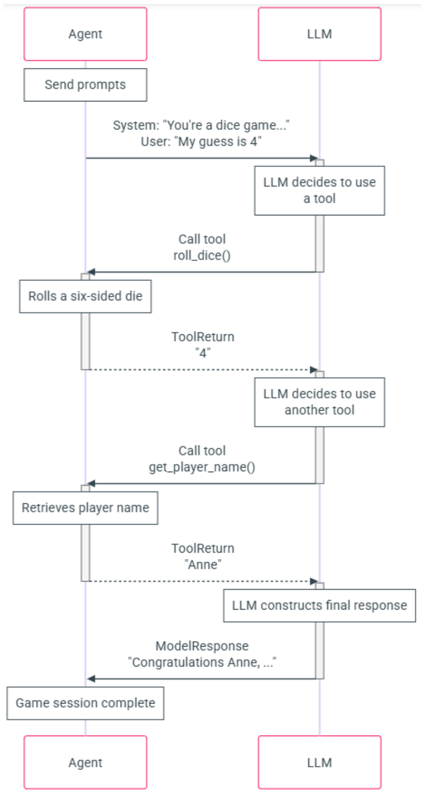


Figure 5.3: Illustration of tool calling within AI agentic frameworks. Shows the clear roles of the LLM and the agent framework in action. The interaction of agents with tools follows the through-action-observation cycle. From Pydantic AI documentation [accessed 15-10-2025].

5.2.4. Tool Development, Verification & Validation

Large language models, by default, are limited to generating text-based responses. Tools extend agent capabilities by enabling them to perform actions beyond text generation, such as searching databases, executing code, or accessing external APIs. In the context of this framework, tools serve as the bridge between the agent's reasoning capabilities and the external resources it needs to generate correct ParaPy code. For instance, a tool might retrieve relevant documentation from the semantic search engine, verify the syntax of generated code, or check whether code executes without errors.

Tool calling is a relatively recent advancement in LLM interaction design (see subsection 3.3.2) that enables models to produce structured outputs specifying which tool to call and with what parameters, typically in JSON format. Most large commercial models support this functionality by default, while smaller open-source models may require specific training, often indicated by suffixes such as `*instruct`, `*chat`, or `*versatile`. The interaction follows a thought-action-observation cycle: the model reasons about the task, selects an appropriate tool, executes it, and integrates the tool's output into its ongoing response, as demonstrated in Figure 5.3. A comprehensive introduction to agents and tool calling is available through HuggingFace's Agents Course (last accessed 02-10-2025).

Tool development in the framework follows three guiding principles that ensure modularity and reliability. First, each tool encapsulates a narrowly scoped capability aimed at addressing a specific model limitation, supporting independent testing and flexible composition. Second, tools return output as structured text using consistent delimiters and labeled sections, reducing the reasoning burden on the language model by providing clearly formatted information. Third, robust error handling distinguishes between recoverable failures (triggering retry logic that allows the agent to adapt its approach) and unrecoverable failures (surfaced with clear messages that halt execution as needed).

The framework implements tools across three functional categories, each targeting distinct challenges identified during baseline evaluations. *Data retrieval tools* provide agents access to the semantic search engines configured in the runtime dependencies (see previous section), enabling dynamic queries against ParaPy documentation, code examples, and API references to address sparse training data. *Code improvement tools* assist in producing framework-appropriate design patterns and idiomatic ParaPy implementations, such as providing correct import suggestions and ParaPy primitive signatures. *Code verification tools* ensure output quality through progressive validation mechanisms, enabling the agent to check the syntax and runtime correctness of its generated code. The detailed tool development process and complete tool inventory are elaborated in section 6.6.

5.3. Framework Architecture and Design Approach

The framework adopts a dual-agent architecture comprising a *Developer Agent* and an *Educational Agent*, each tailored to support different aspects of KBE application development. The Developer Agent constitutes the primary mechanism through which the framework satisfies the core requirements, automating code generation tasks (either from scratch or from skeleton code) to address the time-intensive nature of KBE development, as outlined in the research clarification (Chapter 2). The Educational Agent was subsequently introduced based on insights from the industrial analysis presented in Chapter 4, specifically addressing REQ-4-3 by flattening the steep learning curve associated with ParaPy development and improving usability for novice users through on-demand pedagogical support.

Both agents share a common design philosophy: each is powered by an LLM (either through API services or locally deployed), has access to semantic search functionalities (subsection 5.2.3), employs a tailored system prompt, and utilizes specially designed tools. However, their specific implementations differ to optimize for their distinct objectives: (1) the Developer Agent produces syntactically correct, runtime-correct, and functional Python code using the ParaPy SDK, while (2) the Educational Agent provides pedagogical support tailored to user expertise levels.

The *Developer Agent* focuses on code generation, completion, and debugging within the ParaPy SDK environment, addressing REQ-1-2 and REQ-1-3. To maximize code generation performance, the agent accesses only the *example index* and *API index*, minimizing noise from explanatory text present in full documentation. The example index provides reference implementations, while the API index supplies specifications for ParaPy primitives. The agent's system prompt contains ParaPy principles, curated examples reinforcing these principles, best practices for KBE development, and general reasoning instructions, detailed in section 6.5. Tools enable semantic search access and implement verification steps checking syntactic accuracy (REQ-2-1) and runtime correctness (REQ-2-2). Additional tools provide import suggestions and function signatures, addressing weaknesses observed in section 4.3. All outputs consist of executable code with minimal explanatory context.

The *Educational Agent* provides conceptual guidance on KBE principles and ParaPy development patterns, specifically targeting REQ-4-3 by reducing domain knowledge requirements through on-demand learning support. To maximize pedagogical effectiveness, this agent accesses the complete *documentation index* and *API index*, enabling comprehensive responses with appropriate documentation references. The system prompt is simpler than the Developer Agent's, containing only reasoning instructions and task descriptions without embedded ParaPy details, as this information is retrieved on demand from the documentation index (see section 6.5). Tools focus on connecting the LLM to the semantic search engine and selectively filtering documentation by type (quickstart content, tutorials, API references). Rather than generating production code, this agent emphasizes structured explanations, documentation references, and illustrative examples tailored to practitioner concerns identified in the industrial study.

The complete tool suite for both agents is discussed in section 6.6, while exact agent implementations are detailed in section 6.7.

5.4. Verification and Validation Methodology

The verification and validation strategy employs four complementary approaches, progressing from component-level verification to system-level validation: unit testing validates core framework components, automated evaluation assesses agent performance across comprehensive test scenarios, manual case studies provide expert verification of complex implementations, and user testing validates practical effectiveness with developers of varying expertise levels. This multi-tiered approach balances automated scalability with human expert judgment, ensuring both technical correctness and practical usability. Chapter 7 operationalizes this framework and succinctly presents its design, while the complete technical implementation details are available in Chapter 13.

5.4.1. Unit Testing

Unit testing constitutes standard software engineering practice and serves as a prerequisite for further verification and validation. Testing ensures core components behave as intended and verifies edge case handling to enhance framework robustness. Critical components under test include knowledge infrastructure operations (semantic search accuracy and retrieval relevance, as implemented in section 6.4), tool execution and output formatting (correct function calls and structured responses, as detailed in section 6.6), agent configuration and initialization (proper model loading and prompt injection, described in section 6.7), and output validation mechanisms (syntax checking and runtime verification). Successful unit testing confirms that individual framework components operate correctly in isolation before proceeding to integrated system evaluation. Results are presented in section 7.3.

5.4.2. Automated Evaluation Framework

The automated evaluation framework assesses both agents using test scenarios tailored to their distinct roles. The Developer Agent is evaluated on code generation tasks spanning complexity levels from simple code completion to complete class generation, stratified by task type (skeleton completion versus generation from scratch) and required ParaPy features, undergoing evaluation across four dimensions: syntactic correctness (REQ-2-1), runtime correctness (REQ-2-2), functional correctness (REQ-2-3), and code quality (REQ-3-3). The Educational Agent is evaluated on pedagogical tasks where it responds to developer questions, explains ParaPy concepts, and provides illustrative examples, assessed only on functional correctness (whether explanations are accurate and helpful) and code quality for example snippets, as syntactic and runtime validation are less relevant for pedagogical examples that may be intentionally incomplete fragments designed to illustrate specific concepts.

The framework employs independent large language models as assessors to evaluate agent performance, a methodology termed "LLM Judge." This approach simulates traditional academic grading, where judges assess agent outputs against defined rubrics and their own domain knowledge, analogous to how an instructor evaluates student work. The methodology operates on three fundamental principles. First, independence ensures that judges are separate models from those powering the agents themselves, introducing external assessment and mitigating self-scoring bias. Second, expertise requires judges to possess sufficient reasoning capability and domain knowledge, typically reflected by high parameter counts serving as a proxy for reasoning ability. Third, rubric-based assessment provides explicit scoring guidelines across weighted quality dimensions, enabling consistent and interpretable evaluation.

The *LLM Judge framework* operates externally to the agentic framework developed in Chapter 6, serving purely as an evaluator without participating in the copilot's operation. This separation ensures evaluation objectivity and prevents contamination of development processes with assessment logic. The methodology employs multiple independent judges to enhance reliability and identify systematic bias patterns, enabling inter-rater reliability analysis that reveals whether different models consistently assess outputs similarly. Additionally, the framework includes self-assessment where the same model that generated an output also evaluates it, revealing how confident the model is about its own work quality. While self-assessment scores are not used as primary evaluation metrics due to inherent bias, comparing them against independent judge assessments provides insights into whether models accurately recognize when their outputs are strong or weak.

The assessment rubrics define weighted quality dimensions tailored to each agent's objectives. The Developer Agent rubric emphasizes functional correctness (25%), completeness (20%), code quality (20%), practical applicability (15%), error handling (10%), and explanation clarity (10%), prioritizing whether code works correctly and is maintainable. The Educational Agent rubric emphasizes pedagogical effectiveness (25%), content accuracy (20%), comprehensibility (20%), learning path coherence (15%), resource integration (10%), and actionability (10%), prioritizing teaching effectiveness and learning support. The specific model selection, judge configurations, and detailed rubric design are presented in section 7.4.

Complementing the functionality evaluation, the *automated code quality framework* operationalizes REQ-3-3 by assessing generated code through three weighted dimensions that combine to produce an overall quality score:

- **Semantic correctness (40%)** evaluates code logic and type safety through static analysis, detecting type errors and logical flaws without executing the code. A maximum score is assigned to error-free code, with penalties applied proportionally to the number and severity of detected issues, ensuring code is logically sound before runtime validation occurs.
- **Maintainability (35%)** utilizes the Maintainability Index (MI) and Cyclomatic Complexity (CC), as defined in MSC-3-3 and section 2.6. The MI provides a composite measure based on code volume, complexity, and comment density, while CC quantifies the number of independent paths through the code, together characterizing how difficult code will be to understand, modify, and extend.
- **PEP-8 compliance (25%)** identifies violations of Python's official style guide [35], normalizing violation counts against lines of code to account for implementation length.

These weights prioritize correctness and maintainability over stylistic conventions, reflecting the practical priorities of production code development. The final quality score is computed through weighted averaging of the three normalized dimension scores, producing a composite metric enabling systematic comparison across different implementations and agents. Complete scoring formulas, tool configurations, and implementation details are presented in subsection 7.2.4.

5.4.3. Manual Case Study Approach

Case studies provide detailed expert verification of agent performance on industrially-relevant applications, confirming or challenging preliminary conclusions from automated evaluations. By manually comparing agent outputs against expert-approved reference solutions, the methodology draws conclusions about both output quality and the rigor of automated verification procedures. Case studies are selected for complexity, feature breadth, and representation of production development patterns encountered in actual ParaPy projects.

Functional analysis compares implementations against expert-developed reference solutions to verify REQ-3-2, examining both overall system behaviour and component-level correctness. Expert review by experienced ParaPy developers assesses architectural appropriateness, idiomatic framework usage, and maintainability considerations that automated metrics may not fully capture. Comparison of expert scoped development time and copilot generation time support evaluation of REQ-3-1. This approach combines automated metrics with expert judgment, where experts evaluate code quality aspects that are difficult to measure automatically and verify that the automated assessment frameworks produce accurate results. The specific use cases and the evaluation of functional correctness, expert quality scores, and automated quality scores are presented in section 7.5.

5.4.4. User Testing Protocol

User testing validates the framework through controlled studies where developers complete representative tasks with and without AI assistance, directly addressing REQ-4-1 through REQ-4-3. This approach moves beyond controlled evaluation scenarios to assess practical effectiveness in realistic development workflows using the CLI application described in section 6.8. Participants are stratified by expertise level: experts possess extensive ParaPy development experience while novices have general programming competence but limited ParaPy exposure. This stratification tests whether the system reduces knowledge requirements as specified in REQ-4-3 while benefiting experienced developers.

The within-subjects design enables direct comparison of individual performance while controlling for ability differences. Participants complete two equivalent tasks in counterbalanced order (one manually and one with AI support), controlling for learning effects and task difficulty variance. Task equivalence is ensured through careful design matching required ParaPy features, geometric complexity, and expected completion time.

Data collection encompasses three dimensions aligned with research objectives. Temporal metrics capture total completion time, time to first successful execution, and debugging duration, directly supporting evaluation of REQ-3-1 and REQ-4-1. Productivity metrics quantify code generation rate and feature implementation velocity. Quality metrics assess functional correctness through expert solution comparison (REQ-3-2), code quality through the automated scoring framework described in subsection 7.2.4 (REQ-3-3), and framework-specific correctness through ParaPy-specific error analysis. These metrics combine to produce the composite score specified in REQ-3-4.

Structured interviews and session recordings provide qualitative context for interpreting quantitative results. Post-task questionnaires capture subjective assessments of task difficulty, perceived AI helpfulness, and solution confidence. Session recordings enable analysis of interaction patterns, prompt engineering strategies (based on guidance provided by the framework), and integration of AI suggestions into workflows. The complete methodology, protocols, and results are presented in section 7.6.

5.5. Summary & Conclusion

This chapter has presented a systematic approach to developing and evaluating an AI-powered coding assistant for ParaPy-based KBE development. The methodology employs a dual-agent architecture that addresses distinct user requirements through specialized agents: a Developer Agent optimized for code generation tasks and an Educational Agent focused on pedagogical support. Knowledge infrastructure employing semantic search compensates for sparse ParaPy-specific training data through dynamic information retrieval from curated documentation, examples, and API references. Code quality is ensured through progressive tool-based verification across syntactic and runtime correctness. A four-tiered evaluation strategy comprising unit testing, automated evaluation, manual case studies, and user testing provides assessment while balancing automation efficiency with qualitative expert judgment.

The methodology directly addresses all four research objectives established in Chapter 2. The dual-agent architecture with flexible deployment options (local via Ollama or remote via privacy-compliant API services) satisfies the first objective concerning local execution capability and tailored KBE support. The progressive verification tools and automated quality assessment frameworks operationalize the second objective's quality thresholds for syntactic (REQ-2-1), runtime (REQ-2-2), and functional correctness (REQ-2-3). The evaluation strategy defines and measures the performance indicators specified in the third objective, including development time reduction (REQ-3-1), functional equivalence (REQ-3-2), code quality (REQ-3-3), and composite performance scores (REQ-3-4). Finally, the user testing protocol enables assessment of the fourth objective concerning effectiveness across varying expertise levels (REQ-4-1 through REQ-4-3).

The following chapter, Chapter 6, details the concrete implementation of this methodological framework, presenting the specific design decisions, tool implementations, and system configurations that realize the approach described here. Subsequent evaluation in Chapter 7 demonstrates how this methodology enables rigorous assessment of the framework's performance against the measurable success criteria defined in the research framework.

6
Agentic Framework

This chapter presents the implementation of the dual-agent framework initially proposed in Chapter 4 and further elaborated in Chapter 5. The design builds on insights from the literature review (Chapter 3) and the industrial study to develop a solution that addresses the challenges outlined in Chapter 2.

The chapter is structured as follows: section 6.2 introduces the overall framework architecture and foundational implementation components. section 6.3 presents baseline performance of default models for reference. Subsequent sections provide detailed implementation insights to support reproducibility:

- section 6.4 outlines the knowledge infrastructure and semantic search functionality integrated into the framework.
- section 6.5 details the prompt engineering process.
- section 6.6 describes the tools developed within the framework.
- section 6.7 synthesizes the preceding elements into the final dual-agent system.
- section 6.8 introduces the Command Line Interface (CLI) application built for improved usability.
- section 6.9 concludes the chapter.

Note on Implementation

The full implementation of the framework is available for inspection on GitHub, upon request, due to privacy and data protection constraints. The co-pilot tool is under continuous development, with features and bug fixes being added beyond the scope of this thesis. As new requirements or usability improvements emerge, they will be incorporated to enhance user experience and maintain relevance within both industrial and academic contexts. For consistency and reproducibility, the results and implementation discussed in this chapter are based on version `0.4.0` of the co-pilot tool.

6.1. Assumptions & Limitations

This section outlines the key assumptions underpinning the framework implementation and acknowledges its inherent limitations. These considerations inform both the design decisions presented in this chapter and the evaluation methodology discussed in Chapter 7.

Assumptions

The framework implementation relies on the following assumptions:

- **Model Capabilities**: Selected models support the necessary communication protocols and, for local deployment via Ollama, include tool-calling capabilities through appropriate fine-tuning. API providers maintain service availability and adhere to stated data retention policies.
- **Documentation Representativeness**: The indexed ParaPy documentation (version 1.14.0) remains representative of current SDK functionality, and the semantic search indices accurately capture the essential knowledge required for ParaPy development.

Limitations

The framework exhibits the following limitations, some of which stem from explicit scope boundaries defined in Chapter 2:

- **Model Selection Constraints**: As outlined in REQ-1-1 and discussed in section 5.1, privacy and data protection requirements restrict model selection to locally deployed open-source models or approved API providers (Anthropic and Groq). This constraint precludes the use of other potentially capable commercial models.
- **Index Maintenance**: Semantic search indices require manual regeneration when ParaPy SDK updates introduce new primitives, modify existing APIs, or restructure documentation. The framework does not automatically detect or adapt to SDK version changes.
- **Execution Security**: The subprocess-based runtime validation system (section 6.6) executes arbitrary Python code on the host machine, introducing potential security vulnerabilities.
- **Deterministic Configuration**: All examples and evaluations use a model temperature of 0.0 to ensure reproducibility. While this maximizes consistency for comparative analysis, it eliminates sampling diversity and may not reflect optimal configurations for all use cases. Temperature variations primarily affect non-critical aspects such as variable naming and stylistic choices rather than functional correctness.
- **Execution Constraints**: Runtime code validation imposes a default 120-second timeout to prevent infinite loops. Complex or computationally intensive ParaPy models may exceed this limit, resulting in false negatives during validation.

6.2. System Architecture Overview

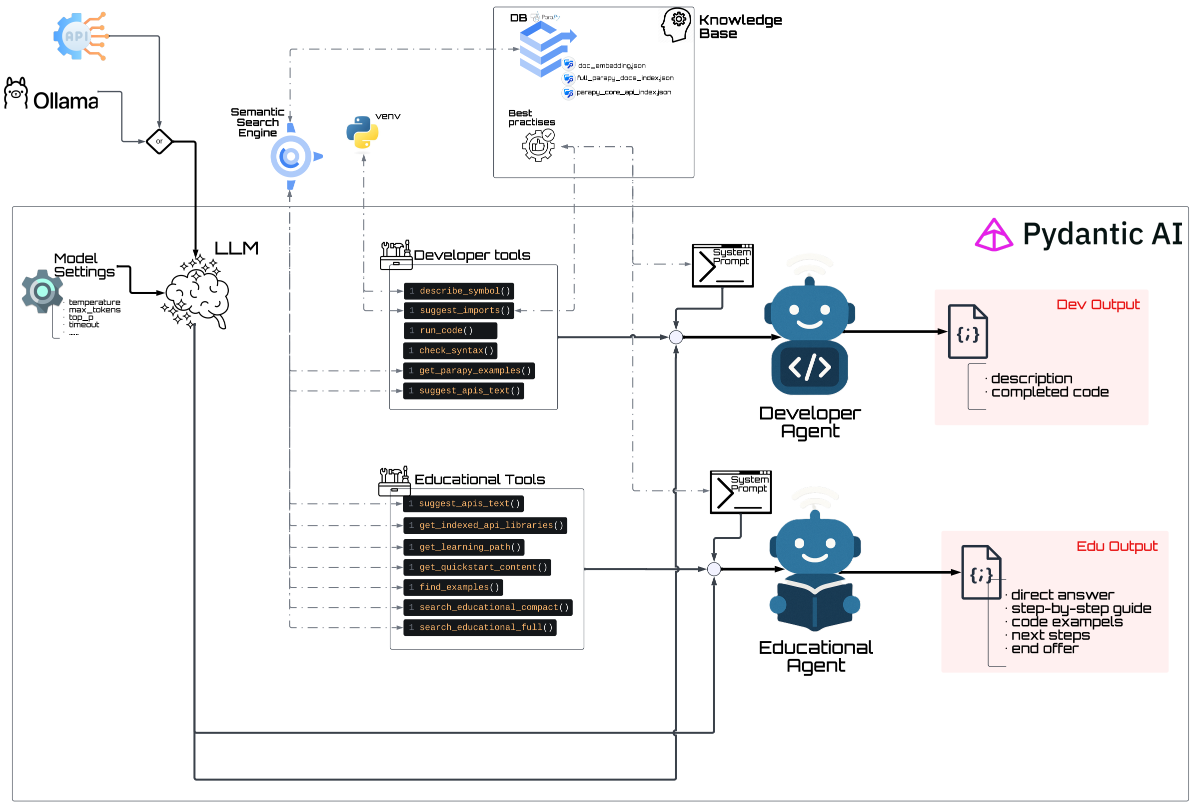


Figure 6.1: Top-level architecture of the dual-agent framework, showing the four main subsystems and the scope boundaries of the Pydantic-based integration layer. Dashed (grey) arrows denote communication flows between elements while the solid (black) arrows denote logic flow.

Figure 6.1 presents a high-level system perspective of the implemented dual-agent framework. It serves as a conceptual foundation for this chapter and a recurring point of reference in subsequent sections.

The framework comprises four main subsystems, aligned with the agent creation pipeline introduced in section 5.2:

1. LLM deployment and customization (top left),
2. Prompt engineering (top section of each agent),
3. Knowledge framework and semantic search (top middle), and
4. Tool development and deployment (center).

As outlined in section 5.3, the framework consists of two AI agents that share a common architectural foundation but differ in implementation and intended use cases. The *Developer Agent* assists with code completion, generation, and debugging for ParaPy SDK development. The *Educational Agent* focuses on learning-oriented tasks and provides documentation-related support for KBE and ParaPy concepts.

The primary framework used in this research is Pydantic AI, which serves as the integration layer for all subsystems. Figure 6.1 highlights scope boundaries, indicating which components fall within the Pydantic framework and which remain external. Tool usage, agent instructions, structured outputs, model architectures, and agent configurations are encapsulated within the framework. Components such as LLM hosting, the semantic search engine, and the knowledge base exist outside the Pydantic framework but maintain full compatibility. The foundational principles of Pydantic AI and custom extensions introduced in this work are discussed in section 12.1.

The example in Figure 6.2 demonstrates how these components integrate during a successful Developer Agent run. The first step involves model initialization and tool registration (Step 1a), handled automatically by the Pydantic framework. In parallel, the model receives custom system prompts designed specifically for ParaPy development (Step 1b). The model then invokes tools as needed; tool design, implementation, and input/output schemas are defined within this work, while tool call execution and output injection into model context (Steps 2–9) are managed by the Pydantic framework. The agent verifies its output for syntactic and runtime correctness using custom validation functions before producing a structured response that adheres to developer-defined schemas. The Pydantic framework enforces structured output formats; developers define expected types and validation logic. The deployment of the agents is revisited in more detail in section 6.7.

In summary, Pydantic AI manages communication and orchestration between components; developers define and implement individual building blocks.

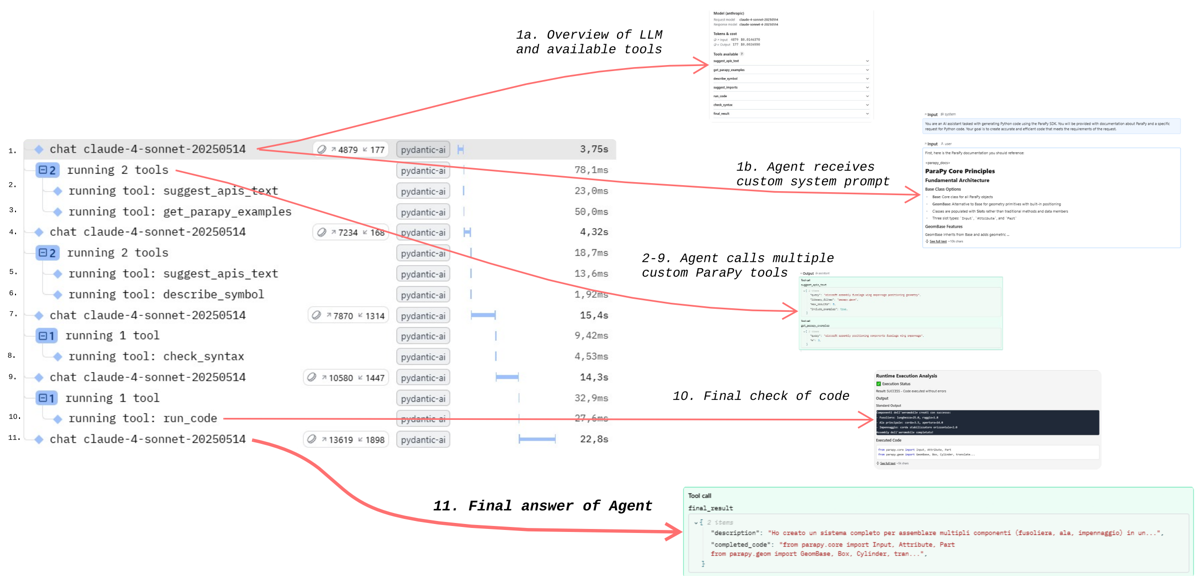


Figure 6.2: Example Developer Agent run trace from Logfire, illustrating tool availability, custom system prompts, and structured output parsing.

6.3. Baseline Performance

This section establishes baseline performance metrics for the two model configurations used throughout the framework: LLaMa 3.1 8B (local) and Claude Sonnet 4 (API-based). Baseline results employ minimal prompts without system instructions, tool integrations, or structured outputs, isolating native model capabilities before enhancement through prompt engineering, semantic search, and verification tools.

6.3.1. Model Configurations

More extensive information on the models, including their capabilities, training processes, and data, is presented in section 12.2.

LLaMa 3.1 8B (Local)

Meta AI - LLaMa 3.1

Version:	llama3.1:8b-instruct-q8_0	**Architecture:**	Dense transformer
Training Data:	Public data from the Web	**Languages:**	8 languages
Parameters:	8B	**Cutoff:**	End of 2023
Context:	128K		

The local configuration uses the `llama3.1:8b-instruct-q8_0` variant, an instruction-tuned model with tool-calling capabilities. While baseline evaluations do not require tool calling, this variant is used consistently across all development phases to enable direct comparison with the fully implemented agent. The LLaMa family was selected because it represents one of the few small-parameter models capable of local execution while supporting tool calling, a capability requiring dedicated fine-tuning as described in subsection 5.2.4. Although specialized coder models often yield superior performance, those currently supported by Ollama lack tool-calling capabilities, making LLaMa 3.1 the optimal trade-off between compatibility and performance for local deployment.

Claude Sonnet 4 (API)

Anthropic - Claude Sonnet 4

Version:	claude-sonnet-4-20250514	**Architecture:**	Transformer
Training Data:	Internet corpora + proprietary	**Languages:**	Not disclosed
Parameters:	Not disclosed (> 100B)	**Cutoff:**	May 2025
Context:	200K		

The API configuration uses Claude Sonnet 4, featuring a 200,000-token context window and 64,000-token output capacity. Released in May 2025 as part of the Claude 4 family, the model's parameter count, architectural details, and supported languages remain undisclosed, following common practices for proprietary models.[1]

6.3.2. Developer Agent

The Developer Agent baseline uses the following prompt:

```
>>> "Generate a parametric model of a simple Box using the ParaPy SDK in Python."
```

For reference, a correctly implemented ParaPy code snippet aligned with this prompt:

```python
from parapy.core import Input, Part
from parapy.geom import GeomBase, Box


class ParametricBox(GeomBase):
    width: float = Input()
    length: float = Input()
```

[1] *"Introducing Claude 4"*, from https://www.anthropic.com/news/claude-4 [accessed 06-10-2025]

```
    height: float = Input()

    @Part
    def solid(self) -> Box:
        return Box(width=self.width, length=self.depth, height=self.height)


if __name__ == "__main__":
    from parapy.gui import display

    obj = Box(width=1, length=1, height=1)
    display(obj)
```

Both models demonstrate the domain-specific knowledge gap identified in the industrial case study (section 4.3). LLaMa 3.1 produces syntactically valid Python but fails to apply ParaPy SDK principles correctly. While the model partially recognizes geometric aspects (width, length, height inputs, inheritance from geometric base classes, CAD operations), the implementation contains hallucinated imports, fabricated methods such as `pg.get_scene()`, and treats parametric class inputs as regular Python methods rather than ParaPy's declarative Attribute slots.

```
import parapy.gui as pg
from parapy.core import Attribute, Field, Input, Output
from parapy.geom import GeomObject, Extrusion

class SimpleBox(GeomObject):
    def __init__(self):
        super(SimpleBox, self).__init__()

    @Input
    def length(self):
        return 1.0

    @Input
    def width(self):
        return 1.0

    @Input
    def height(self):
        return 1.0

    @Output
    def box(self):
        return Extrusion(
            curve=GeomObject.line([0, 0, 0], [self.length(), 0, 0]),
            start=self.width() / 2,
            end=-self.width() / 2,
            direction=[0, 1, 0]
        )

    @Output
    def box_edges(self):
        return [
            GeomObject.line([0, 0, 0], [self.length(), 0, 0]),
            GeomObject.line([self.length(), 0, 0], [self.length(), self.width(), 0]),
            GeomObject.line([self.length(), self.width(), 0], [0, self.width(), 0]),
            GeomObject.line([0, self.width(), 0], [0, 0, 0])
        ]

if __name__ == '__main__':
    app = pg.get_scene()
    box = SimpleBox()
    app.add(box)
```

Claude Sonnet 4 generates code more closely resembling ParaPy syntax, with correct class definitions and geometric parameter identification. However, runtime errors occur in the ParaPy GUI, including incorrect `Point` type parsing for `location` arguments. The model lacks trained knowledge of the ParaPy SDK and demonstrates no awareness of the CAD kernel connection underlying the framework.

```python
from parapy.core import *
from parapy.geom import *


class SimpleBox(GeomBase):
    """
    A parametric box model with configurable dimensions and position.
    """

    # Input parameters
    length = Input(10.0, validator=val.Positive())
    width = Input(5.0, validator=val.Positive())
    height = Input(3.0, validator=val.Positive())

    # Position parameters
    position = Input(Point(0, 0, 0))

    # Optional parameters
    centered = Input(True)  # Whether the box is centered at the position

    @Attribute
    def corner_point(self):
        """Calculate the corner point based on centering option."""
        if self.centered:
            return Point(
                self.position.x - self.length / 2,
                self.position.y - self.width / 2,
                self.position.z - self.height / 2,
            )
        else:
            return self.position

    @Part
    def box(self):
        """The main box geometry."""
        return Box(
            length=self.length,
            width=self.width,
            height=self.height,
            position=translate(self.corner_point, "x", 0, "y", 0, "z", 0),
        )

    # Additional geometric properties as attributes
    @Attribute
    def volume(self):
        """Calculate the volume of the box."""
        return self.length * self.width * self.height

    @Attribute
    def surface_area(self):
        """Calculate the surface area of the box."""
        return 2 * (
            self.length * self.width
            + self.length * self.height
            + self.width * self.height
        )

    @Attribute
    def diagonal_length(self):
        """Calculate the space diagonal of the box."""
        return (self.length**2 + self.width**2 + self.height**2) ** 0.5


# Example usage and test cases
if __name__ == "__main__":
    from parapy.gui import display

    # Create a simple box instance
    simple_box = SimpleBox(
        length=20.0, width=10.0, height=5.0, position=Point(0, 0, 0), centered=True
    )
```

```

    display(simple_box)
```

These results confirm the persistent domain-specific knowledge gap discussed in section 4.3, where models trained on open-source code and general programming examples lack exposure to specialized frameworks like ParaPy. This limitation manifests as hallucinated syntax, violated design rules, and failure to follow ParaPy's grammar and semantics.

6.3.3. Educational Agent

The Educational Agent baseline uses the following prompt:

```
>>> "How can I create an L-shaped flange extrusion using the ParaPy SDK?"
```

Without semantic search or pedagogical prompts, both models misinterpret the educational intent and default to code generation. LLaMa 3.1 produces explanatory text followed by code containing hallucinated methods, incorrect class structures, and non-existent primitives. Claude Sonnet 4 generates more detailed explanations but similarly provides code snippets with runtime errors and incorrect SDK usage. Neither model recognizes that educational queries require documentation-grounded explanations rather than speculative code generation.

Both models fail to provide learning-oriented support. Responses lack documentation references, omit concept explanations, and do not guide users toward correct implementation patterns. This behavior reinforces findings from the industrial study (subsection 4.3.1) that general-purpose models cannot adequately support domain-specific development without targeted augmentation. Complete baseline responses are available in subsection 12.6.1.

The following sections address these limitations through semantic search infrastructure (section 6.4), targeted prompt engineering (section 6.5), and progressive verification tools (section 6.6).

6.4. Knowledge Infrastructure & Semantic Search

As discussed in the previous section, the primary limitation of commercial off-the-shelf LLMs lies in their inability to reliably generate code involving domain-specific frameworks. While fine-tuning is explicitly out of scope (see Chapter 2), two viable strategies remain: enriching the system prompt with additional context and equipping agents with tools to retrieve external information. Both approaches are pursued in this work.

The framework implements a semantic search engine to provide dynamic access to ParaPy-specific knowledge, as introduced conceptually in subsection 5.2.3. The implementation employs the `SentenceTransformers` library with the `all-MiniLM-L6-v2` embedding model, which maps text into a 384-dimensional vector space. Three distinct search engines were developed, each optimized for different information types:

- `SphinxDocSearcher` retrieves content from the complete ParaPy SDK documentation (version 1.14.0), including tutorials, guides, and conceptual descriptions. The indexing system parses Sphinx-generated output and preserves semantic structure through content classification and strategic metadata weighting.
- `ExampleSearcher` provides access to curated ParaPy code snippets extracted from the documentation source tree. AST-based docstring parsing generates descriptive labels, while vector normalization ensures representational fairness between short and long examples.
- `APISemanticSearcher` indexes callable objects from core ParaPy libraries (`core`, `geom`, `exchange`, `mesh`) through runtime introspection. Each entry includes qualified names, signatures, and docstrings, with dual indexing (dot-separated and space-separated forms) to improve matching across different query styles.

All indices support optional encryption to address data privacy requirements outlined in section 5.1. Indices are pre-built, encrypted, and stored as JSON files to avoid repeated computational overhead. At runtime, the Pydantic framework injects these search engines as dependencies into agent context through dedicated dataclasses, enabling tools to query indexed content on demand.

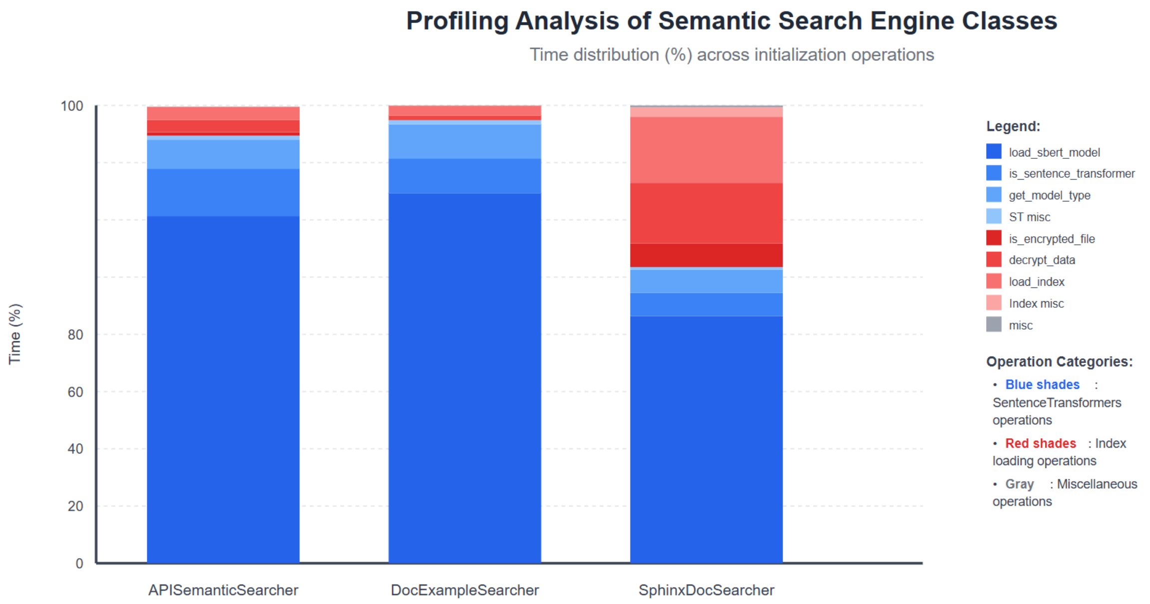


Figure 6.3: Time distribution of initialization operations across three semantic search engine classes. Model loading operations (blue) dominate initialization time, with `SphinxDocSearcher` showing notably higher index loading overhead (red) compared to the other implementations.

Runtime profiling (Figure 12.2) reveals that initialization overhead is dominated by embedding model loading, followed by index decryption and loading. The `SphinxDocSearcher` exhibits the highest overhead due to its larger index size. To minimize query latency, all engines are instantiated once during agent startup rather than at each tool invocation. While this introduces a modest initial delay, it significantly reduces inference-time costs by avoiding redundant processing.

Detailed implementation specifications, including filtering strategies, indexing algorithms, mathematical formulations, and integration patterns, are provided in section 12.3. The integration of these search engines into the agent toolset is discussed in section 6.6.

6.5. Prompt Engineering Implementation

This section details the implementation of system prompts for both agents, building on the methodological foundation established in subsection 5.2.2. The design adopts Anthropic's prompt engineering principles [79], including clear instructions, template structures, illustrative examples, chain-of-thought reasoning, XML-like formatting, and role prompting (for a summary, see section 11.2). Both agents share a common architectural foundation while incorporating content tailored to their distinct purposes: code generation for the Developer Agent and pedagogical support for the Educational Agent.

System prompts operate as templates with runtime substitution. Template placeholders (marked with `$variable_name`) are replaced with actual content when the agent processes a user query. This enables dynamic injection of retrieved documentation, curated examples, and user requests while maintaining consistent prompt structure. The substitution process employs simple Python string templates, that merge all relevant information before feeding the complete prompt to the respective agent. Both agents incorporate four foundational components that establish their operational framework:

Role Prompting Each agent receives a distinct identity aligned with its purpose. The Developer Agent is positioned as a code generation assistant, while the Educational Agent functions as a learning guide for KBE and ParaPy development. Role definitions appear at the beginning of each system prompt and frame subsequent instructions.

Chain-of-Thought Reasoning Both agents employ multi-step reasoning strategies before generating outputs. The Developer Agent follows: *analyze* → *identify* → *plan* → *map* → *check* → *consider*, while the Educational Agent uses: *analyze* → *identify* → *determine* → *plan*. These thinking steps are explicitly specified in the system prompt and guide the model through structured problem decomposition.

Best Practices Configurable guidelines establish coding standards, validation strategies, and interaction patterns. Best practices are provided through template substitution (the `$best_practises` placeholder) and define agent behaviour boundaries. These guidelines serve as a baseline but can be customized for specific organizational contexts during production deployment.

Guardrails Boundary conditions define queries outside agent scope and specify appropriate redirection responses. Guardrails prevent agents from attempting tasks beyond their expertise or knowledge domains.

6.5.1. Developer Agent Prompt Structure

The Developer Agent system prompt assembles multiple components through template substitution to optimize code generation performance. The complete template is provided in section 12.4. The template includes five runtime-substituted components:

- `$documentation`: The `PARAPY_PRINCIPLES` constant (see subsection 12.4.3) summarizes fundamental architecture, slot types, syntax patterns, and constraints. This provides conceptual grounding before presenting code examples.
- `$examples`: The `PARAPY_EXAMPLES` constant (see subsection 12.4.3), containing curated code snippets demonstrating ParaPy syntax patterns. These examples are distinct from the semantic search results; they provide static pattern recognition templates embedded directly in the prompt, focusing on core grammatical structures like Input, Attribute, and Part definitions.
- `$best_practises`: The `DEVELOPER_BEST_PRACTISES` constant (see subsection 12.4.2), which specifies implementation rules, message passing syntax, data integrity requirements, validation strategies, and code quality standards. These guidelines establish operational boundaries for code generation.
- `$request`: The user's query, appended verbatim at runtime. May optionally include file contents provided through the CLI application (see section 6.8).

Best Practices Implementation

The Developer Agent best practices (subsection 12.4.2) establish critical implementation constraints that address ParaPy's unique requirements. Key guidelines include:

- **Part Grammar Restrictions**: Enforces that Part methods contain only an optional docstring and return statement, moving complex logic to Attribute slots. This addresses ParaPy's declarative programming model and lazy evaluation requirements.
- **Message Passing Syntax**: Specifies correct formatting for `pass_down` and `map_down` operations, which are essential for ParaPy's dependency tracking.
- **Data Integrity**: Prohibits in-place mutations that break dependency tracking, a critical constraint for maintaining ParaPy's reactive evaluation model.
- **Import Practices**: Explicitly prohibits wildcard imports to improve code clarity and maintainability.
- **KBE Mindset**: Emphasizes parametric thinking, systematic decomposition, and architecture planning, reflecting Knowledge-Based Engineering principles of automation, reusability, and knowledge capture.

Notably, PEP-8 compliance is not explicitly listed in the best practices, despite being an evaluation criterion (REQ-3-3). The decision to rely on models' pretrained knowledge of Python style guidelines rather than explicit instruction reflects a design trade-off between prompt length and specificity. This discrepancy is discussed further in Chapter 9.

ParaPy Principles and Examples

The `PARAPY_PRINCIPLES` constant (subsection 12.4.3) provides a structured explanation of ParaPy's core concepts, including Base and GeomBase classes, slot types, positioning mechanisms, quantification, and grammar constraints. This content functions as a condensed reference manual embedded in the prompt, enabling the model to understand ParaPy-specific patterns without requiring extensive pretraining on the framework.

The `PARAPY_EXAMPLES` section within the same constant presents seven curated code snippets demonstrating correct syntax for Input, Attribute, and Part definitions, including advanced features like quantification and DynamicType. These examples serve as pattern recognition templates, enabling the model to mimic correct grammatical structures. However, positioning and orientation examples are limited to basic transformations (`translate`, `rotate90`). The semantic search infrastructure (section 6.4) may provide additional positioning examples if relevant to the user query, but coverage depends on query formulation and index content. This limitation in static positioning examples potentially contributes to observed positioning errors in generated code, as discussed in Chapter 7.

6.5.2. Educational Agent Prompt Structure

The Educational Agent system prompt prioritizes pedagogical effectiveness over code generation, reflecting distinct design objectives. The complete template is provided in section 12.5. The Educational Agent template includes three runtime-substituted components:

- `$request`: The user's query, presented at the beginning of the prompt to establish context.
- `$best_practises`: The `EDUCATIONAL_BEST_PRACTISES` constant (see subsection 12.5.2), which defines response guidelines, learning support strategies, scope management, and adaptive communication patterns.
- No `$documentation` or `$examples` placeholders appear in the template. Early experiments demonstrated that the Educational Agent achieves satisfactory performance without static code examples embedded in the prompt, instead relying entirely on semantic search tools for dynamic content retrieval during execution.

The Educational Agent's reliance on runtime semantic search rather than static prompt content reflects a fundamental design difference from the Developer Agent. As detailed in subsection 5.2.3, the Educational Agent uses `SphinxDocSearcher` to access the complete ParaPy SDK documentation and `APISemanticSearcher` for API references. These search engines are invoked dynamically through tool calls when the agent requires documentation context, enabling access to significantly more content than could be embedded in a static prompt. This architecture explains how the Educational Agent can reference specific documentation sections and provide detailed examples without explicit `$documentation` placeholders in its template.

Best Practices Implementation

The Educational Agent best practices (subsection 12.5.2) establish pedagogical guidelines that distinguish it from the Developer Agent:

- **Core Documentation and Accuracy**: Emphasizes basing responses on retrieved documentation, explicitly stating information gaps when queries exceed available knowledge, and providing actionable steps tailored to user experience level.
- **Knowledge-Based Engineering Mindset**: Emphasizes declarative design capture where relationships are defined through `Attribute` slots that automatically update when dependencies change, rather than procedural calculation sequences that require manual recalculation. This enables parametric models where engineering intent is preserved through ParaPy's dependency tracking and lazy evaluation.
- **Adaptive Communication**: Instructs the agent to match response complexity to user expertise, identify prerequisite knowledge gaps, and prioritize learning paths for overwhelmed users.
- **Learning Support**: Normalizes learning difficulties, sets realistic expectations, and provides focused starting points rather than comprehensive overviews.
- **Scope Management**: Distinguishes between ParaPy SDK capabilities and broader programming or CAD tool questions. This complements the explicit guardrails in the system prompt; while guardrails provide templated redirection responses for out-of-scope queries, scope management

guidelines inform the agent's reasoning about categorization and appropriate depth of response for boundary cases.

- **Practical Guidance**: Prioritizes actionable advice with specific next steps, relevant code examples from documentation, and logical follow-up topics for continued learning.

The KBE mindset guidelines assume LLMs possess understanding of caching, dependency tracking, and lazy evaluation through semantic search retrieval of relevant documentation. The educational prompt does not explicitly enumerate these concepts because they are covered in the indexed ParaPy documentation that agents access through tools. When queries involve these topics, the agent retrieves appropriate documentation sections and explains concepts based on retrieved content. The effectiveness of this approach is evaluated in Chapter 7 and discussed in Chapter 9.

6.5.3. Prompt Engineering Impact

To demonstrate the effectiveness of enhanced system prompts, both agents were re-evaluated using the same baseline queries from section 6.3. These results illustrate intermediate performance before tool integration and semantic search.

Developer Agent with Prompt Engineering

For the Developer Agent, the baseline prompt "Generate a parametric model of a simple Box using the ParaPy SDK in Python" was re-evaluated with the complete system prompt structure.

Claude Sonnet 4 (API):

```python
from parapy.core import Input
from parapy.geom import Box, GeomBase


class ParametricBox(GeomBase):
    """
    A parametric box model with configurable dimensions.

    This class creates a box geometry with user-defined width, length, and height.
    Inherits positioning capabilities from GeomBase for easy placement and orientation.
    """

    # Input parameters for box dimensions
    width = Input(
        1.0,
        validator=lambda x: x > 0,
        label="Width",
        doc="Width of the box in the X direction",
    )

    length = Input(
        1.0,
        validator=lambda x: x > 0,
        label="Length",
        doc="Length of the box in the Y direction",
    )

    height = Input(
        1.0,
        validator=lambda x: x > 0,
        label="Height",
        doc="Height of the box in the Z direction",
    )

    @Part
    def box(self):
        """The main box geometry"""
        return Box(
            width=self.width,
            length=self.length,
            height=self.height,
            position=self.position,
        )
```

```


if __name__ == "__main__":
    from parapy.gui import display

    # Create a parametric box with custom dimensions
    my_box = ParametricBox(width=2.0, length=3.0, height=1.5)
    display(my_box)
```

LLaMA 3.1 8B (Local):

```
from parapy.core import Base, Input, Attribute
from parapy.geom import Box

class SimpleBox(Base):
    length = Input(1)
    width = Input(1)
    height = Input(1)

    @Attribute
    def volume(self):
        return self.length * self.width * self.height

    @Part
    def box(self):
        return Box(length=self.length, width=self.width, height=self.height)


if __name__ == '__main__':
    from parapy.gui import display
    obj = (SimpleBox(length=2, width=3, height=4))
    display(obj)
```

Both models demonstrate improved adherence to ParaPy grammar and semantic conventions compared to baseline performance. While neither implementation is fully executable, they avoid the elaborate, hallucinated attributes and methods that characterized baseline outputs. The more concise, focused responses indicate that targeted prompt engineering successfully guides models toward domain-specific patterns even without tool integration or semantic search.

Educational Agent with Prompt Engineering

The Educational Agent's intermediate performance with prompt engineering is presented in section 12.6 to avoid redundancy in the main text. Results demonstrate improved response structure and pedagogical focus compared to baseline configurations, though functional accuracy remains limited without semantic search access to actual documentation.

The next section describes the tool development that enables agents to access semantic search indices, validate code syntax and runtime correctness, and perform other specialized operations that enhance the prompt engineering foundation established here.

6.6. Tool Development & Deployment

Building on the semantic search infrastructure and prompt engineering foundations, this section describes the tools that enable agents to retrieve domain-specific information, verify code correctness, and enhance generation quality. Tools are organized into four functional categories corresponding to the distinct capabilities required by each agent:

The Developer Agent tools emphasize code generation support through access to curated examples (`ExampleSearcher`), API specifications (`APISemanticSearcher`), import suggestions, and validation mechanisms. The Educational Agent tools prioritize pedagogical content through comprehensive documentation access (`SphinxDocSearcher`) with flexible filtering by content type (tutorial, guide, API reference). Code verification tools (syntax checking via `check_syntax()` and runtime validation via `run_code()`) enable the Developer Agent to validate outputs before finalizing responses, implementing the progressive verification strategy outlined in subsection 5.2.4.

Table 6.1: Tool Categories and Functions Summary.

Category	Count	Tools	Primary Function
Data Retrieval (Developer)	4	DT01-DT04	Retrieve ParaPy examples and API interface specifications
Data Retrieval (Educational)	5	ET01-ET05	Retrieve ParaPy documentation with optional code examples
Code Improvement	2	DT05-DT06	Provide import suggestions and primitive metadata
Code Verification	2	DT07-DT08	Verify syntactic and runtime correctness

All tools follow a consistent architectural pattern separating core logic from presentation. Each tool implements domain-specific functionality (semantic search queries, code execution, syntax validation) and returns structured data objects. Wrapper functions transform these objects into formatted text suitable for prompt injection, enabling both programmatic access and agent-compatible output. This separation ensures maintainability and enables reuse across different presentation contexts. For example, `suggest_apis()` returns structured `APISuggestion` objects containing metadata, scores, and docstrings, while `suggest_apis_text()` formats these suggestions as readable text for agent consumption.

A complete tool inventory with technical specifications is provided in subsection 12.7.4. Detailed implementation walkthroughs for representative tools, including the API reference search (`suggest_apis`), runtime verification (`run_code`), and documentation search (`search_educational`), are presented in section 12.7. Full implementations with inline documentation are accessible via the `tools` subpackage.

The integration of these tools with system prompts, structured outputs, and semantic search dependencies forms the complete agent implementation described in the next section.

6.7. Agent Implementation

This section synthesizes the semantic search infrastructure (section 6.4), system prompts (section 6.5), and tool suite (section 6.6) into fully operational agents within the Pydantic AI framework. Both agents are instantiated via factory functions that configure model selection, system prompts, tool access, and structured output schemas based on agent role.

The Developer Agent receives access to tools DT01 through DT08, covering data retrieval from curated examples and API references, import suggestions, and progressive verification through syntax checking and runtime validation. Outputs conform to the `DevOutput` structure, which enforces inclusion of descriptive explanations, completed code, and documentation of problems encountered during execution. An output validator automatically invokes `run_code()` to verify runtime correctness before finalizing responses. If validation fails, a `ModelRetry` exception prompts code revision within configured retry limits. Complete output schema specifications, validation mechanisms, and runtime instantiation details are provided in section 12.8.

The Educational Agent accesses tools ET01 through ET05 for documentation retrieval, plus DT01 and DT03 for API reference and example searches. Outputs follow the `EduOutput` structure, requiring direct answers, optional step-by-step guides, relevant code examples with documentation references, suggested next steps, and concluding invitations for follow-up questions. This structure enforces pedagogical completeness while maintaining flexibility for varied query types. Unlike the Developer Agent, the Educational Agent does not employ output validation, as responses prioritize pedagogical clarity over executable completeness.

Both agents operate at temperature 0.0 for reproducibility, following the deterministic configuration outlined in section 6.1. Complete agent configurations, including dependency injection specifications and retry behavior, are implemented in the `agents` subpackage.

6.7.1. Performance Demonstration

The following examples demonstrate fully integrated Developer Agent performance using the baseline prompt from section 6.3. Both models now generate syntactically correct, runtime-executable ParaPy code adhering to SDK grammar and best practices, representing substantial improvement over baseline performance. Educational Agent results exhibiting similar improvements are provided in section 12.6, where both models demonstrate pedagogically structured responses grounded in retrieved documentation with correct API references and implementation guidance. Comprehensive evaluation across varying complexity levels for both agents is presented in Chapter 7.

```
>>> "Generate a parametric model of a simple Box using the ParaPy SDK in Python."
```

Developer Agent: LLaMa 3.1 8B (Local)

```python
from parapy.core import Base, Input, Attribute, Part
from parapy.geom import Box


class ParamBox(Base):
    width = Input(1)
    height = Input(1)
    length = Input(1)

    @Attribute
    def area(self):
        return self.width * self.height

    @Attribute
    def volume(self):
        return self.area * self.length

    @Part
    def box(self):
        return Box(pass_down="width, height, length")


if __name__ == '__main__':
    from parapy.gui import display
    obj = (Box(width=2, height=3, length=4))
    display(obj)
```

Developer Agent: Claude Sonnet 4 (API)

```python
from parapy.core import Base, Input, Attribute, Part
from parapy.geom import Box


class ParametricBox(Base):
    """A parametric box model with configurable dimensions and properties."""

    # Required input for width
    width = Input()

    # Optional inputs with default values
    length = Input(2.0)
    height = Input(1.0)

    # Optional color input
    color = Input("blue")

    @Attribute
    def volume(self):
        """Calculate the volume of the box."""
        return self.width * self.length * self.height

    @Attribute
    def surface_area(self):
        """Calculate the surface area of the box."""
        return 2 * (self.width * self.length +
                    self.width * self.height +
```

```
                    self.length * self.height)

    @Part
    def box(self):
        """The main box geometry."""
        return Box(width=self.width,
                   length=self.length,
                   height=self.height,
                   color=self.color)


if __name__ == '__main__':
    from parapy.gui import display

    # Create an instance with required width parameter
    obj = ParametricBox(width=3.0)
    display(obj)
```

Both agents are exposed to users through the CLI application described in the next section, which provides model selection, parameter configuration, and file upload capabilities while maintaining the single-turn interaction mode defined by the framework's architectural constraints.

6.8. CLI Application Design

To make the agentic framework more accessible to users developing KBE applications, a Command Line Interface (CLI) application was developed on top of the framework to enhance user interaction and to allow the copilot tool to run alongside the codebase under development. Although the CLI does not introduce additional capabilities, such as codebase awareness, file read/write permissions, or IDE integration, it significantly improves usability by abstracting away tedious setup steps, preventing runtime errors, and reducing initialization time. The application provides a structured configuration workflow that guides users through mode selection (Developer or Educational), model provider configuration (local via Ollama or remote via approved API providers), and optional model settings adjustments. Configurations are persisted locally, enabling streamlined quick-start functionality in subsequent sessions. A visual overview of the CLI's logical flow and communication with the underlying framework is presented in Figure 12.3.

During interactive sessions, users can leverage several utility commands to enhance their workflow. The CLI supports dynamic mode switching (`/mode`), model changes (`/model`), settings adjustment (`/settings`), and application restart (`/restart`) without terminating the session. A particularly useful feature is the `/file` command, which appends file contents directly to the agent's context, enabling analysis of specific code files. File contents persist across the session until explicitly cleared via the `/clear` command. All agent interactions are managed through a dedicated function that handles error propagation and formats structured agent outputs into human-readable console text.

With the completion of the Command Line Interface, the framework achieves full end-to-end functionality: from model configuration and agent interaction to user-facing deployment. The CLI operationalizes the framework for practical use while exemplifying its modular design philosophy, allowing seamless integration of local and remote models within a controlled environment. Importantly, all interactions follow a single-turn model: previous messages are neither recorded nor added to the agent's context for subsequent runs, maintaining stateless operation aligned with the framework's design. Comprehensive technical details of the CLI implementation, including the complete interaction flow, command specifications, and underlying functions, are provided in section 12.9. Having now detailed the architecture, implementation, and user interaction layers of the agentic framework, the following section concludes the chapter by reflecting on the key design outcomes, their alignment with the research objectives, and their implications for subsequent evaluation.

6.9. Summary & Conclusion

This chapter presented the design and implementation of a dual-agent framework for ParaPy SDK development, directly addressing the first two research objectives introduced in Chapter 1. Through a structured design process encompassing LLM deployment, knowledge infrastructure, prompt engineering, tool development, and user interface design, the framework delivers a solution to the challenges of domain-specific code generation within Knowledge-Based Engineering contexts. The specific requirements addressed in this chapter are summarized in Table 6.2.

Table 6.2: Requirements Traceability Matrix for research objectives 1 and 2.

Req.	Implementation	Reference
REQ-1-1	Custom `Agent` wrapper with unified local/API deployment supporting Ollama and allowed API providers, accessible through the CLI	section 12.1
REQ-1-2, REQ-1-3	Developer Agent with structured output (`DevOutput`), comprehensive toolset (DT01–DT08), semantic search integration, and ParaPy-specific prompt engineering	section 6.4, 6.5, 6.6, 6.7
REQ-2-1	Syntax verification via `check_syntax()` tool and ParaPy-specific prompt engineering with SDK examples and best practices	section 6.5, 6.6
REQ-2-2	Subprocess-based `run_code()` execution with mandatory output validation and ParaPy-specific prompt engineering with SDK examples and best practice	section 6.5, 6.6
REQ-2-3	Semantic search infrastructure through `ExampleDocSearcher` & `APISemanticSearcher` and associated tools, iterative refinement via `ModelRetry` mechanism	section 6.4, 6.6

Baseline experiments (section 6.3) established that both local (LLaMa 3.1) and API-based (Claude Sonnet 4) models exhibit fundamental limitations when applied to domain-specific frameworks without additional context, manifesting as syntax hallucinations and violations of ParaPy's declarative programming model. Enhanced prompt engineering (section 6.5) improved structural coherence but proved insufficient to eliminate hallucinated method calls. The full framework implementation (section 6.7), integrating semantic search, domain-specific tools, and output validation, successfully generated syntactically correct, runtime-executable ParaPy code in single-shot interactions, representing substantial improvement over baseline performance.

Beyond fulfilling the stated requirements, the chapter contributes several reusable technical components: (i) a modular, encryption-capable semantic search infrastructure with specialized engines for documentation, examples, and API references; (ii) a consistent tool architecture separating core logic from presentation; and (iii) a user-friendly CLI application abstracting framework complexity. These components are generalizable to other domain-specific code generation contexts beyond ParaPy.

The framework does exhibit certain limitations that warrant acknowledgment. It requires re-indexing for SDK updates, introduces security considerations through subprocess execution, and operates in single-turn mode without conversation history. More significantly, the performance improvements demonstrated in this chapter are based on illustrative examples rather than systematic evaluation. Quantitative assessment across metrics such as correctness rates, token efficiency, and latency is reserved for Chapter 7.

This chapter establishes the technical foundation for the evaluation phase. The next chapter systematically evaluates the framework's performance through controlled experiments, assesses alignment with defined requirements, and analyses behaviour across varying complexity levels to determine whether the framework meets quality thresholds necessary for practical deployment in industrial KBE development contexts.

7

Verification & Validation

This chapter evaluates the dual-agent framework described in Chapter 6 against the requirements and objectives outlined in Chapter 2. A comprehensive verification and validation strategy is applied to assess whether the framework achieves the intended improvements in KBE application development. The outcomes presented in this chapter serve as the foundation for the conclusion (Chapter 8) and recommendations (Chapter 9) chapters, where the usability and applicability of the framework will be critically assessed. It is in those chapters that a final judgment will be made as to whether the research gap identified in Chapter 2 has been (partially) addressed by the proposed approach.

To assess the framework's performance and determine whether the specified requirements have been met, a four-pronged evaluation strategy is employed. Building on the foundations laid in Chapter 5, the specific implementation of each verification and validation procedure is first described in section 7.2, with the aim of promoting transparency and reproducibility. Table 7.1 provides a summary and reference overview, mapping each requirement to its corresponding evaluation method and the section in which it is addressed. The chapter then proceeds with the following evaluation components:

1. **Unit Testing** (section 7.3): This initial verification approach tests core components (e.g., wrappers, functions) to ensure correctness, stability, and future maintainability of the codebase.
2. **Automated Evaluation** (section 7.4): Using the `Evals` package by PydanticAI, this step evaluates the agent's output in terms of syntax validity, runtime correctness, response accuracy, and code quality.
3. **Manual Case Studies** (section 7.5): To reinforce findings from the automated evaluation and address potential weaknesses, this section presents a case study from industrial settings. It includes manual validation of runtime behaviour, syntax and functional correctness, alongside expert reviews of code quality.
4. **User Testing** (section 7.6): This final approach focuses on verifying whether REQ-4-1, REQ-4-2, and REQ-4-3 are met in real-world usage. Through testing with actual users from industry, it is assessed whether the copilot tool effectively supports KBE application development in practice.

Drawing from all four evaluation components, particularly the user testing results, a synthesis of findings and key conclusions is presented in section 7.7. A *Requirements Traceability Matrix* is included at the end of the chapter in Table 7.11, summarizing the extent to which each requirement has been fulfilled based on the testing strategies outlined in Table 7.1.

7.1. Assumptions & Limitations

This section addresses the key assumptions and limitations encountered during the verification and validation process, which should be considered when interpreting the results presented in this chapter.

Methodological Assumptions

- **Determinism and Reproducibility**: Despite setting temperature to $T = 0.0$ for all evaluations, fully deterministic behaviour cannot be guaranteed due to inherent stochasticity in sampling mechanisms and floating-point operations. Minor output differences may occur across repeated runs.

- **Judge Model Selection**: The `LLMJudge` framework employs models with varying parameter counts. Qwen3 operates with only 32 billion parameters, below the preferred threshold for complex reasoning tasks, potentially impacting assessment reliability. LLaMa 3.1 – 8B was excluded from self-reflection due to insufficient reasoning capacity.
- **Enhanced Prompting for Smaller Models**: The LLaMa 3.1 – 8B model received an extended system prompt during automated evaluations to improve schema adherence. This guidance was not provided to Claude Sonnet 4, introducing methodological inconsistency that may affect direct comparability.

Table 7.1: Requirements Verification Test Plan

Req. ID	Test Scenarios	Success Criteria	Results
REQ-1-1	By implementation, not covered in this chapter	Local deployment via Ollama and API deployment via allowed providers	N/A
REQ-1-2	Evaluation Cases (developer) & Case Studies using a comprehensive suite of automatic evaluators for syntax correctness, runtime correctness, LLM-asserted quality and automated/expert reviewed code quality score	Minimum 85% successful completion rate for skeleton code structures	section 7.4, 7.5
REQ-1-3	Evaluation Cases (developer) & Case Studies using a comprehensive suite of automatic evaluators for syntax correctness, runtime correctness, LLM-asserted quality and automated/expert reviewed code quality score	Minimum 75% successful completion rate for generating class definitions from natural language specifications	section 7.4, 7.5
REQ-2-1	Automated syntax checking through `check_syntax` and `SyntaxEvaluator`	Generated code shall achieve syntactic correctness in 95% of test scenarios	section 7.4, 7.5, 7.6
REQ-2-2	Automated runtime checking through `run_code` and `RuntimeEvaluator`	Generated code shall execute without runtime errors in 80% of test scenarios	section 7.4, 7.5, 7.6
REQ-2-3	Manual comparison of functional correctness of Case Studies and User Testing against reference solutions. Automated testing by LLM in Evaluaton Cases (soft)	Generated code shall achieve intended functionality in 70% of test scenarios	section 7.5, 7.6
REQ-3-1	Comparison of case studies w.r.t manual solution and test cases in user testing	Primary performance improvements introduced by the co-pilot tool shall be measured in development time reduction	section 7.6
REQ-3-2	Manual comparison of functional equivalence of Case Studies and User Testing against reference solutions	Functional equivalence to manually completed reference solutions in 85% of test scenarios	section 7.5, 7.6
REQ-3-3	Manual expert review, or automated review through `QualityEvaluator` for evaluation cases, case studies and code generated from user testing	Expert quality score of 7.5 out of 10, using a standardized rubric covering semantic correctness (40%), maintainability (35%), and PEP-8 compliance (25%)	section 7.4, 7.5, 7.6
REQ-3-4	Case studies and user testing by implementation	Composite performance score of normalized development time improvement (40%), functional correctness rate (35%), and expert quality score (25%), with a minimum composite score of 75%	section 7.5, 7.6
REQ-4-1	Time metrics of case studies (partially) and user testing	Minimum development time reductions of 40% for novice users and 25% for expert users	section 7.5, 7.6
REQ-4-2	Time metrics (development time) and coding metrics (functional correctness and quality score) of user testing	Composite score improvements of minimum 30% for novice users and 15% for expert users	section 7.6
REQ-4-3	Time & code metrics and coded questions of user testing	Novice task completion rates equivalent to 80% of expert baseline performance and a 50% reduction in domain-specific knowledge errors compared to unassisted novice performance	section 7.6

User Testing Limitations

- **Task Comprehension as Part of Timed Sessions**: The 20-minute sessions included time for task comprehension, conflating understanding with execution efficiency. This may disadvantage participants less familiar with the problem domain, though it reflects realistic usage scenarios.
- **Sample Size and Composition**: Nine participants represent a limited sample size, restricting

statistical generalizability. ParaPy employees may introduce familiarity bias, and the controlled environment may have created social desirability effects.

- **Test Environment Constraints**: Testing under time pressure in monitored settings may not represent typical development workflows. Artificial constraints may have influenced decision-making patterns, particularly regarding code review depth.

Framework Limitations

- **Educational Agent Misuse**: During user testing, several participants used the Educational Agent to generate production code rather than relying on the Developer Agent, significantly impacting code quality metrics in affected cases.

These assumptions and limitations should be considered when evaluating the framework's compliance with specified requirements and when interpreting findings presented in Table 7.11. They also inform recommendations for future work in Chapter 9.

7.2. Test Framework Foundations

This section outlines the specific implementation of the verification and validation (V&V) methods employed in the remainder of this chapter. While Chapter 5 introduced the conceptual foundations of the evaluation strategy, the current section—and the chapter as a whole—focuses on how these methods have been operationalized in practice. In essence, this section serves as a design overview of the V&V suite, preceding the presentation and discussion of results in the subsequent sections. To reflect this role, the structure of this section mirrors the structure of the chapter itself.

Unless otherwise indicated, the full implementation of each method is included here. Additional supporting material, including outputs and logs too extensive for the main text, is made available in Part II and the appendices. The complete V&V codebase is accessible via the project repository on GitHub.

7.2.1. Unit Testing

Unit testing ensures that core components behave as intended and that newly added features do not compromise existing framework functionality. While unit tests do not directly address research requirements, they form the foundational verification layer, as further testing is meaningful only when core components function correctly. Moreover, unit testing represents standard software engineering practice that prevents breaking changes in production environments.

The framework employs `pytest` with a test suite structured to mirror the `src` folder hierarchy, enabling efficient execution of targeted tests in response to changes within specific sub-packages. Figure 13.1 illustrates the decision tree for test prioritization and selection based on code changes. Tests employ patching to isolate functionality and snapshot testing to verify exact output matches against reference implementations.

Because the framework relies heavily on large language models, traditional integration testing becomes impractical due to non-deterministic model outputs. The automated evaluations of the next section fill this gap through evaluation methods specifically designed for LLM-based systems. The full test suite and coverage report for the current framework (version `0.4.0`) are detailed in section 7.3. Complete implementation specifications, including concrete test examples, patching patterns, snapshot testing methodology, and the `TestModel` feature, are provided in section 13.1.

7.2.2. Automated Evaluation

The automated evaluation implements the framework described in section 5.4, executing both agents across comprehensive test scenarios. Test cases are stratified by complexity and task type, with complete specifications provided in section 13.4 and section 13.5. Evaluation employs two model configurations: *Claude Sonnet 4* (API-based) and *LLaMa 3.1 – 8B* (local via Groq API for accelerated testing, functionally equivalent to Ollama deployment). This dual-model approach enables assessment of behavioral differences between large API-accessible models and smaller local execution models.

Evaluation categories implement the four-dimensional assessment methodology: syntax validation via `check_syntax()`, runtime verification via `run_code()`, functional assessment via `LLMJudge`, and quality scoring via the automated quality framework. Structural conformance additionally validates output adherence to predefined schemas (`DevOutput`, `EduOutput`).

The `LLMJudge` and automated quality frameworks are detailed in subsection 7.2.3 and subsection 7.2.4 respectively, including specific model configurations, scoring implementations, and evaluation execution protocols. Results are presented in section 7.4.

7.2.3. Large Language Model as Judge Framework

This section details the specific implementation of the LLM Judge framework described in section 5.4. Table 7.2 presents the experiment configuration, specifying agent models, judge models, and temperature settings across eight evaluation scenarios encompassing both self-assessment and independent evaluation.

Table 7.2: Experiment Configuration Overview

Nr.	Experiment	Agent Model	LLM Judge Model	Agent Temperature	LLM Judge Temperature
1	self-reflection-claude	claude-4-sonnet-20250514	claude-4-sonnet-20250514	0.0	0.0
2	self-reflection-llama3.1	meta:llama-3.1-8b	NA	0.0	0.0
3	claude-gpt-variance	claude-4-sonnet-20250514	openai:gpt-oss-120b	0.0	0.0
4	llama3.1-gpt-variance	meta:llama-3.1-8b		0.0	0.0
5	claude-qwen3-variance	claude-4-sonnet-20250514	qwen:qwen-3-32b	0.0	0.0
6	llama3.1-qwen3-variance	meta:llama-3.1-8b		0.0	0.0
7	claude-llama3.3-variance	claude-4-sonnet-20250514	meta:llama-3.3-70b-versatile	0.0	0.0
8	llama3.1-llama3.3-variance	meta:llama-3.1-8b		0.0	0.0

Three judge models were selected from Groq's approved provider list: *GPT-OSS-120B*, *LLaMa 3.3 – 70B*, and *Qwen3 – 32B*. While *Qwen3 – 32B* falls below the preferred parameter threshold, it was included due to absence of higher-capacity alternatives within the approved environment. *LLaMa 3.1 – 8B* was excluded from self-assessment (experiment 2) due to insufficient reasoning capacity for evaluation tasks. While comprehensive multi-dimensional studies varying agent models, judge models, and temperature settings could provide deeper insights, computational and financial resource constraints necessitated focus on the most critical evaluation comparisons.

The complete judge system prompts implementing the rubrics defined in section 5.4 are provided in section 13.6 and section 13.7. Detailed rationale for model selection, temperature configuration, and requirement mappings for each evaluation criterion are documented in section 13.2.

7.2.4. Code Quality Framework

The automated code quality framework operationalizes REQ-3-3 through the three-dimensional assessment methodology described in section 5.4. Implementation employs industry-standard Python libraries: `mypy` for type checking, `pylint` for logical correctness, `radon` for maintainability metrics, and `pycodestyle` for PEP-8 compliance. Table 7.3 presents the complete scoring methodology, including tools, scoring rules, and dimension weights.

Tool configurations include ParaPy-specific adjustments to reduce false positives. For `mypy`, custom arguments handle dynamic ParaPy features and missing type stubs. For `pylint`, specific error codes triggered by unsupported ParaPy constructs are excluded from scoring. The final quality score combines normalized sub-scores through weighted averaging as specified in MSC-3-3. Complete implementation specifications, including tool configurations, scoring formulas, and ParaPy-specific adjustments, are provided in section 13.3.

7.2.5. Case Studies

The case study methodology described in section 5.4 is implemented through evaluation of a Turbine Rear Structure (TRS) demonstration application provided by ParaPy. This application was selected for its comprehensive coverage of ParaPy capabilities, including geometric primitives, translation and

Dimension	Sub-metric	Tool	Scoring Rule	Weight
Semantic Correctness	Type Errors (N_{type})	`mypy`	Maximum: 5 points Deduction: -0.5 pts per error	40%
	Logical Errors (N_{logic})	`pylint`	Maximum: 5 points Deduction: -1.0 pts per error	
Maintainability	Maintainability Index (MI)	`radon`	Maximum: 100 points (MI)	35%
	Cyclomatic Complexity (CC)	`radon`	Maximum: 3 points Tiered Scoring: $\max(0, 3 - \lfloor CC/5 \rfloor)$	
PEP-8 Compliance	Style Violations per 100 LOC (v_{100})	`pycodestyle`	Maximum: 10 points	25%

LOC: Lines of Code

Table 7.3: Code Quality Scoring Framework

orientation operations, boolean operations, and meshing for FEM analysis, ensuring broad evaluation of the agent's accessible knowledge base (section 6.4).

Skeleton code for completion tasks was derived directly from the expert implementation by stripping implementation logic from major attributes and components while removing non-essential helper functions. This approach minimizes variance relative to the reference solution, enabling controlled comparison of agent-generated versus expert-developed implementations.

Scoping of the required effort for manual completion of the case study was established through consultation with ParaPy engineers:

- Expert ParaPy developer (daily SDK experience): approximately 4 hours
- Proficient ParaPy user (typical engineer): approximately 8 hours
- Programming-proficient but ParaPy-inexperienced developer: approximately 2 days

Based on these estimates, the baseline temporal effort is conservatively set at 8 hours.

The evaluation protocol applies the automated assessment pipeline (section 7.4) for syntax, runtime, and quality scoring, supplemented by manual functional analysis comparing generated implementations against the expert reference at both system and component levels. Expert review by ParaPy employees employs the rubric defined in Appendix C. For code quality benchmarking, the expert solution is assigned a reference score of 10 as the highest achievable standard. Complete case study results, including variant analysis, functional correctness assessment, and code quality comparisons, are presented in section 7.5. The demonstration application source code and generated outputs are available in the `case_study` directory on GitHub.

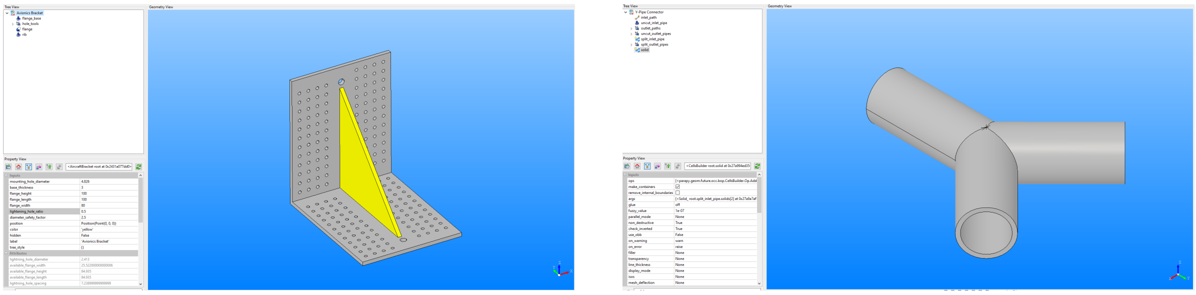
7.2.6. User Testing

The user testing protocol described in section 5.4 was implemented through controlled sessions with nine industry participants from GKN Aerospace Sweden and ParaPy B.V. This section details the participant composition, test case specifications, and data collection procedures employed during evaluation.

Participants

Nine participants took part in the user testing sessions, representing diverse professional backgrounds across aerospace engineering and software development. All participants are academically trained in technological fields but vary in Python programming experience, ParaPy SDK proficiency, and CAD software familiarity. The ParaPy employee subset includes developers involved in SDK platform development yet possessing limited KBE application development experience, ensuring expertise stratification within the sample. Prior to sessions, participants reviewed and signed consent forms granting permission to use collected data for research purposes and anonymous presentation. The consent form is available in section B.2.

Test Cases



(a) Expert reference solution for TC1 (Aircraft Mounting Bracket). **(b)** Expert reference solution for TC2 (Y-Pipe Connector).

Figure 7.1: Expert reference solutions for TC1 and TC2 rendered in the ParaPy GUI. These serve as functional correctness benchmarks for user testing evaluation.

Two mutually exclusive test cases of comparable difficulty were developed: an aircraft mounting bracket geometry and a Y-pipe connector geometry. Each test case is provided as skeleton code requiring implementation completion within 20-minute timed sessions. Participants were randomly assigned to Group A or Group B, completing one case manually and one with AI assistance in counterbalanced order to mitigate ordering effects. Complete test case specifications and expert solutions are available in the user_testing folder on GitHub. Interview scripts are provided in section B.3. Figure 7.1 shows the expert solutions of the two test cases rendered in the ParaPy GUI.

Test Case 1 (TC1): Aircraft Mounting Bracket

TC1 requires development of an L-shaped mounting bracket for aircraft avionics systems featuring mounting holes, lightening holes for weight reduction, and structural stiffener ribs. The implementation tests geometric translation and orientation, primitive selection, and advanced boolean operations while enforcing aerospace edge distance requirements (2.5D rule). Key skeleton code structure:

```
class AircraftBracket(GeomBase):
    mounting_hole_diameter = Input()
    base_thickness = Input()
    flange_height = Input()
    flange_length = Input()

    @Attribute
    def mounting_hole_positions(self) -> Sequence[Position]:
        """TODO: Apply 2.5D rule"""
        pass

    @Part
    def flange(self) -> ...:
        """TODO: L-shaped solid with holes"""
        pass
```

Test Case 2 (TC2): Y-Pipe Connector

TC2 requires development of a Y-shaped pipe connector for fluid systems with configurable branch angles and consistent wall thickness. While comparable in complexity to TC1, it emphasizes boolean operations over positioning tasks, requiring smooth junction transitions and proper pipe cutting operations. Key skeleton code structure:

```
class PipeConnector(GeomBase):
    inner_diameter = Input()
    wall_thickness = Input()
    branch_angle = Input()

    @Part
```

```
    def inlet_pipe(self) -> ThinWalledPipe:
        """TODO: Base inlet pipe"""
        pass

    @Part
    def solid_y_connector(self) -> ...:
        """TODO: Merged Y-connector with cut pipes"""
        pass
```

Data Collection and Functional Correctness Scoring

Session data comprises three categories of metrics. Temporal metrics capture completion time, execution time, and debugging duration. Productivity metrics quantify lines of code written and features implemented, where a feature is defined as any class subcomponent (method, attribute, or part) executing without error. Quality metrics include ParaPy-specific error counts, automated quality scores from the code quality framework, and functional correctness assessed through comparison with expert solutions.

Functional correctness evaluation compares participant implementations against expert reference solutions. Evaluation employs structured rubrics assessing both component-level and system-level functionality. Each rubric criterion receives a binary score: 1 point if the criterion is satisfied, 0 points if not. These binary scores are then multiplied by predefined weights and summed to produce a final correctness percentage.

For TC1 (Aircraft Mounting Bracket), the rubric evaluates 15 criteria across five subcomponents: base flange construction, rib implementation, mounting hole placement, lightening hole placement, and boolean operations. All hole-related criteria include compliance with the 2.5D aerospace edge distance rule. For TC2 (Y-Pipe Connector), the rubric evaluates 13 criteria across six subcomponents: outer diameter calculation, thin-walled pipe implementation, inlet pipe construction, two outlet pipe constructions, and boolean solid operations. Complete rubric specifications with criterion weights are provided in section B.4.

7.3. Unit Testing Results

Table 7.4 presents the test coverage scores and associated tests used to verify the base implementation of the framework. A total coverage score of 76% is achieved, with all tests passing in the current `version 0.4.0` of the framework. This is considered an acceptable result, particularly given that the core functionalities, defined as the agent core, model wrapper, semantic search engine, and tool framework, each achieve close to 100% test coverage. In contrast, secondary components such as the CLI application attain lower coverage.

Although the CLI is important for eventual user experience and adoption in both academic and industrial contexts, it does not reflect core architectural decisions or influence the design principles underlying the framework. As such, it is considered of lower academic relevance. Nonetheless, these components are still tested and covered to enhance confidence in the framework and to support long-term maintainability.

The `evaluator` package warrants special attention. This package implements the four evaluation dimensions employed throughout the chapter: syntax validation via `check_syntax()`, runtime verification via `run_code()`, functional assessment via the LLM Judge framework, and quality scoring via the automated code quality framework. While independent of the core framework functionality and not affecting runtime performance, verification through testing remains essential. These evaluators are used extensively in subsequent sections for automated evaluation (section 7.4), case study assessment (section 7.5), and user testing analysis (section 7.6). The full coverage reports can be accessed in the `coverage` directory on GitHub.

Component Coverage Breakdown

Overall Coverage: 76% | **Tests Run:** 716/716
Last Run: 2025-10-17 09:41 | **Build:** `0.4.0`

Component	Coverage		Tests
Agent Core		**100%**	(59 tests)
Data Types		**84%**	(NA*)
Model Wrapper		**100%**	(15 tests)
Prompt Engineering		**72%**	(NA*)
Semantic Search Engine		**99.4%**	(62 tests)
Tool Framework		**92%**	(253 tests)
CLI Application		**51%**	(192 tests)
Utility Package		**99%**	(73 tests)
Evaluator Package		**77%**	(62 tests)

* – These functionalities are tested within the dedicated sub-packages that use them, not separately.

Table 7.4: Test Coverage Dashboard - Framework Components

7.4. Automated Evaluation

Table 7.5: Summary statistics of performance metrics from automated evaluation across agent types and models.

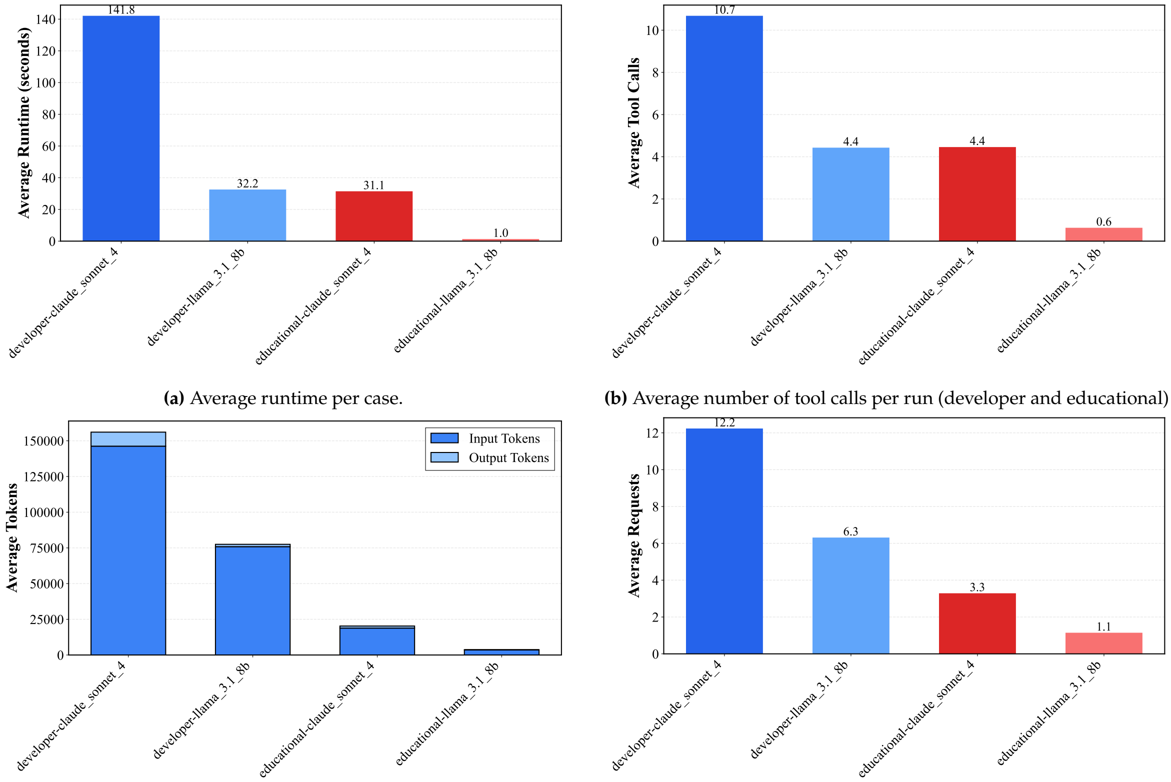
Agent Type	Model	Input Tokens	Output Tokens	Runtime (s)	Requests	Tool Calls
Developer	claude_sonnet_4	146234 ± 97068	9828 ± 8397	141.79 ± 93.02	12.2 ± 5.5	10.7 ± 4.6
Developer	llama_3.1_8b	75858 ± 166117	1643 ± 4205	32.23 ± 38.20	6.3 ± 3.0	4.4 ± 3.6
Educational	claude_sonnet_4	18835 ± 5874	1451 ± 326	31.12 ± 5.16	3.3 ± 0.7	4.4 ± 1.4
Educational	llama_3.1_8b	3569 ± 2723	109 ± 112	1.02 ± 0.32	1.1 ± 0.9	0.6 ± 0.5

This section presents the automated evaluation results for both agents using the test cases from Table 13.2 and Table 13.3. The analysis assesses whether requirements REQ-1-2, REQ-1-3, REQ-2-1 through REQ-2-3, and REQ-3-3 are satisfied within the scope of automated testing. Extended results are available in Appendix D.

Table 7.5 and Figure 7.2 summarize usage patterns across both agents and models. Claude Sonnet 4 consistently consumes more resources than LLaMa 3.1 8B, particularly in token usage (Figure 7.2c). Since language models process text through sequential operations, higher token counts directly increase computational requirements and runtime (Figure 7.2a). Tool usage differs between agents as well. The Developer Agent calls tools more frequently than the Educational Agent because it iteratively refines code based on validation feedback from `run_code` and `check_syntax` tools. The Educational Agent typically produces responses in a single pass without requiring code validation cycles.

7.4.1. Developer Agent

Table 7.6 presents performance metrics for the Developer Agent across the four evaluation dimensions: syntactic correctness, runtime correctness, functional quality (assessed by `LLMJudge`), and overall code



(a) Average runtime per case.

(b) Average number of tool calls per run (developer and educational)

(c) Average number of total tokens per run, split between input and output tokens.

(d) Average number of requests per run.

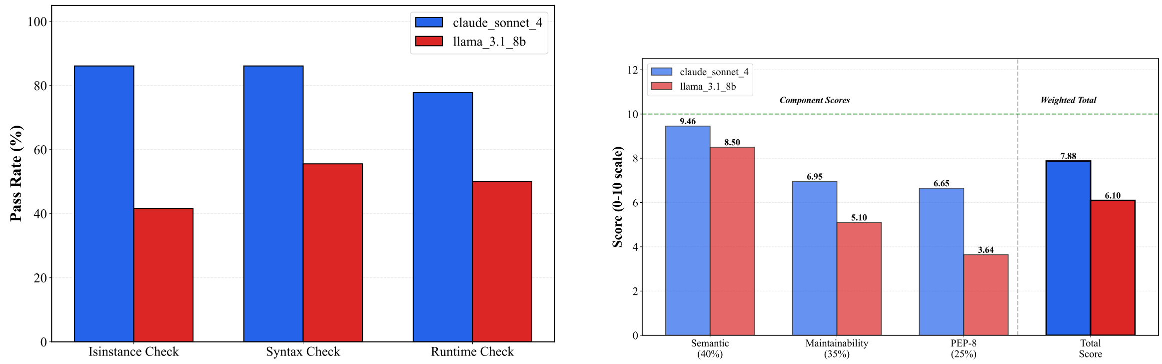
Figure 7.2: Usage metrics from the automated evaluations of both the Developer and Educational Agents, comparing the large *Claude Sonnet 4* model—representative of API-based usage—with the smaller *LLaMa 3.1 – 8B* model, representative of local execution environments.

quality (from the automated framework), and additionally structured output production. Claude Sonnet 4 substantially outperforms LLaMa 3.1 8B across all metrics. The smaller model struggled particularly with tool-calling schema adherence despite its 128K token context window, frequently limiting itself to basic functions like `run_code` and `check_syntax` or failing to call tools entirely. This restricted tool engagement prevented effective use of the knowledge framework and contributed to high failure rates, suggesting that smaller models lack the capacity to generalize across multiple structured tool schemas.

Table 7.6: Summary evaluation statistics for Developer Agent automated evaluation.

Model	isinstance %	Syntax %	Runtime %	Quality	Judge Avg	N
Claude Sonnet 4	86.1	86.1	77.8	7.88	4.13	36
LLaMa 3.1 - 8B	41.7	55.6	50.0	6.10	2.30	36

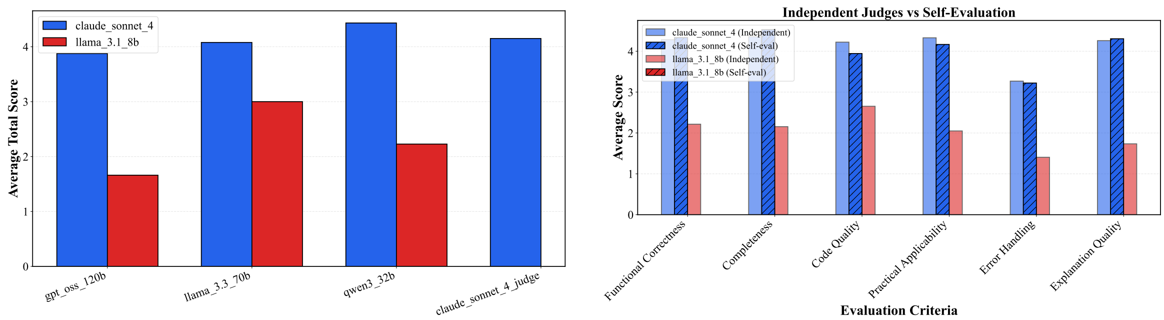
The quality analysis (Figure 7.3) reveals that while only Claude Sonnet 4 meets the overall quality threshold, both models demonstrate strong semantic correctness, with LLaMa consistently scoring above 7.5. Most quality deductions stem from maintainability and PEP-8 compliance rather than logical correctness, suggesting that external formatting tools could address these deficiencies in use cases prioritizing functional accuracy. The `LLMJudge` evaluation (Figure 7.4) confirms Claude's superiority across all dimensions while identifying error handling as a universal weakness. This finding has significant implications: while evaluation cases are stand-alone, production KBE applications require robust error handling in production. Self-assessment results (Figure 7.4b) show no strong bias, with Claude often rating its own outputs slightly lower than independent judges. The following section complements these quantitative metrics through visual inspection of generated ParaPy geometries,



(a) Instance check rates for Developer Agent cases.

(b) Quality scores and sub-scores.

Figure 7.3: Performance metrics for Developer Agent evaluation, showing pass rates and code quality breakdown into semantic, maintainability, and PEP-8 scores.



(a) LLMJudge scores by model.

(b) Self-assessment by Claude Sonnet 4.

Figure 7.4: LLMJudge evaluation scores across six dimensions: functional correctness, completeness, code quality, practical applicability, error handling, and explanation quality. No self-assessment was performed for LLaMa 3.1 8B due to insufficient reasoning capacity.

examining spatial reasoning capabilities critical for CAD applications that numerical scores may not fully capture.

Generated ParaPy Models

Visual inspection of geometric outputs reveals substantial performance differences between models. Claude Sonnet 4 demonstrates strong capability in skeleton code completion tasks (see Figure D.1 in appendix), correctly applying ParaPy principles including quantification constructs in cases *D-29* and *D-36*, and multi-step construction in *D-13* requiring multiple section profiles before `LoftedSolid` generation. Claude consistently identifies appropriate geometric primitives and sequences operations correctly, attributed to effective use of the semantic API search engine. Figure 7.5 shows additional complex geometries generated from scratch, including gear-like structures and organic forms in cases *D-5* and *D-10*. LLaMa 3.1 8B performs significantly worse, frequently failing to generate executable code or selecting inappropriate primitives despite occasionally applying ParaPy principles correctly. This stems from limited tool engagement, rarely invoking `suggest_apis` and restricting itself primarily to `run_code` and `check_syntax`, preventing effective knowledge framework utilization.

Both models exhibit significant limitations in spatial reasoning and 3D positioning tasks. The successful geometries primarily involve simple 1D translations or 2D layouts (e.g., case *D-19*), but performance degrades substantially when 3D positioning and orientation are required, with models frequently misplacing components or applying incorrect rotations. This reflects a fundamental architectural constraint and/or implementation failure: LLMs trained predominantly on text and code lack inherent geometric intuition despite producing syntactically correct geometric operations. Future work should explore geometry-aware feedback components (AI-based or traditional software) that enable iterative refinement based on spatial correctness alongside syntax and runtime validation, addressing this core limitation. The findings presented here are substantiated in section 7.5 through manual review of a

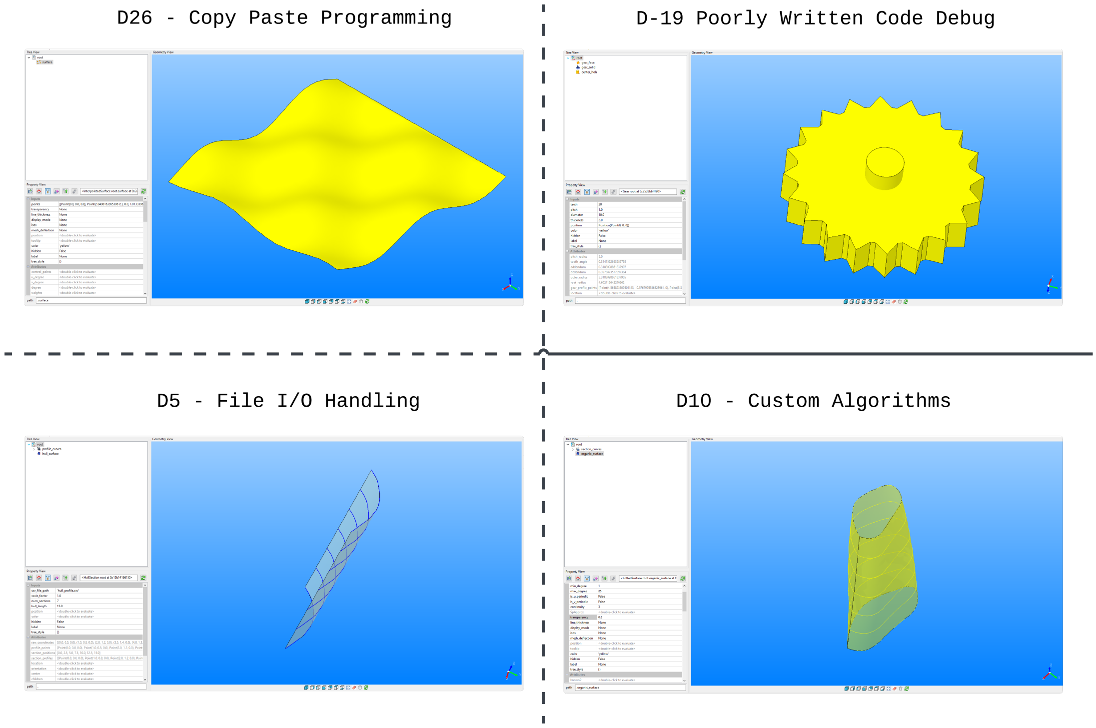


Figure 7.5: Complex geometries produced by Claude Sonnet 4 during Developer Agent evaluation, demonstrating advanced reasoning capabilities and effective ParaPy SDK usage.

turbine rear structure case study.

7.4.2. Educational Agent

Table 7.7 presents performance metrics for the Educational Agent evaluation cases. Claude Sonnet 4 achieved 100% successful output generation with consistent quality, while LLaMa 3.1 8B demonstrated limited effectiveness with only 52.2% successful runs. The quality score comparison requires careful interpretation: LLaMa's reported score of 7.30 derives from a single successful run, while Claude's 7.28 score reflects consistent performance across all 23 cases. The sub-score analysis (Figure 7.6) reveals that lower overall quality scores stem primarily from PEP-8 compliance rather than semantic or maintainability issues. This is expected for educational code snippets, which prioritize clarity and illustration over production formatting standards. Excluding PEP-8 considerations, Claude substantially exceeds the 7.5 quality threshold, demonstrating strong semantic correctness and maintainability in pedagogical contexts.

Table 7.7: Summary evaluation statistics for Educational Agent automated evaluation.

Model	isinstance [%]	Quality	Judge Avg	N
Claude Sonnet 4	100.0	7.28	4.33	23
LLaMa 3.1 - 8B	52.2	7.30*	2.06	23

* – The quality score of LLaMa 3.1 - 8B is based on **one** valid run only.

The `LLMJudge` evaluation (Figure 7.7) confirms Claude's superiority across all six pedagogical dimensions, with self-assessment scores (Figure 7.7b) closely aligned with independent judge evaluations, showing no systematic bias. The sub-score analysis identifies resource integration as Claude's relative weakness, suggesting incomplete utilization of the documentation framework and opportunities for improved

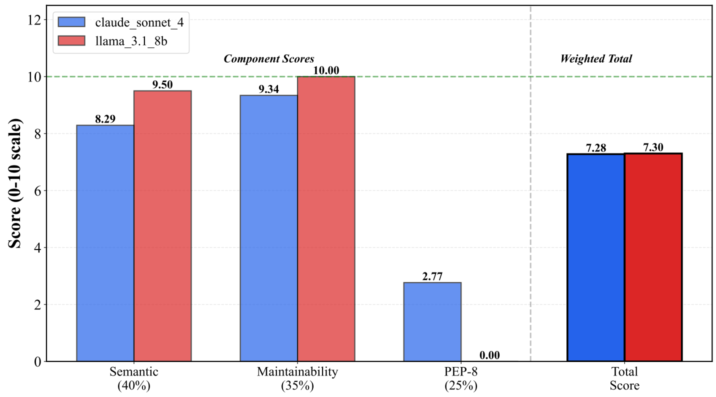
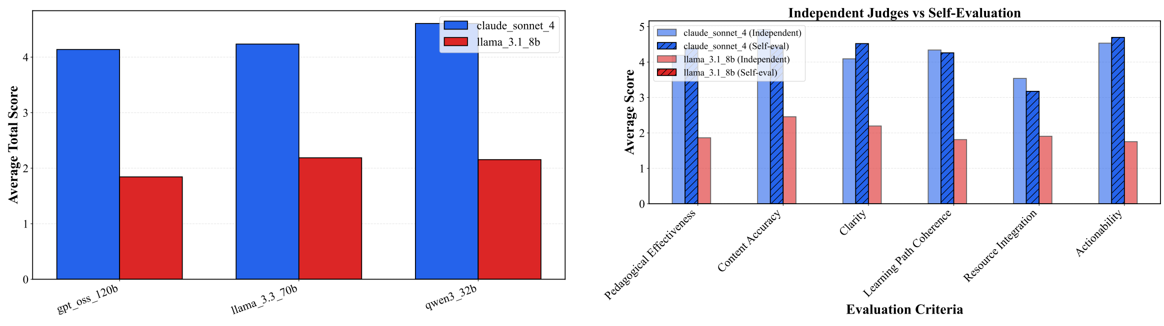


Figure 7.6: Automated quality scores and subscores for Educational Agent cases. LLaMa 3.1 8B statistics reflect only one successful run.



(a) LLMJudge scores by model.

(b) Self-assessment by Claude Sonnet 4.

Figure 7.7: LLMJudge evaluation scores across six pedagogical dimensions: pedagogical effectiveness, content accuracy, clarity, learning path coherence, resource integration, and actionability. No self-assessment was performed for LLaMa 3.1 8B due to insufficient reasoning capacity.

guidance toward relevant references. LLaMa demonstrates uniformly poor performance across all dimensions without particular strengths, reinforcing earlier observations about tool usage limitations. These Educational Agent results, combined with the Developer Agent assessment, complete the automated evaluation phase and establish the foundation for requirement compliance mapping in Table 7.8.

7.4.3. Summary

This section concludes the automated evaluation of both the Developer and Educational Agents and assesses whether the requirements targeted during these evaluations have been met, based on the results presented. The insights gained from this analysis will inform the subsequent discussion of the Developer Agent in the following section, and the broader validation of the Educational Agent and overall framework in section 7.6.

Important notes on the score computation in Table 7.8 are:

- To determine the skeleton code (REQ-1-1) and class definition (REQ-1-2) completion rates, evaluation cases `D13-D15` and `D27-D36` are classified as *skeleton code generation* tasks; all other cases are treated as *class definition generation* tasks.
- The passing rates for *syntax correctness* (REQ-2-1), *runtime correctness* (REQ-2-2) and *code quality* (REQ-3-3) are taken directly from the results in Table 7.6 and Table 7.7.
- *Functional correctness rates* (REQ-2-3) are computed based on the percentage of evaluation cases in which the average judge score met or exceeded the threshold of 3.0.

Table 7.8: Requirements verification results for Developer and Educational Agents based on automated evaluations.

Requirement	Pass metric	Developer Agent		Educational Agent	
		Claude Sonnet 4	*LLaMa 3.1 - 8B*	*Claude Sonnet 4*	*LLaMa 3.1 - 8B*
REQ-1-2	Minimum 85% successful completion rate for skeleton code completion	85%	23.1%	NA	NA
REQ-1-3	Minimum 75% successful completion rate for class definition generation	87.0%	53.2%	NA	NA
REQ-2-1	95% passing rate for syntax correctness of generated code	86.1%	55.6%	NA	NA
REQ-2-2	80% passing rate for runtime correctness of generated code	77.8%	50.0%	NA	NA
REQ-2-3	Intended functionality (LLM-Judge score > 3.0) in 70% of test scenarios	91.7%	8.3%	100%	34.8%
REQ-3-3	Minimum average expert quality score of 7.5 out of 10	7.88	6.10	7.28*	7.3**

* – When excluding PEP-8 score, the code quality exceeds the 7.5 threshold.
** – The quality score of LLaMa 3.1 - 8B is based on **one** run only.

7.5. Industry Case Study - TRS Application

This section presents the industry case study of a Turbine Rear Structure (TRS) demonstration application. It serves to consolidate the findings from section 7.4 and the requirements evaluated therein—namely REQ-1-2, REQ-1-3, REQ-2-1 through REQ-2-3, and REQ-3-3—in order to draw final conclusions on the framework's verification before proceeding to validation via end-user testing in section 7.6.

What follows is a detailed manual evaluation of the code and geometries generated by the Developer Agent. This includes checks for syntax and runtime correctness, code quality (evaluated via both the automated framework and expert review), and functional correctness—including geometric accuracy. The manual code review was conducted by an independent ParaPy employee, using the evaluation rubric defined in Appendix C.

7.5.1. Variant Analysis

Figure 7.8 presents geometries generated by the Developer Agent across four evaluation scenarios compared to the expert-developed reference. The *Minimal Prompt* case uses a single-sentence instruction, the *Skeleton Code Completion* case provides stripped expert implementation structure, and the *Extended Prompt* cases (I and II) offer progressively detailed natural language specifications of required subcomponents and geometric properties. All code and geometry outputs are available in the `case_study` directory on GitHub, with corresponding quality scores and runtimes summarized in Table 7.9.

Minimal Prompt

```
>>> "Create a parametric model of a turbine rear structure (TRS)."
```

The minimal prompt demonstrates poor functional performance despite achieving relatively high automated code quality scores, highlighting the distinction between syntactic correctness and engineering validity. While the agent correctly identifies TRS components (casings, flanges, blades), the generated geometry lacks functional accuracy: casings are solid rather than hollow (preventing airflow), and blade geometry resembles simplified toy structures rather than engineering-grade components. The 134-second generation time represents substantial acceleration over the 8-hour expert baseline, revealing a use case differentiation: experienced ParaPy users can leverage the framework for rapid base architecture generation followed by manual refinement, while novice users may struggle with the cognitive load of identifying and correcting functional deficiencies without domain expertise to guide debugging efforts.

Skeleton Code

```
>>> "Complete the following skeleton code of a turbine rear structure (TRS) parametric model:

<skeleton code>"
```

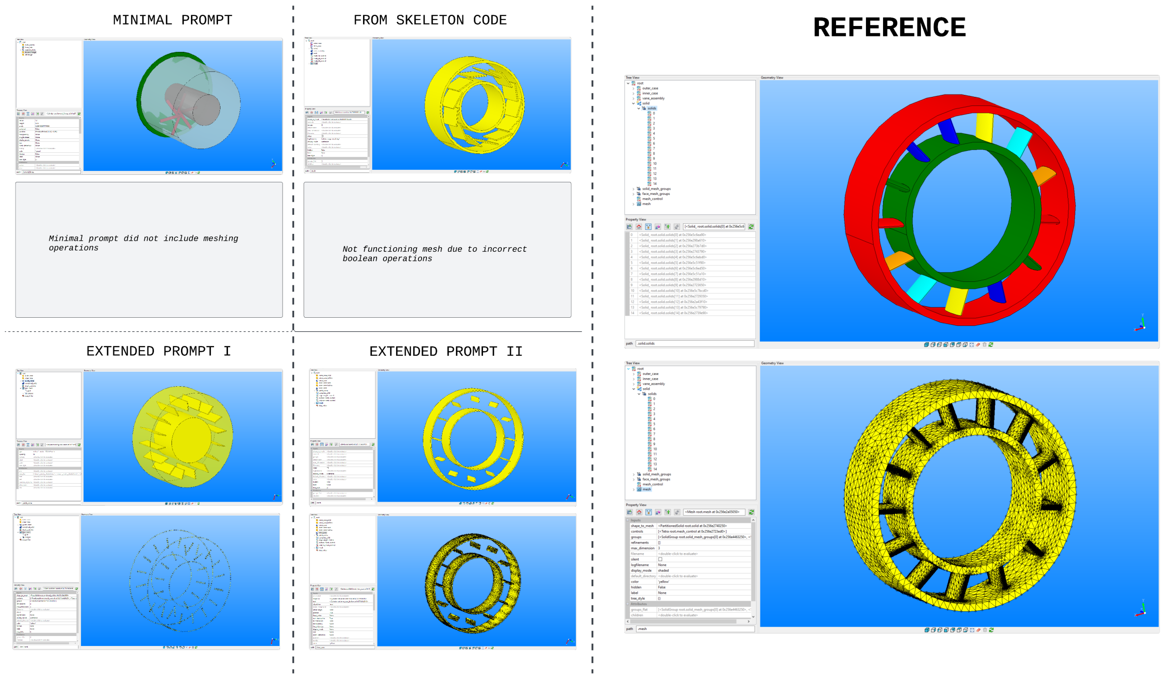


Figure 7.8: Comparison of expert-generated application and AI-generated solutions using the Developer Agent. Minimal prompting shows limited performance while extended prompts demonstrate significant improvements, with spatial orientation remaining the primary challenge. Skeleton code completion underperforms compared to natural language prompts.

See section 13.8 for the skeleton code.

The skeleton code completion case demonstrates counterintuitive results, appearing visually superior to the minimal prompt but exhibiting lower functional performance than extended natural language prompts. Despite correctly modeling hollow casings (unlike *Extended Prompt I*), the implementation failed to generate valid solid geometry or successful mesh outputs, indicating parametric pipeline breakdown. Expert review confirmed the lowest code quality among AI variants, primarily due to reduced semantic correctness scores. The functional deficiencies stem from two factors: recurring error misclassification where the Developer Agent interprets ParaPy GUI runtime errors as known SDK limitations rather than output faults, preventing effective iterative refinement; and model overconstraint, where partially completed code structures limit generative flexibility compared to open-ended natural language prompts.

An additional experiment using skeleton code derived from *Extended Prompt II* output (agent completing its own previously generated structure) showed marginal geometry improvements but continued solid generation and meshing failures, confirming that performance degradation results from external skeletal constraints rather than architectural discrepancies between expert and AI solutions. This finding suggests that skeleton-based workflows, while conceptually aligned with the MBSE translation engine integration goal (Chapter 2), may inadvertently constrain model reasoning capacity and reduce output quality.

Extended Prompt

```
>>> "I need to develop a parametric model of a turbine rear structure (TRS).
It consists of a ring shaped outer case and inner case, with simple guide vanes inbetween.
I then need to make one solid from the sub-solids (inner case, outer case, and guide vanes)
and mesh it for use in FEM solvers. You should make sure to develop all necessary components
for successfully meshing in ParaPy. Finally, the solid should be written to a .step file."
```

The extended prompt cases demonstrate superior performance across all evaluated dimensions. *Extended Prompt I* achieved higher maintainability index scores despite slightly lower overall quality due to increased PEP-8 violations and cyclomatic complexity from expanded codebase size. The lack of explicit hollow casing specifications led to solid body generation, underscoring the model's limitation in inferring unstated engineering requirements. However, successful solid generation and mesh control application marked substantial improvement over previous cases.

```
>>> "I need to develop a parametric model of a turbine rear structure (TRS).
It consists of a hollow, cylinder shaped outer case and inner case, with ovally-shaped guide
    vanes inbetween.
I then need to make one solid from the sub-solids (inner case, outer case, and guide vanes)
and mesh it for use in FEM solvers. You should make sure to develop all necessary components
for successfully meshing in ParaPy. Finally, the solid should be written to a .step file."
```

Extended Prompt II produced the highest-quality output, closely approximating expert solutions while requiring only 2% of manual development time (640 seconds versus 8 hours). Detailed geometric specifications enabled correct hollow casing construction, oval vane extrusion, valid solid definition, and successful meshing, achieving optimal results in both automated quality assessment and expert evaluation. The persistent limitation remains spatial reasoning: vane orientation errors occurred across all cases despite prompt improvements, reflecting the fundamental LLM constraint in 3D coordinate system understanding discussed in section 7.4. This deficiency persists without explicit spatial instructions and represents a broader architectural limitation when applying language models to geometrically complex engineering tasks, as further demonstrated in section 7.6.

Table 7.9: Code quality comparison between expert-developed and AI-generated code for the TRS case study using the Developer Agent.

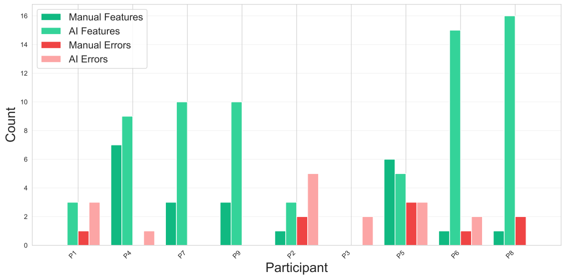
Variant	**Quality Score**		**Semantic Score**		**Maintainability Score**		**PEP-8 Score**		**Effort**		*Remarks*
	Auto	*Expert*	*Auto*	*Expert*	*Auto*	*Expert*	*Auto*	*Expert*	*Auto*	*Expert*	
Baseline	8.84	10	10.0	10	8.13	10	8	10	NA	8 h	*MI: 73.26* *CC: 1.23* *PEP-8: 1*
Minimal Prompt	8.56	NA	10.0	NA	7.32	NA	8.0	NA	134s	8 h	*MI: 61.66* *CC: 1.08* *PEP-8: 3*
From Skeleton Code	8.09	7.3	10.0	6.0	7.41	9.0	6.0	7.0	505s	8 h	*MI: 63.05* *CC: 1.13* *PEP-8: 5*
Extended Prompt - I	8.12	*NA*	10.0	*NA*	7.48	*NA*	6.0	NA	335s	8 h	*MI: 63.98* *CC: 1.42* *PEP-8: 5*
Extended Prompt - II	8.62	7.7	10.0	7.0	7.48	9.0	8.0	7.0	640s	8 h	*MI: 64.06* *CC: 1.14* *PEP-8: 5*

7.5.2. Summary

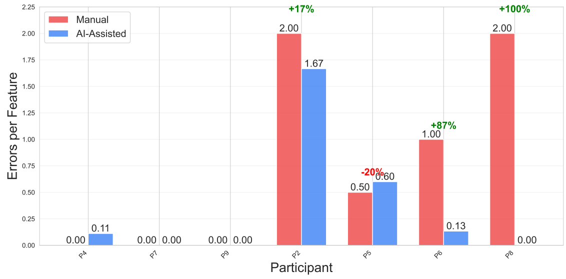
This section investigated Developer Agent performance across four industry-derived test scenarios, with all generated code achieving syntactic and runtime correctness while successfully defining parametric class structures from both skeleton code and natural language inputs. The 100% pass rate across all TRS variants, including complex tasks, reinforces findings for REQ-1-2 and REQ-1-3 while justifying reassessment of REQ-2-1 and REQ-2-2 from the near-threshold automated evaluation results. All outputs exhibited sufficient code quality, supporting REQ-3-3 compliance. However, persistent spatial reasoning limitations necessitate marking REQ-1-2, REQ-1-3, and REQ-3-3 as "partially fulfilled" in Table 7.11, while REQ-2-1 and REQ-2-2 achieve full compliance.

The framework substantially exceeded time reduction objectives (REQ-3-1), generating code in 2% of manual development time (Table 7.9), marking this requirement as fulfilled. Functional equivalence (REQ-3-2) receives partial fulfillment: while *Extended Prompt II* closely approximates expert solutions, persistent vane orientation errors limit immediate production deployability. Two critical limitations emerged: the Developer Agent's misclassification of ParaPy GUI runtime errors as SDK limitations rather than output faults, disrupting iterative refinement; and fundamental spatial reasoning deficiencies requiring future integration of geometric validation modules, as discussed in Chapter 9.

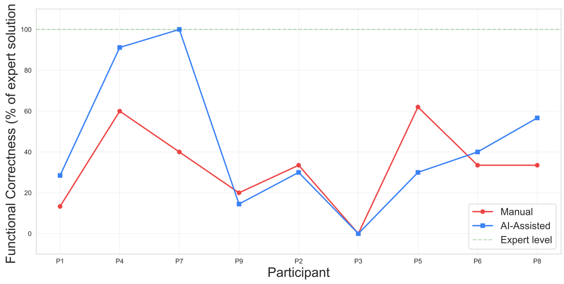
This concludes the verification phase covering requirements through REQ-3-3, providing sufficient



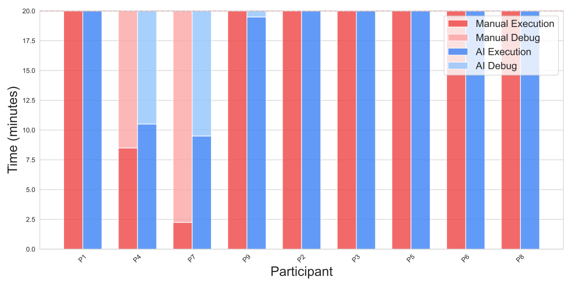
(a) Features completed vs errors (absolute counts)



(b) ParaPy error rate: errors normalized by features completed, positive values denote a *decrease* of normalized errors in the AI-assisted case w.r.t the manual case.



(c) Functional correctness comparison (normalized to expert solution)



(d) Time breakdown: manual vs. AI-assisted

Figure 7.9: High-level results from the user testing sessions involving KBE application development using the ParaPy SDK. Metrics include development speed, ParaPy SDK-specific error rates, functional correctness and equivalence to the expert solution, as well as a breakdown of session time per participant.

confidence to progress to validation. The next phase evaluates whether high-level research goals from Chapter 2 are achieved through user testing, addressing REQ-3-4, REQ-4-1, REQ-4-2, and REQ-4-3.

7.6. User Testing

Figure 7.9 presents the high-level outcomes of the user testing sessions, broken down by participant. A key observation from Figure 7.9a is that, in nearly all cases, AI-assisted sessions resulted in a higher number of completed features compared to manual sessions. It should be noted, however, that feature completion here refers solely to syntactically and runtime-valid outputs—it does not guarantee functional correctness. While it may initially appear that this increase in completed features also leads to more ParaPy-specific errors (e.g., Part violations), Figure 7.9b reveals otherwise: when normalized against the number of implemented features, the use of the agentic framework actually reduces ParaPy-related coding errors, particularly for novice users (a point expanded upon later).

Figure 7.9d shows the breakdown of session time across execution and debugging phases. It reveals that many participants failed to achieve a successful run within the allotted time. This justifies the use of normalized temporal metrics—such as features per minute or lines of code per minute—rather than absolute time, for evaluating framework performance.

Figure 7.9c compares the functional correctness of solutions between manual and AI-assisted sessions. The results vary: in several cases, AI-assisted solutions outperform their manual counterparts due to the higher number of implemented features. However, in other instances, functional correctness is lower in the AI-assisted case. This is largely due to geometrical limitations—especially in positioning and orientation—already identified in previous evaluations. These issues often resulted in functionally incorrect geometry, even when more features were technically implemented. In contrast, manually developed solutions may have included fewer features, but those features were generally geometrically sound.

One particularly noteworthy case (P7) achieved 100% functional correctness in the AI-assisted session. It was the only instance in which a fully correct solution was produced, matching the expert-developed reference. This outcome was achieved through human-in-the-loop iterative development, where generated code was incrementally improved by feeding it back into the Developer Agent with detailed clarifications of the errors encountered—particularly those related to orientation and positioning. This process enabled the participant to overcome the model's spatial limitations and arrive at a correct solution within 20 minutes—a task that would typically take experts approximately 2 hours. This result lends strong support to earlier suggestions that incorporating geometric feedback mechanisms into the framework could drastically improve its ability to generate functionally correct geometry, a point further explored in Chapter 9.

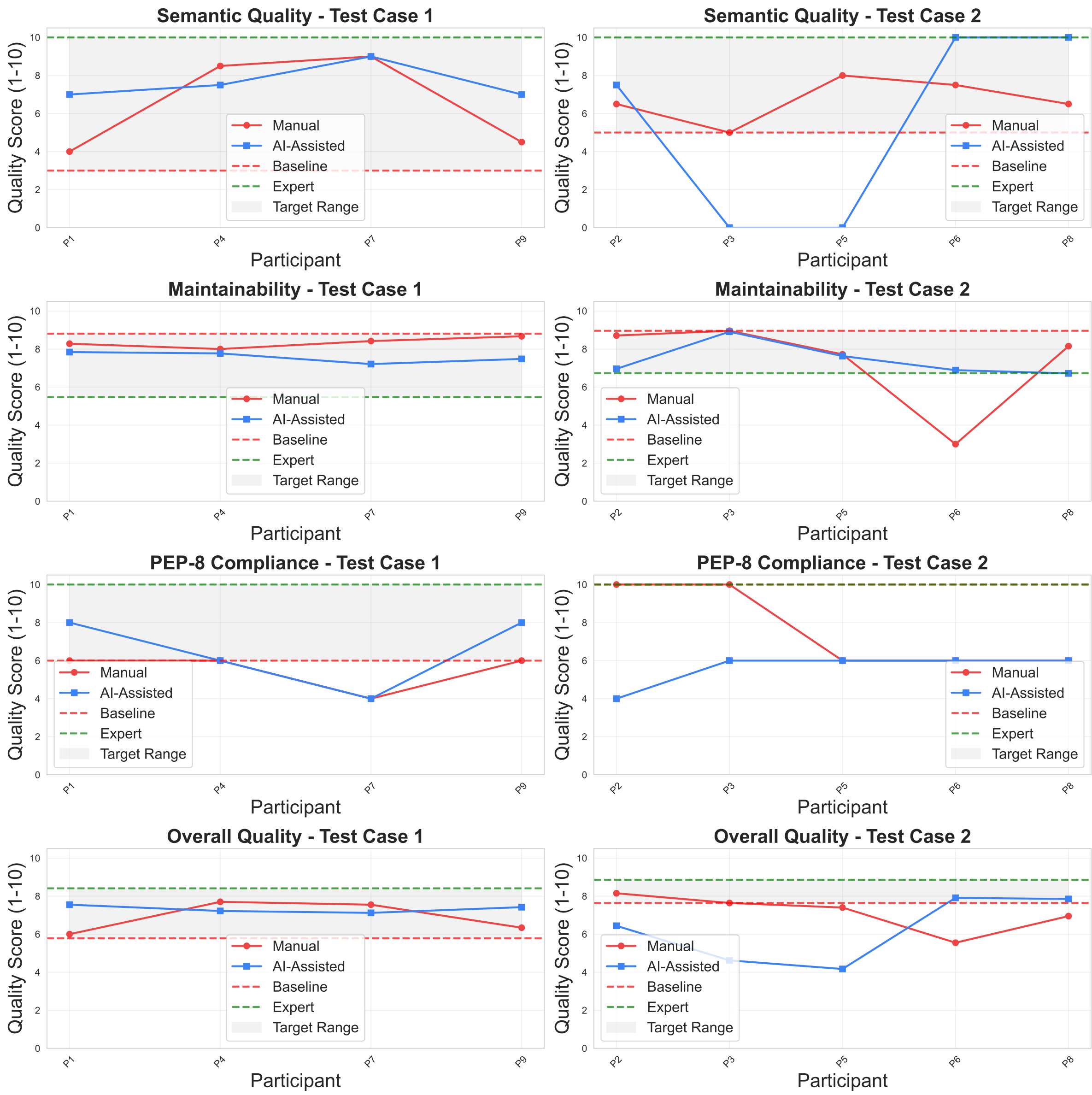


Figure 7.10: Comparison of manual and AI-assisted cases per participant, with respect to code quality as evaluated by the automated code quality framework. Scores are subdivided into the three dimensions: semantic correctness, maintainability, and PEP-8 compliance. The expert-level reference represents the score of the expert-developed solution, while the baseline corresponds to the score of the provided skeleton code. Results are separated by TC1 and TC2 to ensure accurate comparison against their respective reference solutions.

Aside from functional correctness, Figure 7.10 compares the manual and AI-assisted solutions in terms of code quality. Test cases are separated into TC1 and TC2 to enable valid comparison against their respective expert and baseline reference scores. For TC1, code quality varied across participants, with no clear trend favoring either manual or AI-assisted approaches—suggesting comparable performance. However, in TC2, a larger proportion of AI-assisted cases received lower quality scores than their manual counterparts.

Looking at the sub-scores for each case, TC1 again shows minimal variation between the two approaches, with participants often achieving equal or similarly distributed scores. In TC2, the most significant contributor to lower AI-assisted scores was semantic correctness, followed by PEP-8 compliance. While PEP-8 violations are generally considered less critical to core framework performance, the decrease in semantic quality is more concerning, as it indicates fundamental issues in code logic or structure.

This drop in semantic quality can be partially attributed to a recurring observation made throughout this thesis: the model's difficulty with geometric reasoning. However, a second contributing factor emerged during user testing—namely, the misuse of the Educational Agent. Several participants used the Educational Agent to generate complete, functional code implementations rather than relying on the Developer Agent. As the Educational Agent lacks the full toolset and knowledge framework of the Developer Agent, it often produced incorrect or incomplete ParaPy code, which negatively impacted the final code quality in TC2.

Despite these limitations, the results do not show conclusive evidence that the agentic framework consistently improves code quality. TC1, where misuse of the Educational Agent was minimal, still presents mixed outcomes. While the sample size is too small for statistically robust conclusions, the current data suggests that the agentic framework does not have a consistent positive or negative impact on code quality produced by end-users.

Taken together, these findings indicate that the primary benefit of the agentic framework lies in reducing development time (as seen in the number of features completed) and alleviating the knowledge burden for users. This is supported both by the higher number of implemented features in the AI-assisted cases and the lower rate of ParaPy-specific errors per feature. The next subsection will quantify these effects further and evaluate whether the tested requirements have been met based on the results from user testing.

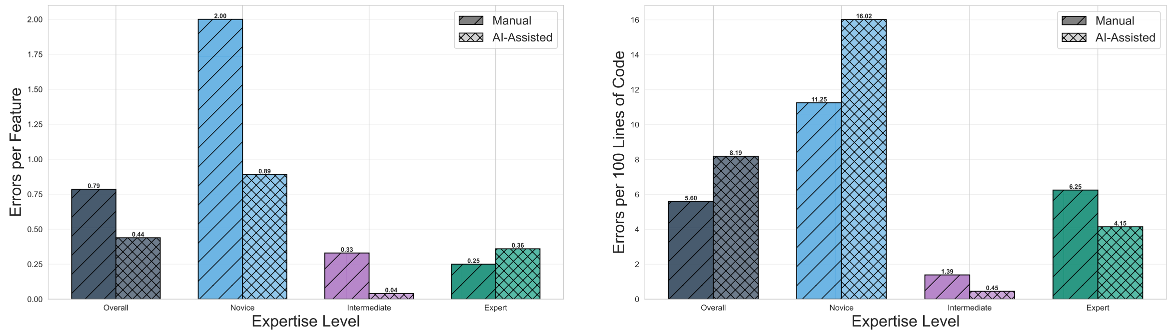
7.6.1. Requirement Compliance

The observations regarding increased productivity, absence of consistent improvements in code quality or functionality, and reductions in error rates are visualized in Figure 7.11 and Figure 7.12, respectively. Figure 7.11a and Figure 7.11b show that, depending on the chosen metric, the agentic framework can both increase and decrease the error rate.

With respect to errors per feature implemented (indicative for testing REQ-4-3), the framework leads to a marked reduction in domain-specific errors—particularly for novice and intermediate users—indicating improved robustness per unit of functional output. However, for expert users, the opposite effect is observed, with a slight increase in errors per feature when using the AI-assisted approach. In contrast, when measuring errors per 100 lines of code, the trend is less pronounced. For novice users, the framework slightly increases the error rate, while for intermediate and expert users, the error rate marginally decreases. This discrepancy between metrics highlights a limitation of LOC-based evaluation: the presence of non-functional or redundant ("dead") code can dilute the error rate without improving overall correctness or quality.

As such, error rate per feature is deemed a more rigorous and meaningful indicator of knowledge burden and logical correctness. The observed reductions for novice and intermediate users suggest that the framework successfully alleviates domain-specific knowledge requirements. For expert users, however, a slight regression is observed. This can likely be attributed to two factors: time pressure during testing, which discourages in-depth code review, and the misuse of the Educational Agent for code generation—resulting in blind copy-pasting rather than critical evaluation of generated output. Since expert users typically catch errors during manual development, this reduced review effort may lead to increased oversight.

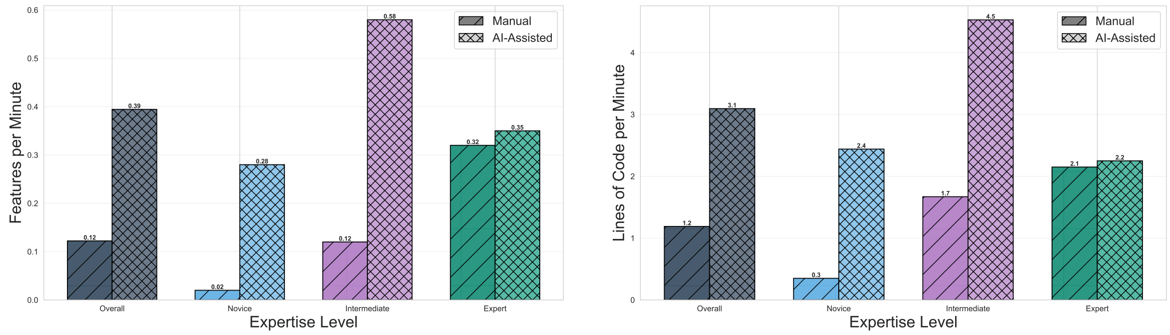
This pattern—improved productivity paired with decreased review effort—may also explain why code quality and functional correctness do not consistently improve in the AI-assisted cases. In conclusion, the agentic framework appears to reduce knowledge burden for novice and intermediate users, while providing marginal or slightly negative effects for expert users in terms of error rate. These insights will inform the final conclusions and requirement assessments presented in subsection 7.6.2.



(a) ParaPy-specific error rate, measured as the number of errors per feature implemented, segmented by user experience level.

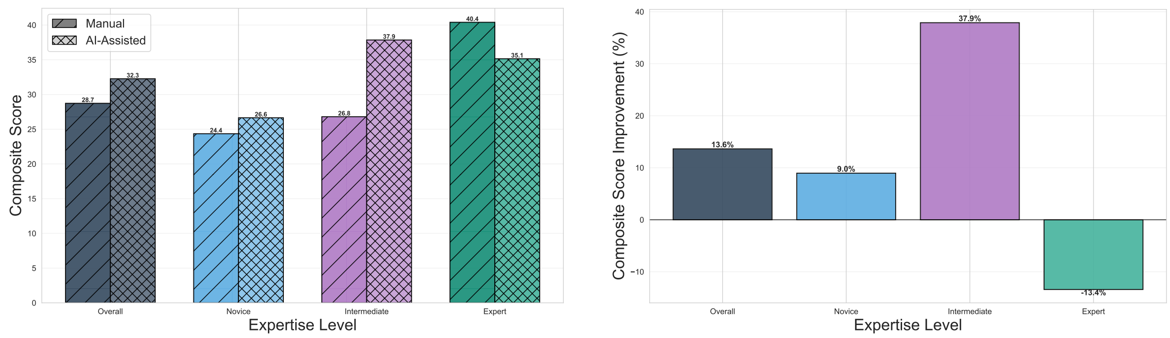
(b) ParaPy-specific error rate, measured as errors per 100 lines of code (LOC), segmented by user experience level.

Figure 7.11: Error rate metrics obtained from user testing, comparing manual and AI-assisted cases across different user experience levels. The error rate serves as an indicative measure for assessing whether the agentic framework effectively reduces the knowledge burden on end-users.



(a) Productivity: features completed per minute (at 20 min)

(b) Productivity: LOC written per minute (at 20 min.)



(c) Composite score: 40% LOC/min + 35% functional correctness + 25% code quality.

(d) Composite score improvements: AI vs manual

Figure 7.12: Time metrics and composite improvement scores from user testing, comparing manual and AI-assisted cases across different experience levels. Productivity is measured in terms of lines of code or features implemented per minute. The composite score offers a holistic assessment of framework performance by integrating productivity, functional correctness, and code quality.

Finally, Figure 7.12 quantifies and consolidates the observations made throughout this section by illustrating the measured improvements in productivity—addressing REQ-4-1—as well as the overall composite score defined by REQ-3-4, including the required improvement threshold specified by REQ4-2.

As shown in Figure 7.12a and Figure 7.12b, the Developer Agent leads to increased development speed across all experience levels—measured in both features completed per minute and lines of code written per minute. As expected, the most substantial productivity gains are observed among novice and intermediate users, for whom the framework alleviates more of the knowledge burden.

More revealing is the composite score displayed in Figure 7.12c, which represents a weighted average of three key performance indicators defined by REQ-3-4: code output rate (LOC/min), functional correctness, and code quality. Here, an interesting phenomenon emerges: intermediate users exhibit the largest increase in composite performance, surpassing even novice users. In contrast, expert users experience a decline in composite score despite increased raw development speed.

This pattern aligns with earlier observations concerning code quality, error rate, and functional correctness. While expert users benefit from faster generation of code, the meaningfulness of that output—defined by its correctness and maintainability—declines. This is likely due to reduced time for manual review and the potential misuse of the Educational Agent, leading to less critically evaluated outputs. The particularly strong performance of intermediate users can be explained by their optimal balance of experience. Unlike novices, they are not overwhelmed by ParaPy's complexity and have enough domain knowledge to validate and adapt the generated code. At the same time, they are not yet as proficient as experts, making them more reliant on the copilot, and thus more likely to benefit from its assistance. Their ability to both leverage and supervise the agent's output likely contributes to the highest observed gains in composite performance.

7.6.2. Summary

Table 7.10: Requirements validation results from user testing across expertise levels.

Requirement	Pass metric	Performance			Remark
		Novice	*Intermediate*	*Expert*	
REQ-3-4	The composite performance score shall be calculated as a weighted average of normalized development time improvement (40%), functional correctness rate (35%), and expert quality score (25%)	Yes	Yes	Yes	*By implementation*
REQ-4-1	The co-pilot tool shall achieve minimum development time reductions of 40% for novice users and 25% for expert users	-1300%	-383%	+9%	*Based on features/minute*
REQ-4-2	The co-pilot tool shall achieve composite score improvements of minimum 30% for novice users and 15% for expert users	+9.0%	+37.9%*	-13.4%	
REQ-4-3	Novice users utilizing the co-pilot tool shall achieve task completion rates equivalent to 80% of expert baseline performance** and demonstrate 50% reduction in domain-specific knowledge errors	87.5% -55.5%	181.3% -87%	100% +44%	*Task completion*** *Error reduction (errors/feature)*

* – Indeterminate as it is higher than 30% but not defined in the MSC.
** – Based on features/minute relative to expert features per minute for manual case

The observations made in the preceding section provide the basis for a final judgment on the requirements tested through user testing, alongside a discussion of encountered limitations. This assessment, together with the evaluations presented in subsection 7.4.3 and subsection 7.5.2, culminates in the final verification and validation judgment for the agentic framework. These findings are consolidated in the form of a Requirements Traceability Matrix (RTM) in Table 7.11. The RTM, alongside the key observations, limitations, and conclusions presented throughout this chapter, serves as the foundation for the overall conclusions in Chapter 8 and informs the future research directions outlined in Chapter 9. Finally, Table 7.10 summarizes the requirements assessed through user testing and determines whether they have been met, based on the measurable success criteria introduced in section 3.5 and the metrics discussed above.

7.7. Summary & Conclusion

This chapter presented the verification and validation of the developed agentic framework for KBE application development using the ParaPy SDK. Through a multi-faceted evaluation strategy encompassing unit testing, automated evaluation, manual case studies, and end-user validation, the framework was assessed against requirements established in Chapter 2. The Requirements Traceability Matrix in Table 7.11 consolidates these findings, demonstrating that 8 of 13 measurable success criteria were fully met and 5 partially met.

The verification phase established foundational confidence through 76% overall code coverage with 100% test success rate, with critical subsystems achieving near-complete coverage. Automated evaluations revealed substantial performance differences between model configurations: Claude Sonnet 4 demonstrated superior performance with 86.1% syntax correctness, 77.8% runtime correctness, and 7.88/10 code quality scores, while LLaMa 3.1 8B struggled with structured output generation and tool utilization, achieving only 41.7% successful instantiation. The TRS industry case study confirmed that all code variants achieved syntactic and runtime correctness (REQ-2-1, REQ-2-2), with *Extended Prompt II* producing geometry closely approximating expert solutions in 2% of baseline development time. However, persistent spatial reasoning limitations prevented full functional equivalence (REQ-3-2), highlighting fundamental constraints of LLM-based approaches for geometric tasks.

User testing with nine industry participants validated the framework's effectiveness in reducing development time (REQ-4-1) and knowledge barriers (REQ-4-3), with novice and intermediate users achieving 1300% and 383% improvements in features per minute respectively, and domain-specific error rates decreasing by 55.5% and 87%. However, the framework's impact on code quality and functional correctness proved nuanced: intermediate users experienced the largest composite score improvement (+37.9%), while expert users showed slight decline (-13.4%), attributed to reduced code review effort and occasional Educational Agent misuse. This pattern indicates that the framework's primary value lies in accelerating development velocity and reducing knowledge burden rather than consistently improving output quality, suggesting deployment as a development aid requiring expert review rather than autonomous code generation.

The differential effectiveness across expertise levels reveals an "optimal assistance zone" where intermediate users, possessing sufficient domain knowledge to validate outputs while still facing knowledge gaps, derive maximum benefit. The composite score requirements (REQ-3-4, REQ-4-2) were met for intermediate users but not for expert users, confirming that framework value depends critically on user expertise level. These findings, combined with persistent spatial reasoning limitations and occasional error misclassification by the Developer Agent, emphasize the need for geometric validation modules and organizational safeguards to maintain quality standards, as discussed in Chapter 9. The verification and validation results provide the foundation for the concluding assessment in Chapter 8, where overall research contribution and gap closure are evaluated, and inform the technical improvements and deployment strategies proposed in Chapter 9.

Table 7.11: Measurable Success Criteria Compliance Matrix

Req. ID	Requirement	Compliance	Method	Justification	Source
MSC-1-1	The co-pilot tool shall support local LLM deployment via the Ollama service and web-based access through API services offered by approved providers.	Yes	By implementation	Verified through system architecture implementation and deployment testing.	Chapter 6, section 7.3
MSC-1-2	The co-pilot tool shall achieve a minimum 85% successful completion rate for skeleton code structures, measured against a standardized test suite of ParaPy code skeletons with varying complexity levels.	Partial	Testing: verification and validation	Initial verification through automated evaluations with conclusive compliance from manual evaluations. Validation through user testing.	section 7.4, 7.5
MSC-1-3	The co-pilot tool shall achieve a minimum 75% successful completion rate for generating class definitions from natural language specifications, measured against a standardized test suite of ParaPy development cases with varying complexity levels.	Partial	Testing: verification and validation	Initial verification through automated evaluations with conclusive compliance from manual evaluations. Validation through user testing.	section 7.4, 7.5
MSC-2-1	Generated code shall achieve 95% syntactic correctness as measured by automated parsing and compilation checks.	Yes	Testing: verification	Verification through automated and manual evaluations, compliance through manual evaluation.	section 7.4, 7.5
MSC-2-2	Generated code shall execute without runtime errors in 80% of test cases, measured through automated execution testing within controlled ParaPy environments using standardized input parameters.	Yes	Testing: verification	Verification through automated and manual evaluations, compliance through manual evaluation.	section 7.4, 7.5
MSC-2-3	Generated code shall achieve intended functionality in 70% of test scenarios, validated through automated testing suites that compare output behaviour against reference implementations and expert-defined acceptance criteria.	Partial	Testing: verification and validation	Initial verification through automated evaluations with conclusive compliance from manual evaluations. Further validation through user testing.	section 7.4, 7.5 , 7.6
MSC-3-1	The primary metric to assess performance improvements introduced by the co-pilot tool shall be development time reduction through controlled user studies using standardized KBE development scenarios, with measurements captured for task completion time and development velocity metrics (e.g., lines per minute, features per minute).	Yes	By implementation	Implemented through industry case study analysis and end-user testing.	section 7.5, 7.6
MSC-3-2	AI-generated code shall achieve functional equivalence to manually completed reference solutions in 85% of test cases, measured through comprehensive output comparison and behaviour verification testing protocols.	Partial	Testing: verification and validation	Initial verification through manual evaluation of case study. Validation through user testing.	section 7.5, 7.6
MSC-3-3	Generated code shall receive an average expert quality score of 7.5 out of 10, using a standardized rubric covering semantic correctness (40%), maintainability (35%), and PEP-8 compliance (25%), evaluated by qualified ParaPy domain experts.	Yes	By implementation and subsequent verification	Implemented through automated code quality framework and/or structured expert evaluation protocol with standardized rubric and scoring methodology. Verified through automated evaluations with compliance from manual evaluations. Further validated through end-user testing.	subsection 7.2.4, section 7.5, 7.6
MSC-3-4	The composite performance score shall be calculated as a weighted average of normalized development time improvement (40%), functional correctness rate (35%), and expert quality score (25%), with the co-pilot tool achieving a minimum composite score of 75%.	Yes	By implementation	Implemented through automated scoring calculation system combining metrics from MSC-3-1, MSC-3-2, and MSC-3-3.	section 7.6
MSC-4-1	The co-pilot tool shall achieve minimum development time reductions of 40% for novice users and 25% for expert users, measured through controlled comparative studies using similar but mutually exclusive KBE development tasks and the primary metrics introduced in MSC-3-1.	Yes	Testing: validation	Validated through user testing and case study.	section 7.6
MSC-4-2	The co-pilot tool shall achieve composite score improvements of minimum 30% for novice users and 15% for expert users compared to their baseline manual development performance.	Partial	Testing: validation	Validated through user testing.	section 7.6
MSC-4-3	Novice users utilizing the co-pilot tool shall achieve task completion rates equivalent to 80% of expert baseline performance and demonstrate 50% reduction in domain-specific knowledge errors compared to unassisted novice performance.	Yes	Testing: validation	Validated through user testing.	section 7.6

8 Conclusion

This research addressed the persistent challenge of time- and knowledge-intensive Knowledge-Based Engineering application development by developing and evaluating a generative AI-powered coding assistant specifically tailored for the ParaPy SDK. Through systematic investigation spanning literature analysis, industrial case studies, framework development, and comprehensive evaluation, this work demonstrates that large language models can effectively support domain-specific code generation when appropriately augmented with specialized knowledge infrastructure and progressive verification mechanisms.

The research was motivated by converging challenges in modern engineering automation. Front-loaded engineering approaches, essential to high-tech industries, require extensive automation through KBE applications. However, developing these applications remains prohibitively time-consuming and dependent on specialized expertise that combines domain engineering knowledge with proficiency in proprietary programming frameworks. While recent advances in generative AI have transformed general software development, commercial language models lack the specialized knowledge necessary for domain-specific frameworks, leading to extensive hallucinations and non-functional code generation. The MBSE-driven DEFAINE methodology successfully automated knowledge modeling and round-trip engineering but left manual code completion as a critical bottleneck, representing both the most time-intensive development phase and the most dependent on domain-specific expertise.

Industrial validation at GKN Aerospace Sweden substantiated these challenges through interviews with domain experts, revealing systematic barriers to KBE adoption including fragmented knowledge management, steep learning curves for proprietary frameworks, integration complexity across heterogeneous toolchains, and organizational hesitancy driven by uncertain return on investment. Technical assessment confirmed that even state-of-the-art commercial language models consistently generate non-functional ParaPy code when prompted without domain-specific augmentation, validating the need for specialized intervention beyond general-purpose AI coding assistants.

The developed dual-agent framework directly addresses this gap through four principal contributions. First, a provider-agnostic architecture supports both local and API-based deployment with flexible model selection, satisfying diverse security requirements encountered in industrial contexts while enabling air-gapped operation when necessary. Second, a comprehensive knowledge infrastructure employs semantic search over indexed ParaPy documentation, examples, and API specifications, enabling dynamic information retrieval without computationally expensive fine-tuning or dependency on static prompt engineering alone. The system maintains three distinct indices optimized for different retrieval patterns, performing lazy evaluation to minimize token consumption while maintaining access to comprehensive domain knowledge. Third, progressive verification mechanisms ensure syntactic and runtime correctness through automated validation tools integrated into the generation pipeline. Fourth, role-specialized agents optimize for distinct user needs: a Developer Agent focused on code generation, completion, and debugging, and an Educational Agent providing conceptual guidance and documentation-based learning support.

Comprehensive evaluation through automated testing, manual case studies, and controlled user testing with nine industry participants validated framework effectiveness against established requirements. The framework achieved substantial productivity improvements, with novice users demonstrating 1300% gains and intermediate users 383% gains in features implemented per minute. Code quality metrics confirmed that generated implementations meet professional standards, with Claude Sonnet 4 achieving 86.1% syntactic correctness, 77.8% runtime correctness, and average quality scores of 7.88 out of 10. The Turbine Rear Structure case study demonstrated that detailed natural language specifications enable generation of production-quality implementations in approximately 2% of expert manual development time.

However, user testing revealed nuanced patterns in framework effectiveness across expertise levels. Intermediate users derived the greatest benefit, achieving 37.9% composite score improvements, while expert users experienced slight performance regression despite increased development velocity. This differential impact challenges assumptions about universal productivity enhancement, suggesting instead that AI coding assistants function most effectively as accelerators for users with sufficient domain knowledge to critically assess generated outputs but who still face knowledge gaps that the framework addresses. Novice users achieved 87.5% of expert baseline task completion rates with 55.5% reduction in domain-specific errors, validating the framework's effectiveness in reducing knowledge barriers. These findings reveal an "optimal assistance zone" where tool capabilities align precisely with user needs, requiring both conceptual understanding to formulate effective queries and recognize semantic violations, and knowledge gaps that semantic search and verification tools effectively address.

The research acknowledges fundamental limitations establishing interpretive boundaries for the findings. The framework exhibits persistent constraints in spatial reasoning and 3D geometric understanding, stemming from underlying language model architectures lacking inherent geometric intuition. This limitation manifests as incorrect component positioning and orientation in complex assemblies, requiring either explicit spatial guidance or iterative human-in-the-loop refinement. The Developer Agent occasionally misclassifies ParaPy GUI runtime errors as known SDK limitations rather than faults in generated code, disrupting autonomous error resolution. The evaluation methodology operated under finite sampling constraints, with user testing limited to nine participants and automated evaluation covering bounded test cases. While stratified by expertise level, samples may not capture full diversity of ParaPy development scenarios across different industrial contexts.

Despite these limitations, the research makes significant contributions to both Knowledge-Based Engineering practice and AI-assisted software development theory. The empirical validation of retrieval-augmented generation as viable alternative to fine-tuning in data-sparse domains informs ongoing debates about language model adaptation strategies. The automated code quality framework provides scalable methodology for evaluating AI-generated specialized code that extends beyond ParaPy to other domain-specific development contexts. The concept of an optimal assistance zone where intermediate expertise maximizes AI tool benefit has broader implications for deployment strategies across various technical domains.

This work demonstrates that AI-powered coding assistants can play valuable roles in industrial KBE development when deployed with clear understanding of their limitations, supported by robust verification mechanisms, and integrated into organizational practices preserving human expertise and oversight. The framework meaningfully addresses identified research gaps while defining boundaries of current AI-assisted development capabilities. Rather than replacing human expertise, effective AI assistance augments performance for users capable of interpreting, correcting, and integrating generated suggestions. The observed quality decline among experts under time constraints emphasizes the need for organizational safeguards maintaining standards in accelerated workflows. As generative AI tools continue evolving, the insights and evidence presented provide grounded foundation for responsible adoption supporting human engineers rather than replacing them, enabling acceleration of development velocity while preserving critical evaluation and domain expertise essential to high-quality engineering work.

9

Discussion & Recommendations

This chapter interprets the empirical findings from verification and validation, synthesizing them within the broader context of AI-assisted software development and Knowledge-Based Engineering practice. Rather than restating results, the discussion examines why particular outcomes emerged, what they reveal about the nature of AI-assisted KBE development, and how these insights should inform future research and industrial deployment strategies.

Interpretation of Core Findings

The user testing results revealed a counter-intuitive pattern where intermediate users derived the greatest benefit from the framework, achieving 383% improvement in features per minute and 37.9% composite score improvement, while expert users experienced slight performance degradation with 13.4% composite score decline despite comparable raw productivity gains. This outcome contradicts assumptions that AI coding assistants universally enhance productivity, instead suggesting a more nuanced relationship between tool effectiveness and user expertise.

This differential impact can be explained through what might be termed an "optimal assistance zone" where the tool's capabilities align precisely with user needs. Intermediate users possess sufficient domain knowledge to critically evaluate generated code but still face knowledge gaps that the framework effectively addresses through semantic search infrastructure surfacing relevant API methods and usage patterns. They understand ParaPy's core concepts well enough to recognize syntactically correct but semantically flawed suggestions, yet benefit substantially from documentation retrieval and verification tools. Expert users, having internalized common patterns through extensive practice, gained less from retrieval augmentation and exhibited reduced critical evaluation under time pressure, prioritizing development velocity over thorough validation. The error rate metrics confirm this pattern: experts showed 44% increase in errors per feature when using AI assistance, while novices demonstrated 55.5% reduction and intermediate users 87% reduction. Novice users, despite showing substantial error reduction, lacked the conceptual framework to formulate effective queries or recognize when generated code violated deeper design principles, though the Educational Agent partially addressed this limitation.

Furthermore, the framework exhibited consistent difficulties with 3D geometric reasoning across all evaluation contexts, with generated code frequently mispositioning components or applying incorrect rotations. This limitation reflects a fundamental architectural constraint of transformer-based language models [47], which learn statistical patterns over discrete token sequences but lack inherent geometric intuition. While models can produce syntactically valid coordinate transformations and geometric operations, they cannot reason about the spatial relationships these operations create.

This constraint was significantly exacerbated by deliberate omissions in framework design. The curated example set within the Developer Agent system prompt lacked demonstrations related to positioning and orientation, despite these concepts being consistently identified as the most challenging aspects of KBE application development. Positioning examples were limited to basic transformations such as `translate` and `rotate90`, leaving agents dependent on learned knowledge and incidentally retrieved positioning information through semantic search, which proved insufficient for complex spatial tasks. Additionally, the system prompt deliberately excluded information on ParaPy's distinguishing features including dependency tracking and lazy evaluation mechanisms, reflecting an initial assessment that these concepts were not essential for generating syntactically and runtime-correct code. While generated implementations achieved high correctness rates, this omission may have contributed to suboptimal exploitation of ParaPy's reactive programming model.

Furthermore, the LLM-judge methodology assessed PEP-8 compliance as a code quality dimension weighted at 25%, yet agents received no explicit instructions or reference materials on Python coding standards beyond their pre-training knowledge. This discrepancy between evaluation criteria and generation guidance may have introduced systematic assessment bias, as models were judged on adherence to standards they were not explicitly instructed to follow. The decision to rely on models' pretrained knowledge reflected a design trade-off between prompt length and specificity, but explicit inclusion of style guidelines could improve consistency.

The finding reinforces broader understanding that retrieval-augmented generation and tool use cannot fully compensate for capabilities absent from the base model architecture [58]. Semantic search provides syntactic patterns and API specifications but cannot supply geometric reasoning. This suggests an upper bound on what inference-time interventions can achieve without architectural modifications. However, the successful case of participant P7 achieving 100% functional correctness through human-in-the-loop iteration demonstrates that spatial correctness can be attained through structured workflows. By providing detailed descriptions of geometric errors, the user enabled the Developer Agent to converge on correct solutions within the time limit, suggesting a practical mitigation strategy where the framework accelerates iteration cycles through rapid generation-evaluation loops guided by explicit spatial feedback.

Validation of RAG for Sparse Training Domains and Technology Landscape

The framework's successful generation of functional ParaPy code despite minimal representation in public training data provides empirical validation for retrieval-augmented generation as a viable alternative to fine-tuning in data-sparse domains [72]. Baseline assessments established that even state-of-the-art models produce non-functional ParaPy code when operating without domain-specific augmentation. The full implementation, integrating semantic search and progressive verification, achieved 86.1% syntax correctness and 77.8% runtime correctness for Claude Sonnet 4, representing substantial improvement occurring entirely through inference-time interventions. However, results also reveal limitations of pure RAG approaches. The persistent spatial reasoning difficulties indicate that some capabilities cannot be injected through retrieval alone, suggesting a more nuanced understanding of when RAG suffices versus when architectural modifications or fine-tuning become necessary.

During the research period, the landscape of AI-assisted development tools evolved rapidly. Claude Code, which integrates large language models with command-line interfaces and supports the Model Context Protocol [73] for tool integration, emerged as a potentially relevant platform. However, several factors justified development of a custom framework. First, Claude Code became publicly available late in the research timeline, after core architectural decisions had been established and implementation was substantially complete. The rapid pace of innovation in generative AI makes it challenging to build academic research around continuously shifting technological foundations. Second, implementing the framework as a subagent within Claude Code would place the solution behind a commercial paywall, limiting accessibility for organizations with budget constraints or requiring air-gapped deployment. The custom framework's support for both local execution via Ollama and privacy-compliant API services addresses diverse industrial requirements that proprietary platforms may not satisfy. Third, the research aimed to demonstrate principles applicable beyond any single platform, establishing generalizable insights about retrieval-augmented generation, progressive verification, and dual-agent architectures that transcend specific implementation choices.

Nevertheless, converting semantic search engines to MCP-compliant servers would enable the framework to leverage ecosystem developments automatically, facilitate benchmarking against alternative approaches, and reduce vendor lock-in by supporting multiple agent frameworks. This approach balances the research goal of platform-independent validation with pragmatic recognition that standardization accelerates adoption and community contribution.

Recommendations for Future Work

Based on the empirical findings and theoretical analysis, the following recommendations guide future research and development:

Develop geometric feedback mechanisms. Implement automated geometric validation tools that compare generated component positions and orientations against explicit spatial requirements, detecting positioning errors without visual inspection. Such tools could operate as additional verification layers in the progressive verification pipeline, instantiating components, extracting spatial properties, and comparing them against formalized requirements. While technically challenging, automatic geometric validation could substantially elevate functional correctness.

Investigate fine-tuning as complementary strategy. While the current research deliberately excluded fine-tuning to validate RAG effectiveness in isolation, future work should explore fine-tuning on ParaPy-specific datasets as both an alternative and enhancement to the retrieval-based approach. Fine-tuning could address limitations in spatial reasoning, dependency tracking understanding, and idiomatic pattern generation that retrieval alone cannot fully resolve. Training incorporating geometric visualizations alongside code may develop stronger spatial reasoning capabilities [44]. Combining fine-tuned models with semantic search infrastructure may yield superior performance to either technique in isolation.

Enhance system prompt coverage. Revise Developer Agent system prompts to explicitly include PEP-8 style guidelines, comprehensive positioning and orientation best practices with detailed examples, and explanations of ParaPy's distinguishing features including dependency tracking and lazy evaluation. While semantic search compensates for such omissions, static inclusion of fundamental concepts ensures consistent guidance across all generation scenarios.

Convert to MCP-compliant architecture. Refactor knowledge infrastructure and tools as standardized MCP servers to enable broader adoption, facilitate community contributions, and reduce maintenance burden while maintaining interoperability with multiple agent platforms.

Extend verification beyond syntax and runtime. Enhance the progressive verification system to include semantic correctness checks through formal verification methods, property-based testing, or integration with additional static analysis tools.

Develop deployment best practices. Create comprehensive guidance for organizations adopting the framework, including staged rollout strategies targeting intermediate users first, training programs emphasizing critical evaluation of generated code, and quality assurance processes tailored to different user segments.

Generalize beyond ParaPy. Extract reusable components and patterns to create a framework-agnostic approach applicable to other domain-specific programming environments with limited training data, demonstrating broader applicability and facilitating knowledge transfer.

Concluding Remarks

This research demonstrates that AI-powered coding assistants can meaningfully support Knowledge-Based Engineering development for proprietary frameworks when appropriately augmented with domain-specific knowledge infrastructure and progressive verification mechanisms. The framework's effectiveness varies substantially across expertise levels, revealing an optimal assistance zone where intermediate users derive maximum benefit. Persistent spatial reasoning limitations reflect fundamental constraints of transformer-based architectures, establishing clear boundaries for inference-time interventions. The successful application of retrieval-augmented generation validates this approach for data-sparse domains while highlighting scenarios where fine-tuning or architectural modifications become necessary.

For aerospace industry practitioners and KBE developers, the framework offers immediately deployable capabilities for accelerating development velocity and reducing knowledge barriers. However, success requires careful attention to deployment strategies, user training, and organizational safeguards preserving quality standards. The framework should be positioned as a development accelerator requiring expert oversight rather than an autonomous code generator, with particular focus on intermediate users who benefit most while possessing sufficient expertise for critical evaluation.

As generative AI capabilities continue advancing, the principles demonstrated here provide foundation for responsible integration of AI assistance in engineering practice. Thoughtful architecture design, progressive verification, expertise-aware deployment, and honest acknowledgment of limitations enable strategic augmentation of human expertise rather than replacement. The goal remains supporting engineers in producing high-quality work more efficiently while preserving the critical thinking and domain knowledge essential to engineering excellence.

Part II

Technical Report

10

Language Modelling

Early approaches to language modelling were *statistical* in nature. A language model assigns probabilities to sequences of words, and the simplest models used n-gram statistics – probabilities of the next word based on the previous $n - 1$ words. For example, a unigram model uses single-word frequencies, a bigram model conditions on the last word, and so on. These n-gram models were supported by information-theoretic insights from the mid-20th century. In 1951, [80] estimated the entropy of the English language by asking humans to predict upcoming letters, effectively constructing a basic n-gram predictor. Such models capture local word correlations but suffer when long-range context or unseen word combinations appear. Work in the 1980s and 90s (e.g., Jelinek and others at IBM) refined n-gram modelling with smoothing techniques to handle sparsity, yet the fundamental "curse of dimensionality" remained: a language model would likely encounter sequences at test time that never appeared in training [81].

A major paradigm shift occurred in the early 2000s with the introduction of *neural language models*. [81] proposed a Neural Probabilistic Language Model that learns a *distributed representation* (vector embedding) for each word, in addition to learning the next-word probability function. By mapping words to continuous vectors, the model generalizes to unseen word sequences: if a new sequence contains words similar (in embedding space) to a known sequence, it can assign it a higher probability [81]. This model, implemented as a feed-forward neural network, significantly outperformed traditional n-grams by leveraging learned word similarities. Soon after, [82] popularized efficient techniques to learn word embeddings on large corpora (the *Word2Vec* skip-gram and CBOW models in 2013), demonstrating that embedding spaces can capture semantic relationships (e.g., $\texttt{king} - \texttt{man} + \texttt{woman} \approx \texttt{queen}$). Word embeddings became a foundational idea in NLP, enabling neural models to represent text in a dense, informative way rather than as sparse one-hot vectors.

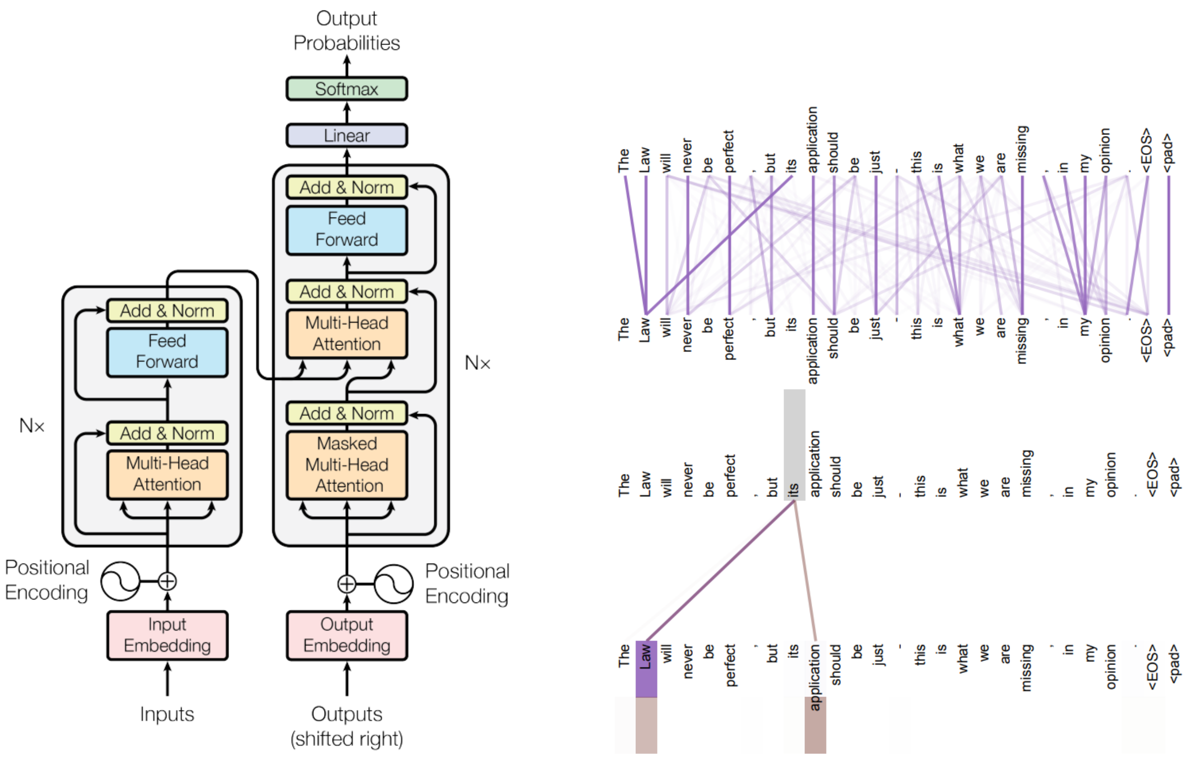
10.1. Sequence Models

While feed-forward networks with embeddings handled fixed-length context windows, researchers sought models that could naturally handle sequences of arbitrary length. *Recurrent Neural Networks (RNNs)* achieved this by maintaining a hidden state that is updated word by word, effectively reading in sequences of unbounded length. In practice, however, vanilla RNNs were hard to train on long sequences due to vanishing/exploding gradients. The introduction of the *Long Short-Term Memory (LSTM)* architecture by Hochreiter and Schmidhuber (1997) addressed this issue by incorporating gating mechanisms [83]. LSTMs use an explicit memory cell and gates (input, forget, output) to control information flow, thereby preserving long-range dependencies and mitigating gradient decay. With LSTMs (and the simplified *Gated Recurrent Unit, GRU*, introduced in 2014), RNN-based language models could finally capture much longer context than n-grams. Throughout the 2010s, LSTMs became the workhorse for language modelling and translation, achieving state-of-the-art results on many benchmarks. [83]

However, a limitation of RNN/LSTM models is their *sequential* nature: processing tokens one by one prevents parallelization across sequence positions during training. This made training on very long sequences slow and computationally costly [47]. To address both efficiency and long-range modeling, researchers also experimented with *Convolutional Neural Networks (CNNs)* for language. CNN-based sequence models apply learned filters over text windows and can be stacked to increase receptive field. Notably, a fully convolutional seq2seq model by [84] showed that CNNs can parallelize computations over all timesteps and still achieve competitive accuracy with LSTMs. Their convolutional model outperformed a deep LSTM on large-scale translation tasks while running an order of magnitude faster by leveraging parallelism. Although CNNs do not capture arbitrarily long dependencies as easily as RNNs (they rely on stacking or dilations for wider context), this work proved that recurrence was not the only way to model sequences. [84]

10.2. The Transformer Architecture

The next revolution came with the *Transformer* architecture introduced in 2017 by Google Brain [47]. The Transformer eliminated recurrence entirely, relying on *self-attention* mechanisms to model dependencies between any pair of tokens, regardless of their distance. In a Transformer, each input token position attends to all other positions within the sequence, computing weighted averages of their representations. Crucially, this attention mechanism is highly parallelizable: all tokens in a layer can be processed simultaneously (since positional dependencies are handled via attention weights rather than sequential state passing). This enabled training on very long sequences with much greater speed than RNNs, overcoming the sequential bottleneck that had limited previous models. The Transformer's success was immediate – it not only improved machine translation quality, but soon became the backbone of nearly all state-of-the-art language models. Within a year, the Transformer was adapted for language understanding in the form of BERT (Bidirectional Encoder Representations, 2018 [51]) and for language generation in GPT (Generative Pre-trained Transformer, first version in 2018 [52]). These models demonstrated the power of pre-training on massive text corpora and then fine-tuning for downstream tasks, kicking off the era of *pre-trained language models*. [47]



(a) Transformer architecture.

(b) Attention heads.

Figure 10.1: Overview of transformer neural network architecture and internal attention mechanisms. (a) illustrates the complete structural design of the transformer model, while (b) examines how the model internally processes language by tracking connections between words, specifically demonstrating how certain components identify and link pronouns to their corresponding references in text. [47]

Modern Large Language Models (LLMs) build upon the Transformer architecture and scale it to unprecedented sizes. We define Large Language Models as neural networks with billions of parameters trained on vast text corpora to predict and generate human-like text. The GPT series in particular showed that increasing model size and training data leads to remarkable emergent capabilities. **GPT-2** (2019) with 1.5 billion parameters already produced fluent long-form text, and shortly after, **GPT-3** (2020) pushed the scale to 175 billion parameters, introducing the phenomenon of few-shot learning [48]. GPT-3 demonstrated that a sufficiently large Transformer-based LM can perform new tasks from only a few examples or instructions given in the prompt (without parameter updates). This was a striking result: scaling up the model and training on diverse internet text made it surprisingly general and adaptable. Subsequently, even larger models have been developed (e.g., with hundreds of billions or trillions of parameters), though often with sparse or mixture-of-experts architectures to manage efficiency. Researchers also improved LLM capabilities via fine-tuning approaches. One key development was instruction tuning and reinforcement learning from human feedback (RLHF) to make models follow user instructions and preferences [85]. This yielded more interactive and helpful models, the most famous example being ChatGPT (released by OpenAI in late 2022), which is essentially a GPT-based model fine-tuned to produce conversational, user-aligned responses.

A typical large Transformer-based LM today operates as follows. First, the input text (prompt) is tokenized (e.g. into subword units) and converted to input embeddings. Positional encodings are added to indicate token positions. The Transformer then processes the sequence through multiple layers of self-attention and feed-forward networks [47]. In a *decoder-only* LM (like GPT), each token attends to earlier tokens (causal attention) and the model outputs a probability distribution over the next-token vocabulary at each step; in an *encoder-decoder* model (like the original Transformer or T5), an encoder first processes the source text (e.g. a prompt or context) bidirectionally, and a decoder then generates outputs attending both to the encoder representations and earlier decoder outputs [47, 86]. In either case, generation is done iteratively: at deployment time, the model produces one token at a time, feeding it back in as input in an auto-regressive manner [48]. The context window of the model defines how many tokens it can attend to at once; anything beyond that is truncated or must be handled via techniques like windowed attention or external memory. Importantly, Transformers do not have persistent long-term memory beyond this context window – they are stateless between inference calls, which is why prompt engineering (including relevant information in the prompt) is crucial to getting correct outputs [48, 49, 50].

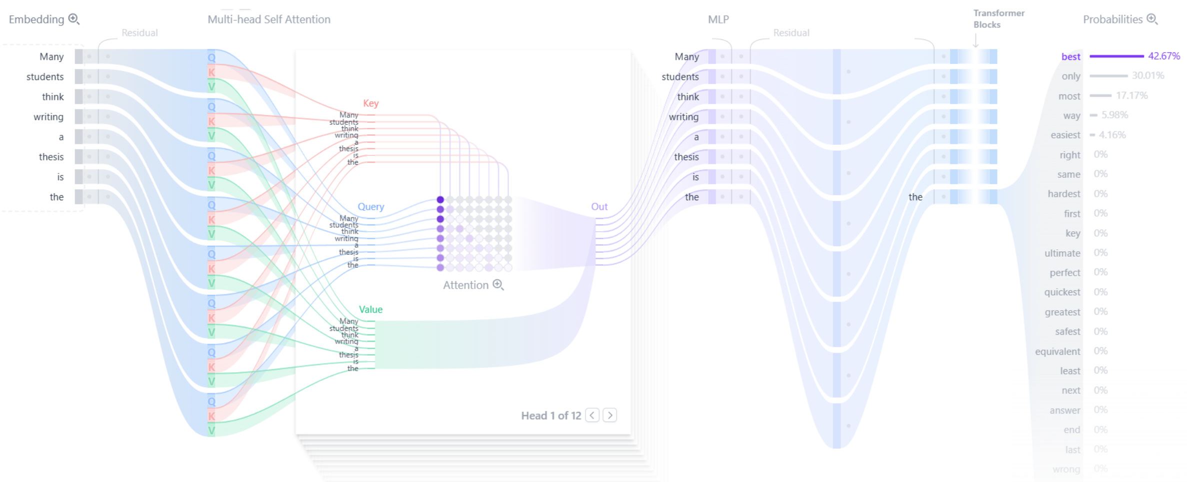


Figure 10.2: Interactive visualization of transformer language model processing pipeline from the Transformer Explainer tool. The diagram illustrates the complete data flow through a transformer block, showing how input embeddings (left) are processed through multi-head self-attention mechanisms (centre-left) where Query, Key, and Value vectors compute attention weights, followed by multi-layer perceptron processing (centre-right), and culminating in the final probability distribution over vocabulary tokens (right) for next-token prediction. From github.io/transformer-explainer [accessed 21-08-2025].

When producing text, LLMs use a decoding strategy to sample the next token from the probability distribution. Choices include *greedy decoding* (always pick highest probability token), *beam search*, or stochastic methods like *top-k sampling* and *nucleus sampling* that introduce randomness to generate more varied outputs [87, 50]. The strategy can significantly affect the style and correctness of generated text. For example, code generation often benefits from more deterministic decoding (to preserve syntax), whereas open-ended creative writing may use nucleus sampling to avoid repetitive or overly conservative outputs [87]. The flexibility of decoding methods allows a single trained model to serve different purposes. In practical deployments (e.g., via an API or an interactive system), these models are often augmented with safety filters or external tools, but fundamentally the process remains: encode prompt into internal embeddings, repeatedly apply self-attention layers to update representations, and sample next tokens until completion [47, 85, 59, 58].

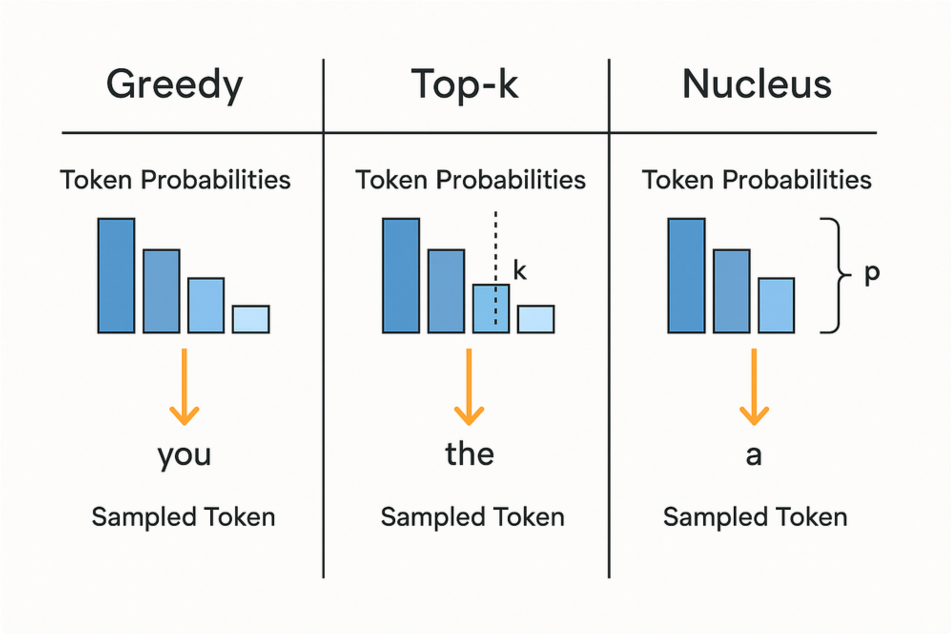


Figure 10.3: Overview of common sampling techniques and how they influence next token selection. Greedy search always picks the highest probability token; top-k sampling chooses from the top k likely tokens; nucleus sampling considers all tokens whose combined probability mass $> p$. From ashutosh.dev [accesses 04-09-2025].

10.3. From Natural Language to Programming

Programming languages are formal languages, but they share many characteristics with human language: a vocabulary (keywords, identifiers), syntax rules (grammar), and even stylistic conventions [35]. It is therefore natural to apply language modelling techniques to source code. Indeed, code can be treated as just another form of text, and many LLMs have been trained or fine-tuned on large code corpora (such as open-source GitHub repositories) in addition to natural language data [54]. By learning from millions of code examples, an LLM can statistically infer how to write syntactically correct and plausible code to accomplish a given task described in natural language. For instance, an LLM-based system can take a prompt like "Implement a Python function to check if a number is prime" and generate a complete function in Python to do so. This ability has huge practical implications: it can automate boilerplate coding, assist in software development, and enable conversational interfaces for programming.

Several dedicated code-focused LLMs have emerged. OpenAI's Codex (2021) was a version of GPT-3 fine-tuned on billions of lines of source code, achieving impressive results in generating correct solutions to programming challenges (it was evaluated on the HumanEval benchmark of Python problems, solving a majority of them) [54]. Codex powers GitHub's Copilot, an AI pair-programmer that suggests code completions inside IDEs [88]. Copilot is trained on public GitHub code and is specialized at producing code that not only is syntactically correct but also matches common libraries' usage patterns. Other organizations have introduced their own code models; for example, DeepMind's AlphaCode and Google's PaLM-Coder have tackled competitive programming problems, and open-source efforts like StarCoder (by HuggingFace BigCode) and DeepSeek-Coder (by Peking University) have released reasonably sized models (6–30B parameters) that perform well on code generation benchmarks [89].

These models are all based on Transformers and are often initialized from a general LLM then further trained on code, as code has different characteristics (for example, needing to predict exact punctuation, indents, and respecting programming syntax).

One key difference between natural language and programming is that code execution is unforgiving to errors. A grammatical mistake in an English sentence might still be understood by a reader, but a syntax error in code will prevent it from running. Likewise, a slightly off factual statement in prose might go unnoticed, but a single incorrect API call in code can cause a program to crash. Thus, *adherence to syntax and semantics* is paramount in code generation. LLMs generally have excellent syntax adherence for languages seen frequently in training (such as Python, JavaScript, etc.), often producing code that compiles or runs on first try. This is evidenced by high scores on competitive coding benchmarks achieved by state-of-the-art models like GPT-4 or Anthropic's Claude [54]. However, ensuring semantic correctness (that the code does what it's supposed to do) is a harder challenge. Models may produce logically flawed or inefficient code that superficially looks right. There is ongoing research on evaluating functional correctness via test suites and on making LLMs not just syntactically but semantically aligned with user intent.

A particularly challenging scenario for code LLMs is dealing with unfamiliar or under-represented libraries and frameworks [56, 57]. An LLM cannot have true "knowledge" of an API it was never trained on; at best it can make educated guesses by analogy to similar libraries [57]. In fact, a LLM does not possess true "knowledge" of any API—public or company-specific—but the abundance of public libraries in training data ensures higher statistical likelihood of producing correct usage; conversely, models benefit markedly when fine-tuned on private, in-house code and APIs [28]. Unlike a human programmer who might read documentation when encountering an unknown library, a vanilla LLM has no retrieval mechanism at generation time (unless augmented with external tools) [58, 59]. As a result, when asked to generate code using a rare or proprietary library, the model is prone to *hallucinate*—i.e., to fabricate function/package names or usage patterns that sound plausible but are incorrect [56, 60, 61]. Hallucination in code generation is often easy to detect empirically: the generated code fails to compile/run or does not pass tests [54, 61]. Studies also show that LLMs will frequently produce an answer rather than abstain when uncertain, which contributes to hallucinations [62, 63, 64].

When the target library is a closed-source, domain-specific framework, like ParaPy, this issue becomes pronounced. Because such code is largely absent from public repositories, a general-purpose LLM has likely never seen the library's classes or functions during training [57]. Consequently, when prompted to generate Python code with ParaPy, the model may invent class names, methods, or import statements that do not exist in the actual SDK [56, 60]. Repository-level studies and industrial deployments observe the same pattern: without codebase-specific knowledge, hallucinations increase and correctness drops [61, 28]. Mitigating this requires techniques beyond standard LM training, such as retrieval-augmented generation (providing relevant documentation or code snippets to the prompt) or fine-tuning the model on the specific library [58, 65, 28, 56].

11

Implementation Details

11.1. Large Language Model Settings

Maximum number of tokens

A hard limit on generated tokens that prevents excessive output and controls computational costs. When reached, generation terminates immediately regardless of response completeness. Tokens are discrete text processing units that may not correspond directly to words, as tokenisation schemes often split words into multiple tokens.

Model temperature

Controls token selection by modifying the probability distribution during softmax normalisation. Values below 1 ($T < 1$) concentrate probability on higher-likelihood tokens, producing more deterministic outputs suitable for code generation. Values above 1 ($T > 1$) flatten the distribution, increasing creativity for generative tasks. The optimal setting varies by task, model architecture, and other sampling parameters. While $T = 0$ approaches deterministic selection, implementation details and floating-point precision may introduce minor variations.

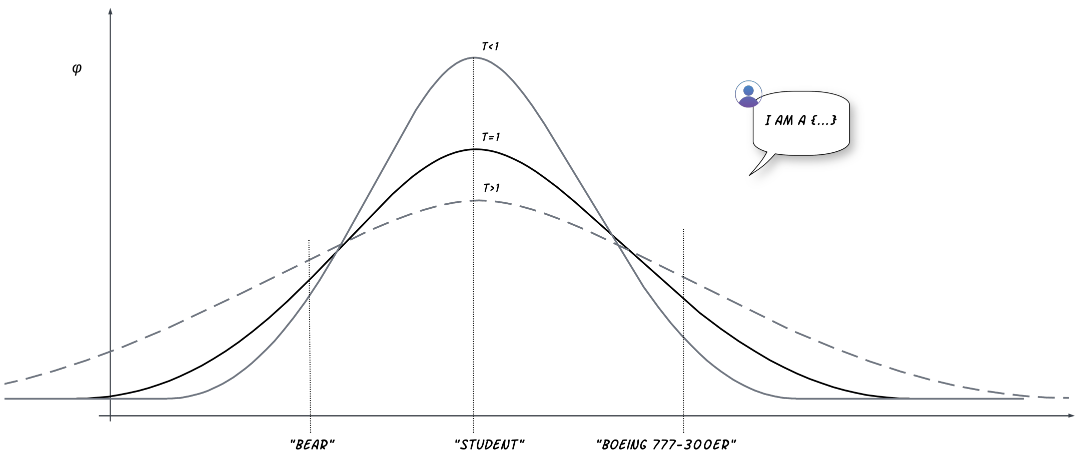


Figure 11.1: The effect of model temperature on the output probability distribution and subsequent effect on sampling.

Type of sampling

Sampling algorithms (detailed in Chapter 3) determine how the model selects tokens from the probability distribution. Greedy sampling, which selects the highest probability token, produces optimal results for code generation tasks. Stochastic methods such as top-k and nuclear sampling perform better for natural language generation. The sampling method directly affects output determinism and creativity.

Token penalty

[Framework-specific] Reduces repetitive content by penalising tokens based on their previous appearance or frequency. This mechanism decreases the likelihood of repeated tokens, promoting more diverse output generation.

(Logit) bias

[Framework-specific] Enables fine-grained control over vocabulary by manually modifying the likelihood of specified tokens. Developers can increase or decrease the probability of specific terms appearing in the output, useful for enforcing terminology conventions (KBE, ParaPy).

Stop sequence

[Framework-specific] Defines custom sequences that trigger generation termination. While models typically stop upon producing an end-of-sequence token (e.g., `<end>`)—which generally occurs when the preceding text statistically resembles natural endpoints seen in the model's training data—custom stop sequences allow for the specification of domain-specific patterns (such as punctuation marks, keywords, or formatting tokens) that indicate completion. This feature is particularly useful for structured outputs or when integrating models with external systems that rely on specific delimiters or formatting conventions.

The latter three settings often vary across frameworks. For example, Pydantic AI includes provider-specific settings to enforce particular reasoning behaviours. Not all settings are compatible with all providers, particularly Ollama models. To ensure maximum compatibility, the current framework modifies only general settings applicable across all models. After configuration, the LLM is ready for deployment in agent applications, operating either remotely or locally as required.

11.2. Prompt Engineering Strategies

Use prompt templates

Templates with variables for dynamic content (user inputs, documentation, tool calls) improve prompt management. In Python, this utilises string `Template` objects or f-strings. Benefits include:

- **Consistency** across interactions
- **Efficiency** through modular content swapping
- **Testability** with rapid case testing
- **Scalability** as complexity increases
- **Version control** through separation of core prompts from dynamic inputs

Be clear and direct

Commercial Off-The-Shelf (COTS) models lack context about KBE conventions and ParaPy-specific requirements. Precise explanation of requirements produces better results. Anthropic's "golden rule of clear prompting" states: "Show your prompt to a colleague, ideally someone who has minimal context on the task, and ask them to follow the instructions. If they are confused, the model [Claude] will likely be too." [79]

Use examples (multishot prompting)

Well-crafted examples can dramatically improve the accuracy, consistency and quality of the output. Effective examples are:

- **Relevant** to expected use cases;
- **Diverse**, covering edge cases and potential challenges;
- **Clear**, with consistent formatting and tagging, e.g. `<example>`.

~~~~
<example>
Input: Give me a parametric box using the ParaPy SDK.
Code:
```
from parapy.core import Attribute, Input
from parapy.geom import GeomBase

class Box(Base):
    width: float = Input()
    depth: float = Input()
    height: float = Input()

    @Attribute
    def volume(self) -> float:
        return self.width * self.depth * self.height

```
</example>
~~~~

Chain of thought (CoT) prompting

CoT prompting improves performance on complex tasks by encouraging step-by-step reasoning. Benefits include:[79]:

- **Accuracy** in mathematical calculations and logical analysis or generally complex tasks;
- **Coherence** through structured thinking;
- **Debugging** capabilities by exposing the reasoning process.

CoT approaches include:

1. **Basic prompt**: including "think step-by-step" in the prompt;
2. **Guided prompt**: outlining specific reasoning steps;
3. **Structured prompt**: using XML tags to separate reasoning from final output.

This technique increases both output length and latency but can significantly improve performance on complex tasks. However, not all tasks require such in-depth reasoning, and developers may risk overengineering this aspect of the prompt design.

Using XML tags

XML tags organise multi-component prompts, improving:

- **Clarity** through component separation;
- **Accuracy** by reducing misinterpretation;
- **Flexibility** in prompt management;
- **Parseability** of structured outputs.

Best practices include consistent tag naming and hierarchical nesting for complex aerospace engineering prompts.

Role prompting

Assigning a specific role to the model (e.g., "aerospace engineer specialising in knowledge based engineering") enhances performance through:

- **Enhanced accuracy** in designs analysis and software engineering tasks;
- **Tailored tone** appropriate for technical engineering communication;
- **Improved focus** on aerospace, KBE and/or ParaPy requirements.

Major model providers, such as Anthropic, offer prompt generator and improver tools to support the process of prompt engineering. These tools often include additional resources for defining success criteria, generating test cases, and designing evaluation procedures. For full documentation, the reader is referred to Anthropic's official documentation.

11.3. MCP Servers

Despite not leveraging MCP[1] servers, the framework remains fully MCP-compliant, allowing future replacement or extension. As such, the in-house semantic search engine could be easily converted into an MCP-compatible implementation, offering a pathway to broader adoption and standardization. Examples of such MCP implementations include:

- **Documentation MCPs** such as Context7, Docs, Docy, and MCP Documentation Server, which retrieve up-to-date documentation and examples from source repositories and inject them into prompts.
- MCP Run Python, an MCP developed by Pydantic AI, which enables secure, sandboxed Python code execution.

An extended, community-maintained list of reference implementations is available via the official MCP GitHub organization at: Model Context Protocol Servers.

Implementation Note

While the framework is fully compatible with MCP servers, none are used in this research. At the time of writing, these servers offer no meaningful advantages over the in-house semantic search solution described in section 6.4. Most documentation MCP servers do not support hosting or isolating local datasets, and those that do typically adopt design principles similar to those already implemented here. As a result, adopting external MCPs would reduce control and flexibility without delivering added value.

The same rationale applies to Python-execution MCPs. Although sandboxing introduces clear security benefits, these environments typically do not support the execution of closed-source libraries such as ParaPy. Supporting such use cases would require custom extensions, which were deemed out of scope given the sufficient performance and safety of the current tool implementation.

11.4. LLM Tool Calling

Tool calling involves the use of a language model that has been trained or fine-tuned to generate structured tool-calling responses. Most large commercial models support this functionality by default, while open-source models—particularly smaller ones—may not consistently exhibit this behaviour. When they do, their names often include suffixes such as `*instruct`, `*chat`, or `*versatile` to indicate instruction-following or tool-compatible capabilities. Within an agent system, tool calling typically follows a thought–action–observation cycle, where the model reasons about the task, selects an appropriate tool, and integrates the tool's output into its ongoing response, Figure 5.3 visually demonstrates this. This process consists of the following steps:

1. Inclusion of available tool signatures (e.g., function definitions) in the system prompt, informing the model of the tools at its disposal.

```
system_prompt = """You are a useful AI assistant capable of calling tools to improve
    user experience.

You have access to the following tools:
{tools_description}
"""
```

   A typical tool signature might look like:

```
{
    "name": "get_weather_at_location",
    "description": "Find the current weather at :param ``location`` through our OS
        FreeWeather API.",
    "parameters": {
        "type": object,
        "properties": {
            "location": {
                "description": null,
```

[1]MCP servers provide standardised context to language models through JSON-RPC 2.0 protocol, as detailed in Chapter 3.

```
                "type": "string"
            }
        },
        "required": ["location"],
    },
}
"""
```

2. Streaming the model's output until a complete tool-calling schema is produced. Generation is paused at this point.

```
<thinking>The user asked a question related to the weather in Delft, this probably means I have to call the `get_weather_at_location` function, let me see</thinking>

{
    "action": "tool_call",
    "function": "get_weather_at_location",
    "input": {
        "location": "Delft, Netherlands"
    }
}
```

3. Executing the specified tool using the parameters provided in the schema.

```
>>> get_weather_at_location(location="Delft")
...
"The current weather in Delft, the Netherlands is: rainy with a non-existing change of sun."
```

4. Optionally validating the output returned by the tool.
5. Appending the tool's output to the current agent response.

```
<system>
You are a useful AI assistant capable of calling tools to improve user experience.

You have access to the following tools:
{
    "name": "get_weather_at_location",
    "description": "Find the current weather at :param ``location`` through our OS FreeWeather API.",
    "parameters": {
        "type": object,
        "properties": {
            "location": {
                "description": null,
                "type": "string"
            }
        },
        "required": ["location"],
    },
}
</system>
<user>
What is the weather like in Delft today?
</user>
<system>
<thinking>The user asked a question related to the weather in Delft, this probably means I have to call the `get_weather_at_location` function, let me see</thinking>

{
    "action": "tool_call",
    "function": "get_weather_at_location",
    "input": {
        "location": "Delft, Netherlands"
    }
}

<tool_out>
The current weather in Delft, the Netherlands is: rainy with a non-existing chance of sun.
```

```
</tool_out>
</system>
```

6. Resuming generation, optionally repeating the tool-calling process if additional calls are needed.

12

Detailed Design

12.1. The Pydantic Framework: Basics & Extensions

The primary framework used in this research is Pydantic AI, specifically its initial official release, version `1.0.0` (published on 05-09-2025). Although earlier development and testing were conducted using pre-release versions, the current implementation has been fully aligned and is compatible with version `1.0.0`. The core of the Pydantic AI (and the current) framework is the `Agent` class, which can be though of as a container for [90]:

- **LLM Model**
  The default language model associated with the agent. A Pydantic AI agent model consists of two core components: the `Model`, tailored to the specific communication protocol of the provider, and a `Provider`, which establishes the API connection. The agent framework abstracts over different providers via custom inference logic, offering a provider-agnostic interface. For instance, initializing an agent with the string `"groq:llama-3.3-70b-versatile"` will automatically instantiate a `GroqModel` with the correct `GroqProvider`.
- **Model Settings**
  Model settings further customize model behaviour and may include provider-specific parameters. These are passed as a dictionary to the `model_settings` argument. Common settings include:
  - `max_tokens` – The maximum number of tokens to generate before stopping.
  - `temperature` – Controls response randomness through soft-max normalization (see subsection 5.2.1).
  - `timeout` – Overrides the default request timeout (in seconds).
  - `top_p` – Enables nucleus sampling by considering tokens that make up the top-p probability mass (see subsection 5.2.1).
- **System Prompt & Instructions**
  System prompts and instructions serve as inputs to the LLM at runtime. These can be:
  - *Static* – Known at development time and passed directly via the `instructions` argument.
  - *Dynamic* – Derived at runtime through functions decorated with `@agent.instructions`.

  The distinction lies in persistence: when an agent is provided with a `message_history`, system prompts are retained, while instructions are only visible in the current interaction. This separation is especially useful in multi-agent setups where agents must be assigned distinct roles without cluttering shared context.
- **Structured Output Type(s)**
  Pydantic AI supports structured outputs to enhance type safety in GenAI workflows. These outputs may include standard Python types (e.g., lists, dictionaries), data classes, or full Pydantic models. Output adherence is enforced through one of three supported strategies [90]:

1. *Tool Output* – Uses tool calls to return structured data as JSON schemas; this is the default method.
2. *Native Output* – Relies on the LLM's built-in support for producing JSON-conformant text. Support varies by model.
3. *Prompted Output* – Embeds a JSON schema into the prompt and parses the response. Universally compatible but less reliable.

- **Dependency Type Constraint**
  Pydantic AI uses a dependency injection system to provide agents with additional runtime context. The `deps_type` field enforces type safety in this process. Incorrect dependency typing may result in runtime errors. Implementation details are discussed in section 6.6.
- **Function Tool(s) and Toolsets**
  Function tools enable agents to perform external actions or retrieve supplementary information to enhance response quality [90]. Tools may be grouped into `Toolsets`, which serve a common purpose and can be registered collectively. Tool calling allows LLMs to emit JSON schemas describing the tool to be called—a process handled transparently by the Pydantic framework. For background, see subsection 3.3.2 and subsection 5.2.4.

The aforementioned points synthesize into the following common signature for a Pydantic AI agent. This structure captures the key configuration elements required to instantiate an agent capable of tool use, structured output, dependency injection, and provider-agnostic model interaction. It serves both as a conceptual summary and as a practical template for initializing agents in the framework.

```
agent = Agent(
    model=..,  # type: str | Model
    model_settings=..,  # type: dict
    system_prompt=..,  # type: str
    output_type=..,  # type: object | Callable
    deps_type=..,  # type: object
    tools=...,  # type: Sequence[Tool]
)
```

12.1.1. Framework Customization

The `pydantic_ai.Agent` class uses the `model` argument to specify which language model the agent should deploy. This argument can be provided either as a string or as an explicit `Model` instance. When passed as a string, the model type is inferred from a prefix—for example, `'anthropic:model_name'` triggers the initialization of an `AnthropicModel` using the specified `model_name`, paired with the corresponding `AnthropicProvider`.

For local models deployed via Ollama (in accordance with the compatibility constraints discussed in the prescriptive study, see REQ-1-1), the user is required to manually instantiate both the model and its provider. This involves using the `OpenAIChatModel` class (as Ollama uses the OpenAI communication protocol as outlined in subsection 5.2.1) in combination with an `OllamaProvider`, as demonstrated below:

```
local_agent = Agent(
    OpenAIChatModel(
        model_name="qwen2.5-coder:latest",
        provider=OllamaProvider(base_url="https://localhost:11434/v1"),
    )
)
```

To simplify local deployment and unify the initialization process for all model types, a custom wrapper around the `Agent` class has been developed. This wrapper supports string-based model selection for both locally hosted Ollama models and models served via external API providers. It extends the built-in model inference logic by detecting when a local model is referenced, checking whether it is available on the host machine, and instantiating the corresponding `OpenAIChatModel` with the appropriate provider. Importantly, the wrapper preserves full compatibility with the original `pydantic_ai.Agent` class, including its method signatures and support for explicit `Model` instances. This enhancement allows for a more concise initialization syntax:

```python
local_agent = Agent("ollama:qwen2.5-coder:latest")
```

In addition to model inference adaptation, a local `run` method has been introduced in the agent wrapper. This method shadows the original `Agent.run` signature but adds an exception-handling layer to improve user experience, particularly in the CLI application discussed in section 6.8. If unexpected model behaviour is detected—most notably when structured output fails due to retry limits—the method attempts to return the `AgentRunResult` as a raw string. This fallback has proven useful in practice, as such failures often occur despite the model having produced a meaningful response. All other exceptions are re-raised to preserve standard debugging behaviour. The fallback behaviour can be disabled by setting `raise_ = True`, in which case the custom run method becomes functionally identical to `pydantic_ai.Agent.run`.

12.2. LLM Selection

LLaMa 3.1 (Local)

Meta AI - LLaMa 3.1

Version:	llama3.1:8b	**Architecture:**	Dense transformer
Training Data:	Public data from the Web	**Languages:**	8 languages
Parameters:	8B	**Cutoff:**	End of 2023
Context:	128K		

LLaMa 3.1 is Meta's latest generation of open-source foundation models that is also supported by Ollama, released in July 2024[1]. The family includes three models with 8B, 70B, and 405B parameters. The flagship LLaMa 3.1 405B is a dense Transformer with 405 billion parameters and a context window of up to 128K tokens. Meta opted for a standard dense Transformer model architecture with minor adaptations, rather than a mixture-of-experts model, to maximize training stability. The models were trained on over 16,000 H100 GPUs using over 15 trillion tokens, making the 405B the first LLaMa model trained at this scale. To support production inference, the models were quantized from 16-bit (BF16) to 8-bit (FP8) numerics, allowing the 405B model to run within a single server node. [91]

The pre-training dataset was created from various sources containing knowledge until the end of 2023, with much of the data obtained from web scraping. The final data mix comprises approximately 50% general knowledge tokens, 25% mathematical and reasoning tokens, 17% code tokens, and 8% multilingual tokens covering 176 languages. The web data underwent extensive cleaning including PII and safety filtering, HTML parsing optimized for mathematical and code content, and multiple rounds of de-duplication at URL, document, and line levels. Domain-specific pipelines extracted code and math-relevant content, with quality filtering performed using both heuristic methods and model-based classifiers trained on LLaMa 2 predictions. For post-training, Meta used synthetic data generation to produce the majority of supervised fine-tuning examples, with several rounds of alignment involving Supervised Fine-Tuning, Rejection Sampling, and Direct Preference Optimization. A final annealing phase on small amounts of high-quality code and mathematical data further boosted benchmark performance, though training sets from commonly used benchmarks were excluded to preserve true few-shot learning evaluation. [91]

[1] *"Introducing LLaMa 3.1: Our most capable models to date"*, from `https://ai.meta.com/blog/meta-llama-3-1/` [accessed 10-10-2025]

Claude Sonnet 4 (API)

Anthropic - Claude Sonnet 4			
Version:	claude-4-sonnet-20250514	**Architecture:**	Transformer
Training Data:	Internet corpora + proprietary	**Languages:**	Not disclosed
Parameters:	Not disclosed (> 100B)	**Cutoff:**	May 2025
Context:	200K		

Claude Sonnet 4 is part of the Claude 4 family, which includes two models: Claude Opus 4 and Claude Sonnet 4, both released in May 2025. The model features a 200,000-token context window for comprehensive document processing and a 64,000-token output capacity for extensive code generation and analysis. As is common with proprietary models, the exact number of parameters, architectural details, and supported languages have not been publicly disclosed.[2]

Claude Opus 4 and Claude Sonnet 4 were trained on a proprietary mix of publicly available information on the Internet as of March 2025, as well as non-public data from third parties, data provided by data-labelling services and paid contractors, data from Claude users who have opted in to have their data used for training, and data generated internally at Anthropic. The training process employed several data cleaning and filtering methods, including deduplication and classification. [92]

For web data collection, Anthropic operates a general-purpose web crawler that follows industry-standard practices with respect to "robots.txt" instructions and does not access password-protected pages or those requiring sign-in or CAPTCHA verification. The models were trained with a focus on being helpful, honest, and harmless, using techniques including human feedback, Constitutional AI based on principles such as the UN's Universal Declaration of Human Rights, and the training of selected character traits. [92]

12.3. Semantic Search Engine Design

This section details the implementation of the semantic search engine introduced in section 6.4. The architecture comprises three main components: the engine design and querying process, the filtering and indexing of data sources, and the integration into the agent toolset. The engine design and querying mechanism are described in the first subsection. The filtering and indexing strategies applied to ParaPy documentation, example sets, and API references are covered in the second subsection. The section concludes with integration details.

12.3.1. Engine Architecture, Design & Querying Process

Semantic search is an information retrieval technique that focuses on understanding the meaning, intent, and context of a user's query rather than relying solely on exact keyword matches[3]. Leveraging natural language processing, machine learning, and vector embeddings, semantic search enables systems to recognize synonyms, related terms, and contextual nuances to retrieve more relevant results.

In a typical semantic search engine, data is embedded into a high-dimensional vector space. Similarity metrics such as cosine similarity are then used to retrieve the most relevant entries. In this research, the semantic search engine serves as a core component of the dual-agent framework, enabling access to indexed information from three distinct sources: the ParaPy documentation, curated usage examples, and the ParaPy API reference.

This implementation uses the SentenceTransformers library developed by HuggingFace, which offers streamlined tools for applying state-of-the-art embedding and reranker models. The embedding model selected is `all-MiniLM-L6-v2`, chosen for its satisfactory semantic performance and lightweight footprint. While these models are technically language models, they are trained on ranked datasets to specialize in sentence similarity and embedding tasks. Unlike large generative transformers, these models are smaller, faster, and purpose-built for tasks like clustering and semantic retrieval. The

[2] *"Introducing Claude 4"*, from https://www.anthropic.com/news/claude-4 [accessed 06-10-2025]
[3] From https://www.geeksforgeeks.org/nlp/what-is-semantic-search/ [accessed 06-10-2025]

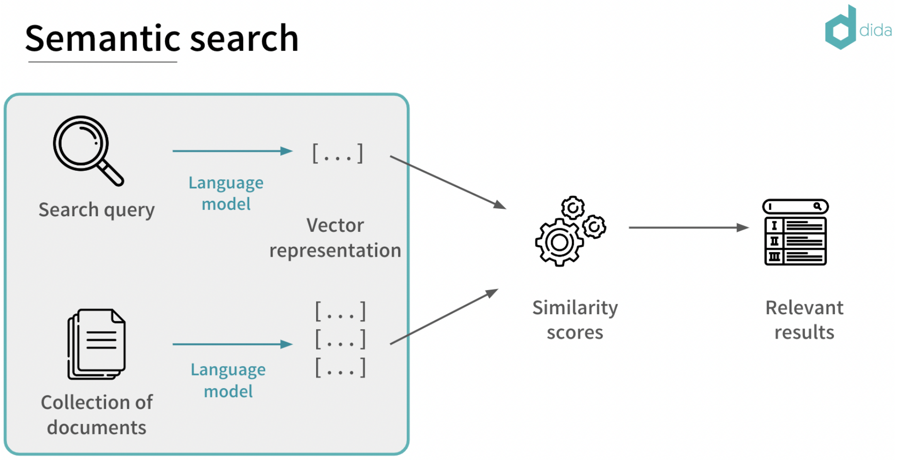


Figure 12.1: High-level overview of semantic search. Both indexed data and user queries are embedded into a shared vector space using a language model. Similarity scores (e.g., cosine similarity) are computed, and the top-matching entries are retrieved based on semantic relevance. From https://dida.do/what-is-semantic-search [accessed 06-10-2025].

`all-MiniLM-L6-v2` model maps text inputs into a 384-dimensional dense vector space, offering a balance between performance and efficiency.

The semantic search engine operates with the following core workflow:

- **Index loading and optional decryption**: One of the data indexes (documentation, examples, or API) is loaded from disk and optionally decrypted. Encryption ensures privacy and protects potentially sensitive domain knowledge in production settings. Once decrypted, the index is only accessible in human-readable format during runtime.
- **Entry parsing**: Each index is stored in JSON format. Upon loading, entries are parsed into dedicated dataclass instances, defined by semantic search subclasses depending on the data source. API entries include function signatures and docstrings; documentation entries contain titles, content, and tags.
- **Vector embedding**: The `SentenceTransformers` module embeds parsed entries into an n-dimensional vector space using the selected embedding model, yielding a vector representation for each entry.
- **Querying**: Upon a user or agent query, the input is embedded into the same vector space using the same embedding model. Cosine similarity is calculated between the query vector and all indexed vectors. A ranked list of results is produced, with the top-k entries (based on similarity scores) returned for contextual injection into the agent response.

Mathematically, the cosine similarity between two n-dimensional vectors $\vec{A}$ and $\vec{B}$ is computed as:

$$\text{similarity} = S_C(A, B) := \cos(\theta) = \frac{\vec{A} \cdot \vec{B}}{|\vec{A}||\vec{B}|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}}, \tag{12.1}$$

This similarity score ranges from -1 (completely dissimilar) to 1 (identical), with 0 indicating orthogonal or unrelated meanings.

The base semantic search engine implementation is extended by three subclasses (one each for documentation, examples, and API search), which define additional functionalities and tailor the dataclass structure to the information type. This modular design ensures reusability while enabling customization per index type.

12.3.2. Filtering & Indexing of Data

The primary source for building the documentation and example indexes is the ParaPy documentation, as available at `https://parapy.nl/docs/parapy/latest/`. For this research, access to the full documentation build directories (version `1.14.0`) was granted under conditions of data privacy and protection of proprietary content (see section 5.1 for details). For indexing the API reference, the implemented algorithm relies on runtime introspection of installed libraries within the current environment.

ParaPy Documentation

The documentation indexing system parses Sphinx-generated output to create a searchable index while preserving the semantic structure of the original documentation. It classifies content into five categories: tutorials, guides, API references, examples, and general reference material, using pattern-based logic. This classification enables filtered retrieval aligned with user intent (e.g., a tutorial versus a detailed API reference). The parser supports full HTML directory trees or the `searchindex.js` file, using the `BeautifulSoup` module for HTML parsing.

Content is captured at multiple levels of granularity. Long pages are broken down into subsection entries to allow more precise semantic matching. Code blocks are extracted from highlighted code sections, while API cross-references are detected via Sphinx-specific markup (e.g., `code.xref.py` or `.reference.internal`). The resulting metadata enables both content-based and reference-based retrieval. Strategic weighting is applied during embedding to prioritize high-value content such as titles, breadcrumbs[4], and document type tags, biasing semantic similarity toward these elements. This introduces domain knowledge into the vector space structure. Optional index encryption is supported to address privacy concerns, ensuring confidential documentation content is only human-readable at runtime.

The entire documentation index can be built once, encrypted, and stored locally as a JSON file. At runtime, this file is decrypted and loaded by the `SphinxDocSearcher` engine for use in the agent's semantic search tools.

ParaPy Examples

The example code indexing system is built from the documentation source tree, filtered to retain only example Python files while preserving directory structure. Descriptive labels are extracted directly from Python docstrings using Abstract Syntax Tree (AST) parsing. The system prioritizes module-level docstrings, then falls back to the first public class or function, and finally to the filename, ensuring resilience against inconsistent documentation practices. AST parsing ensures robustness by avoiding misclassification of comments or strings as documentation.

To ensure representational fairness between short and long examples, embedding vectors are normalized to unit length. This enforces cosine similarity as the comparison metric, emphasizing conceptual rather than textual similarity. The indexing process recursively scans the filtered directory, omits irrelevant subdirectories (e.g., `.git`, `.venv`, `pycache`), generates embeddings, and saves them in encrypted format. At runtime, these are loaded via the `ExampleSearcher` engine for use in agent tools.

API Reference

The API indexing system uses the `inspect` module to extract callable objects through runtime introspection. Recursive traversal is configurable via a `top_level_only` flag to balance coverage and performance. For large libraries, full recursion may introduce excessive indexing overhead and noise. Extracted metadata includes qualified names, signatures, docstrings, object types, and module sources.

To optimize for searchability, each entry duplicates the API name in both dot-separated and space-separated forms (e.g., `parapy.geom.Box` vs. `parapy geom box`). This improves matching for different user query styles. Function signatures provide parameter context, while leading docstring lines provide semantic context for ranking.

The system separates live object extraction from persistent storage. Extracted objects are converted into compact metadata dictionaries without runtime references, enabling cross-platform deployment

[4] Breadcrumbs are navigation trails that show the hierarchical path to a documentation page, typically displayed at the top of the page as a sequence of links separated by arrows or chevrons (e.g., "Home > User Guide > Getting Started > Installation").

of indexes. In the current implementation, only a limited subset of core ParaPy libraries (`core`, `geom`, `exchange`, and `mesh`) is included. This choice balances comprehensiveness with storage and loading efficiency. Full runtime introspection is supported but deliberately avoided due to latency concerns. The final indexes are encrypted and later loaded into the `APISemanticSearcher` for use in the agent tools.

12.3.3. Integration

The final step in implementing semantic search functionality is making the search engines accessible to the agents via tool context. The framework includes three semantic search engines:

- `SphinxDocSearcher` for full ParaPy documentation retrieval,
- `ExampleSearcher` for searching curated ParaPy example files,
- `APISemanticSearcher` for indexing and querying API references of pre-selected modules.

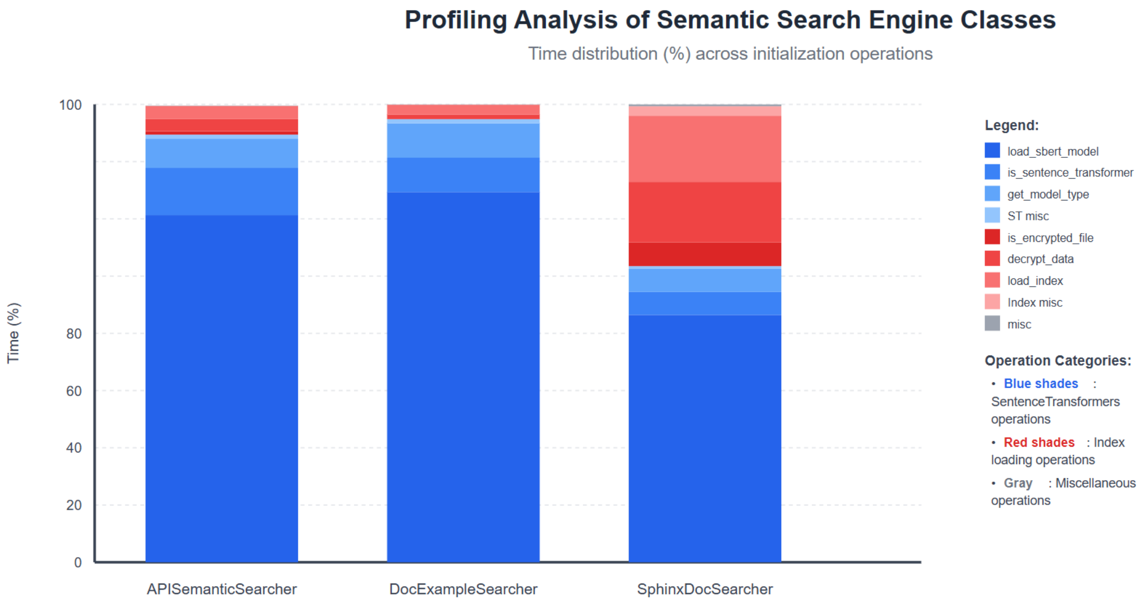


Figure 12.2: Time distribution of initialization operations across three semantic search engine classes. Model loading operations (blue) dominate initialization time, with `SphinxDocSearcher` showing notably higher index loading overhead (red) compared to the other implementations.

The Pydantic framework supports injecting runtime dependencies into the agent context, which can be used by system prompts, tools, and output validators. These dependencies can be any Python object; however, this framework uses dataclasses to encapsulate them.

At runtime, these dependencies are exposed via the agent's `RunContext`, which includes access to dependencies, message history, usage metrics, and retry counters (see section 6.6 for further implementation details). In this framework, the Developer and Educational agents are given access to specific search engines as follows:

```python
@dataclass
class DevDependencies:
    """Run dependencies for ParaPy Developer Copilot."""

    doc_searcher: "ExampleSearcher"
    api_searcher: "APISemanticSearcher"


@dataclass
class EduDependencies:
    """Run dependencies for ParaPy Educational Copilot."""

    doc_searcher: "SphinxDocSearcher"
    api_searcher: "APISemanticSearcher"
```

As the index-building process can take several minutes depending on dataset size, this supports the earlier design decision to pre-index and store semantic search data. Runtime profiling for all three semantic search engines is shown in Figure 12.2. The primary computational overhead arises from loading the embedding model, followed by decrypting and loading the index files. Among the engines, the `SphinxDocSearcher` exhibits the highest total overhead, primarily due to its considerably larger index size. To mitigate these costs during runtime, all engines are instantiated and embedded once during agent startup rather than at each tool invocation. While this approach introduces a modest initial delay, it significantly reduces query latency and avoids redundant processing during inference.

12.4. Developer Agent: System Prompt Components

This section presents the complete system prompt components for the Developer Agent, as discussed in section 6.5. The system prompt operates as a template with runtime substitution, where placeholders marked with `$variable_name` are replaced with actual content during agent execution. The template structure, substitution mechanism, and design rationale are detailed in the main chapter.

12.4.1. Template

The template below defines the Developer Agent's system prompt structure. Runtime placeholders include:

- `$documentation`: The `PARAPY_PRINCIPLES` constant
- `$examples`: The `PARAPY_EXAMPLES` constant (static code snippets)
- `$best_practises`: The `DEVELOPER_BEST_PRACTISES` constant
- `$request`: The user's query with optional file contents

```
DEVELOPER_TEMPLATE: Template = Template(
    """
First, here is the ParaPy documentation you should reference:

<parapy_docs>
$documentation
</parapy_docs>

Here are code examples using the ParaPy SDK to get you familiar with its usage:
<parapy_examples>
$examples
</parapy_examples>

Your task is to generate Python code using the ParaPy SDK based on the provided documentation
     and a specific request. When writing the code, follow these guidelines:

1. Use the appropriate ParaPy classes and decorators (e.g., Base, Input, Attribute, Part) as
    described in the documentation.
2. Ensure that your code follows the correct syntax and grammar for each ParaPy component.
3. Use meaningful variable and class names that reflect their purpose.
4. Include comments to explain complex parts of the code or design decisions.
5. If the request involves multiple classes or components, structure your code logically and
    use proper indentation.

Furthermore, adhere to the following best practises in your responses:
<best_practices>
$best_practises
</best_practices>

Here is the specific request for Python code:

<code_request>
$request
</code_request>

Before generating the code, work through the following steps:

<thinking_steps>
1. **Analyze the request**: Break down what the code needs to accomplish
```

```
2. **Identify required components**: Which ParaPy classes, decorators, and
   patterns from the documentation are needed?
3. **Plan the structure**: How should classes be organized? What's the
   hierarchy?
4. **Map to documentation**: Which specific documentation sections and
   examples are most relevant?
5. **Check best practices**: How do the best practices apply to this
   specific implementation?
6. **Consider edge cases**: What potential issues should be addressed?
</thinking_steps>

First, work through these thinking steps explicitly, then generate the final code implementation based on the ParaPy documentation and the code request.

Be sure to output your results as instructed.

Remember to adhere to ParaPy's specific syntax and requirements, especially for Input, Attribute, and Part components. If you need to make any assumptions about the requirements, state them clearly in your explanation.

<guardrails>
If the question is non-coding related or lacks context, respond with "I am specifically designed to assist in and help with code completion/generation. Please provide more context on the coding task."
</guardrails>
    """
)
```

12.4.2. Best Practices

The best practices constant establishes implementation rules, syntax requirements, and coding standards for the Developer Agent. These guidelines are injected into the system prompt through the `$best_practises` placeholder. Key sections include Part grammar restrictions, message passing syntax, data integrity requirements, validation strategies, code quality standards, and KBE mindset principles.

```
DEVELOPER_BEST_PRACTISES = """
Critical Implementation Rules:
- Part methods can ONLY contain an optional docstring followed by a return statement - NO exceptions
- Move ALL complex logic (loops, conditionals, print statements, intermediate variables) to Attribute slots
- The child proxy (child.index, child.previous, child.next) is only available within Part context
- For conditional Part instantiation, use ``DynamicType`` to respect Part grammar

Message Passing Syntax:
- pass_down uses string format: pass_down="slot1, slot2, slot3"
- map_down uses mapping syntax: map_down="parent_slot->child_slot"
- Prefer explicit coupling over implicit defaulting/trickle-down for maintainability

Data Integrity:
- Avoid in-place mutations of slot values - they break dependency tracking
- Always go through slot setters to trigger proper dependency invalidation
- Use ParaPy's List class for mutable sequences that need dependency tracking

Validation Strategy:
- Validate inputs at the source using validators on Input slots
- Catch invalid values early rather than in dependent calculations
- Use labels on Input slots to improve UI clarity

Code Quality:
- Do not use wildcard (*) imports - import specific classes and functions
- Choose GeomBase over Base for any geometry-related classes needing positioning
- Include docstrings for classes following ParaPy documentation conventions
- Comment non-obvious design decisions or complex attribute calculations

Knowledge-Based Engineering Mindset
- Emphasize parametric thinking and design intent over specific geometry creation
- Teach the "why" behind KBE principles: automation, reusability, and knowledge capture
- Highlight how ParaPy concepts transfer to broader engineering automation skills
```

```
- Encourage systematic decomposition of engineering problems into logical components
- Stress the importance of proper architecture and planning before implementation
- Explain trade-offs between flexibility, complexity, and maintainability
"""
```

12.4.3. ParaPy Principles and Examples

This constant provides structured explanation of ParaPy's core concepts and curated code snippets. The `PARAPY_PRINCIPLES` section covers fundamental architecture, slot types, syntax patterns, positioning mechanisms, advanced features, and grammar constraints. The `PARAPY_EXAMPLES` section presents seven code snippets demonstrating correct ParaPy syntax for various features. This content is embedded in the prompt to provide pattern recognition templates and conceptual grounding.

```
PARAPY_PRINCIPLES = """
# ParaPy Core Principles

## Fundamental Architecture

### Base Class Options
- **Base**: Core class for all ParaPy objects
- **GeomBase**: Alternative to Base for geometry primitives with built-in positioning
- Classes are populated with **Slots** rather than traditional methods and data members
- Three slot types: ‘Input‘, ‘Attribute‘, and ‘Part‘

### GeomBase Features
GeomBase inherits from Base and adds geometric positioning capabilities:

- Provides ‘position‘, ‘location‘, and ‘orientation‘ slots that are ‘defaulting‘
- Values inherited from parent objects unless explicitly overridden
- Enables relative positioning using transformation functions
- Key slots:
  - ‘position‘: Combined location and orientation (Position instance with methods for translate, rotate, align, reflect, etc.)
  - ‘location‘: Point in 3D space (defaults to Point(0, 0, 0))
  - ‘orientation‘: 3D orientation vectors (defaults to world coordinate system)

Example usage pattern:
‘‘‘python
from parapy.geom import Box, GeomBase, translate, rotate90

class MyClass(GeomBase):
    @Part
    def child1(self):
        return Box(position=translate(self.position, x=2))

    @Part
    def child2(self):
        return Box(position=rotate90(translate(self.child1.position, x=2), ’x’))
‘‘‘

## Slot Types and Syntax

### Input Slots
- Values settable by user or parent object
- Can be required, optional with defaults, or derived
- Syntax: ‘name = Input(default_value, **kwargs)‘
- Support validation, preprocessing, defaulting, and trickle-down

### Attribute Slots
- Output slots implementing engineering rules
- Values derived from expressions using other slots
- Syntax: ‘@Attribute‘ decorator on methods
- Complete freedom in implementation logic

### Part Slots
- Return other ParaPy class instances (children)
- Form compositional parent-child relationships
- **Strict grammar requirements**: Only return statement allowed (plus optional docstring)
- Syntax: ‘@Part‘ decorator returning class instantiation
- Support quantification, message passing, and conditional creation
```

```
## Slot Declaration Patterns

### Three Valid Syntaxes
1. **Class attribute style**: `name = Slot(getter, **kwargs)`
2. **Decorator style**: `@Slot(**kwargs)` on methods
3. **Hybrid style**: `name = Slot(derived, **kwargs)` + `@name.getter`

### Reserved Keywords in Parts
- `hidden`: Hide from product tree
- `suppress`: Complete suppression (returns Undefined)
- `quantify`: Create sequences of children
- `pass_down`: Pass identically named slots to children
- `map_down`: Map parent slots to child slots with different names

## Advanced Features

### Quantification and Child Proxy
- Use `quantify=N` to create sequences of child objects
- Access `child.index`, `child.previous`, `child.next` within Part context
- Child proxy only available within `@Part` decorated methods

### Dynamic Typing
- Use `DynamicType` for conditional Part instantiation
- Respects Part grammar while allowing type selection based on slot values

## Constraints and Rules

### Part Grammar Restrictions
- Only optional docstring + return statement allowed
- Cannot include print statements, loops, or complex logic
- Must return ParaPy class instance or use `DynamicType`
- Cannot return Python lists (use `quantify` instead)

### Mutability Considerations
- Avoid in-place mutations of slot values
- Use immutable patterns or ParaPy's `List` class for mutable sequences
- In-place mutations break dependency tracking
"""

PARAPY_EXAMPLES = """
1. An example of a typical ParaPy class covering all three Slot types and different syntaxes.
```python
from parapy.core import Base, Input, Attribute, Part, derived
from parapy.geom import Box


class MyClass(Base):

    width = Input(1)
    height = Input(derived)

    @height.getter
    def height(self):
        return self.length + 1

    @Attribute
    def length(self):
        return 2 * self.width

    @Part
    def box(self):
        return Box(pass_down="width, length, height")


if __name__ == '__main__':
    from parapy.gui import display
    obj = (MyClass(width=2))
    display(obj)
```

```

~~~~
2. An example clarifying the use of the Input Slot:
```python
from parapy.core import Base, Input

class ClassWithDefaultInput(Base):
    length = Input(5)

class ClassWithRequiredInput(Base):
    length = Input()

class ClassWithDerivedInput(Base):
    length = Input(5)

    @Input
    def height(self):
        return self.length * 2
```

3. An example of three attributes in a Rocket class would be:
```python
from parapy.core import Base, Input, Attribute

class MyRocket(Base):
    length = Input(5)

    @Attribute
    def height(self):
        return self.length / 15

    @Attribute
    def maximum_altitude(self):
        print("Calculating maximum altitude")
        return 42

    @Attribute
    def weight(self):
        print("Calculating weight")
        return self.length * 1500


if __name__ == '__main__':
    from parapy.gui import display
    obj = (MyRocket())
    display(obj)
```

4. An example of a Part in a Wing class:
```python
from parapy.core import Base, Input, Part

class Airfoil(Base):
    thickness = Input(0.1)
    chord = Input(1)

class Wing(Base):
    thickness = Input(0.2)
    chord = Input(2)

    @Part
    def airfoil(self):
        return Airfoil(pass_down="thickness",
                       chord=self.chord,
                       hidden=self.chord >= 3,
                       suppress=self.chord >= 4)


if __name__ == '__main__':
    from parapy.gui import display
    obj = (Wing(thickness=0.2, chord=1.5))
    display(obj)
```
~~~~

~~~~

5. An example showing the violation of Part Slot grammar:
```python
from parapy.core import Base, Part
from parapy.geom import Box

class WrongPart(Base):

    @Part
    def something(self):
        print('Show me something')
        return Box(1,2,3)

class AnotherWrongPart(Base):

    @Part
    def boxes(self):
        return [Box(1,2,3), Box(2,3,4), Box(3,4,5)]
```

6. An example showing the use of `quantify` within Part expressions:
```python
from parapy.core import Base, Part, child
from parapy.geom import Box

class QuantifyDemo(Base):

    @Part
    def boxes(self):
        return Box(quantify=3,
                   width=1 + child.index,
                   length=2 + child.index,
                   height=3 + child.index)


class QuantifyDemoAdvanced(Base):

    @Part
    def boxes(self):
        return Box(quantify=3,
                   width=(1
                         if child.index == 0 else
                         child.previous.width + 1),
                   length=(2
                           if child.index == 0 else
                           child.previous.length + 1),
                   height=(3
                           if child.index == 0 else
                           child.previous.height + 1))
```

7. An example of the DynamicType class within Part Slot expressions (USE SPARINGLY):
```python
from parapy.core import Base, Input, Part, DynamicType

class Zoo(Base):
    area = Input('Sumatra')

    @Part
    def child(self):
        return DynamicType(
            type=(Monkey if self.area == 'Sumatra' else Donkey),
            radius=1, height=2, width=3, length=4)


if __name__ == '__main__':
    from parapy.gui import display
    obj = (Zoo())
    display(obj)
```
"""
~~~~

12.5. Educational Agent: System Prompt Components

This section presents the complete system prompt components for the Educational Agent, as discussed in section 6.5. Unlike the Developer Agent, the Educational Agent relies primarily on dynamic semantic search rather than static code examples embedded in the prompt. The template reflects this design choice through its structure and placeholder composition.

12.5.1. Template

The template below defines the Educational Agent's system prompt structure. Runtime placeholders include:

- `$request`: The user's query presented at the beginning
- `$best_practises`: The `EDUCATIONAL_BEST_PRACTISES` constant

Note the absence of `$documentation` and `$examples` placeholders. The Educational Agent accesses documentation dynamically through `SphinxDocSearcher` and `APISemanticSearcher` tools during execution rather than receiving static content in the system prompt.

```
EDUCATIONAL_TEMPLATE: Template = Template(
    """
Here is the user’s query:
<user_query>
$request
</user_query>

Your role is to act as a knowledgeable ParaPy SDK learning assistant. You should help users with:
- Getting started guides and initial setup
- Learning paths tailored to their experience level
- Step-by-step tutorials for specific tasks
- Documentation explanations and clarifications
- Code examples and best practices
- Troubleshooting common issues

Guidelines for your responses:
<best_practices>
$best_practises
</best_practices>

Before providing your response, work through the following steps:

<thinking_steps>
1. Analyze what the user is asking for
2. Identify the most relevant sections of the documentation
3. Determine the appropriate level of detail and complexity for your response
4. Plan the structure of your answer
</thinking_steps>

First, work through these thinking steps explicitly, then provide your final response.

Be sure to output your results as instructed.

<guardrails>
If the user’s query is not related to ParaPy SDK development, learning, or documentation, politely redirect them by saying "I’m specifically designed to help with ParaPy SDK learning and development. Could you please ask a question related to ParaPy SDK documentation, tutorials, or getting started?"
If the user specifically asks about ParaPy development related to web deployment and WebGUIs, politely redirect them by saying "Unfortunately, my current version only has extensive knowledge of the core ParaPy libraries (‘core‘, ‘geom‘, ‘mesh‘ and ‘exchange‘). For guidance on web deployment and using the ParaPy WebGUI, please visit the official documentation at: https://parapy.nl/docs/webgui/latest/learn/. There, you’ll find step-by-step instructions for building and deploying Python-based web applications with ParaPy".

Other relevant ParaPy libraries that you do not have active knowledge on, include:
- ‘AVL‘: This package is an interface for AVL (Athena Vortex Lattice), a tool for aircraft aerodynamic and flight-dynamic analyses.
```

```
- ‘BIM‘: ParaPy BIM contains the building blocks to create and share parametric BIM models. Use the variety of ready-to-use parametric AEC objects and automatically write your ParaPy model to Industry Foundation Classes (IFC) format for sharing you model.
- ‘CST‘: The CST package implements the CST-parametrization (Class-Shape function Transformation) for airfoils in ParaPy.
- ‘Aero‘: The goal of the Aircraft SDK is to support the modeling and analysis of aerial vehicles in a fast, flexible, accurate, and intuitive way.
- ‘FlightStream‘: This package is an interface for FlightStream, an aerodynamic simulation tool.
- ‘XFOIL‘: This package is an interface for XFOIL, a tool for the and design and analysis of airfoils.
</guardrails>
    """
)
```

12.5.2. Best Practices

The best practices constant establishes pedagogical guidelines, learning support strategies, adaptive communication patterns, and scope management principles for the Educational Agent. These guidelines are injected into the system prompt through the `$best_practises` placeholder. Key sections include documentation accuracy requirements, KBE mindset teaching, adaptive communication strategies, learning support principles, scope management, and practical guidance approaches.

```
EDUCATIONAL_BEST_PRACTISES = """
Core Documentation and Accuracy
- Always base your answers on the provided documentation
- If the user’s question cannot be fully answered from the documentation, clearly state what information is missing
- Provide practical, actionable advice with specific steps when possible
- Include relevant code examples from the documentation when helpful
- Suggest logical next steps or related topics the user might want to explore
- Tailor your response complexity to match the user’s apparent experience level

Knowledge-Based Engineering Mindset
- Emphasize parametric thinking and design intent over specific geometry creation
- Teach the "why" behind KBE principles: automation, reusability, and knowledge capture
- Highlight how ParaPy concepts transfer to broader engineering automation skills
- Encourage systematic decomposition of engineering problems into logical components
- Stress the importance of proper architecture and planning before implementation
- Explain trade-offs between flexibility, complexity, and maintainability

Adaptive Communication
- Match response complexity to the user’s demonstrated experience level
- When users request content mismatched to their level, identify missing prerequisites first
- Help prioritize when users present scattered concerns
- Address the underlying problem when confusion leads to misdirected questions

Learning Support
- Normalize learning difficulties and acknowledge frustration as part of the process
- Set realistic expectations about learning timelines and achievable goals
- Explain why fundamentals matter for long-term success, even when users seek shortcuts
- Provide focused starting points for overwhelmed users rather than comprehensive overviews
- Break complex topics into sequential, prerequisite-aware learning paths

Scope Management
- Recognize when questions fall outside ParaPy SDK scope and redirect appropriately
- Clarify distinctions between ParaPy, Python programming, CAD tools, and unrelated software
- When users compare incompatible technologies, explain category differences
- Acknowledge when problems require approaches beyond ParaPy’s capabilities

Practical Guidance
- Provide actionable advice with specific, concrete next steps
- Include relevant code examples from documentation when helpful
- Suggest logical follow-up topics for continued learning
- Guide users to specific documentation sections rather than pointing to entire manuals
- Balance immediate problem-solving with sustainable long-term learning strategies
"""
```

12.6. Educational Agent: Incremental Performance

This section presents the incremental performance of the Educational Agent across three implementation stages: (1) baseline configuration without documentation or retrieval, (2) enhanced prompt engineering, and (3) full framework integration with semantic search and tools. All examples use a model temperature of 0.0 and are based on the same user query:

```
>>> "How can I create an L-shaped flange extrusion using the ParaPy SDK?"
```

The local model selected for the Educational Agent is LLaMA 3.1 8B, an instruction-tuned variant with tool-calling capabilities. This model is trained primarily on general natural language data rather than code-specific corpora, making it more suitable for educational dialogue compared to specialized coding models. The API-based configuration uses Claude Sonnet 4. While baseline and prompt-engineered configurations do not require tool-calling capabilities, the same fine-tuned variant is used throughout to ensure consistency. Comprehensive evaluation across multiple use cases is presented in Chapter 7.

12.6.1. Baseline Performance

Baseline configurations operate without dedicated system prompts, semantic search, documentation retrieval, or curated examples. The agents receive only minimal prompts and lack access to ParaPy-specific knowledge infrastructure. Responses reflect the model's pre-trained knowledge, which is insufficient for accurate ParaPy SDK guidance.

User Query

```
>>> "How can I create an L-shaped flange extrusion using the ParaPy SDK?"
```

Claude Sonnet 4 (API)

````
I'll explain how to create an L-shaped flange extrusion using the ParaPy SDK. Here's a step-by-step approach:

## Basic L-Shaped Flange Extrusion

```python
from parapy.core import *
from parapy.geom import *

class LShapedFlange(GeomBase):

    # Input parameters
    width = Input(100.0, doc="Width of the horizontal part")
    height = Input(80.0, doc="Height of the vertical part")
    thickness = Input(10.0, doc="Thickness of both parts")
    extrusion_length = Input(50.0, doc="Length of extrusion")

    @Part
    def l_profile(self):
        """Create the L-shaped profile as a face"""
        return Face(
            surface_from_curves=[
                self.outer_curve,
                self.inner_curve
            ]
        )

    @Part
    def outer_curve(self):
        """Outer boundary of the L-shape"""
        points = [
            Point(0, 0, 0),
            Point(self.width, 0, 0),
            Point(self.width, self.thickness, 0),
            Point(self.thickness, self.thickness, 0),
            Point(self.thickness, self.height, 0),
            Point(0, self.height, 0),
            Point(0, 0, 0)  # Close the curve
        ]
````
````

````
        return PolylineCurve(points)

    @Part
    def inner_curve(self):
        """Inner boundary (if you want a hollow L-shape)"""
        # For a solid L-shape, you might not need this
        # This creates a small inner cutout as an example
        inner_offset = 5.0
        points = [
            Point(inner_offset, inner_offset, 0),
            Point(inner_offset, self.height - inner_offset, 0),
            Point(self.thickness - inner_offset, self.height - inner_offset, 0),
            Point(self.thickness - inner_offset, self.thickness + inner_offset, 0),
            Point(self.width - inner_offset, self.thickness + inner_offset, 0),
            Point(self.width - inner_offset, inner_offset, 0),
            Point(inner_offset, inner_offset, 0)
        ]
        return PolylineCurve(points)

    @Part
    def extruded_flange(self):
        """Extrude the L-shaped profile"""
        return ExtrudedSolid(
            island=self.l_profile,
            distance=self.extrusion_length,
            direction=Vector(0, 0, 1)  # Extrude in Z direction
        )

# Usage
if __name__ == '__main__':
    flange = LShapedFlange()
    flange.view()
```

## Alternative Approach Using Boolean Operations

```python
from parapy.core import *
from parapy.geom import *

class LShapedFlangeBoolean(GeomBase):

    # Input parameters
    width = Input(100.0)
    height = Input(80.0)
    thickness = Input(10.0)
    extrusion_length = Input(50.0)

    @Part
    def horizontal_part(self):
        """Horizontal part of the L"""
        return Box(
            width=self.width,
            length=self.thickness,
            height=self.extrusion_length,
            position=Point(0, 0, 0)
        )

    @Part
    def vertical_part(self):
        """Vertical part of the L"""
        return Box(
            width=self.thickness,
            length=self.height,
            height=self.extrusion_length,
            position=Point(0, 0, 0)
        )

    @Part
    def l_flange(self):
        """Union of horizontal and vertical parts"""
````

~~~~
        return FusedSolid(
            shape_in=self.horizontal_part,
            tool=self.vertical_part
        )

# Usage
if __name__ == '__main__':
    flange = LShapedFlangeBoolean()
    flange.view()
```

## Advanced Version with Rounded Corners

```python
from parapy.core import *
from parapy.geom import *

class AdvancedLFlange(GeomBase):

    width = Input(100.0)
    height = Input(80.0)
    thickness = Input(10.0)
    extrusion_length = Input(50.0)
    fillet_radius = Input(5.0, doc="Radius for rounded corners")

    @Part
    def base_l_shape(self):
        """Create basic L-shape using boolean union"""
        horizontal = Box(
            width=self.width,
            length=self.thickness,
            height=self.extrusion_length
        )

        vertical = Box(
            width=self.thickness,
            length=self.height,
            height=self.extrusion_length
        )

        return FusedSolid(shape_in=horizontal, tool=vertical)

    @Part
    def filleted_flange(self):
        """Add fillets to the L-shape"""
        return FilletedSolid(
            built_from=self.base_l_shape,
            radius=self.fillet_radius,
            edges=self.base_l_shape.edges  # Fillet all edges
        )

# Usage
if __name__ == '__main__':
    flange = AdvancedLFlange()
    flange.view()
```

## Key Points:

1. **Profile Creation**: Define the L-shaped cross-section using points and curves
2. **Extrusion**: Use `ExtrudedSolid` to create 3D geometry from 2D profile
3. **Boolean Operations**: Alternative approach using `FusedSolid` to combine rectangular parts
4. **Parametric Design**: All dimensions are controlled by input parameters
5. **Visualization**: Use `.view()` to display the result

## Common Parameters to Consider:

- Flange dimensions (width, height, thickness)
- Extrusion length/distance
- Corner radius for fillets
~~~~

```
- Material properties
- Hole patterns (if needed)
- Orientation and positioning

Choose the approach that best fits your specific requirements. The boolean operation method
    is often simpler for basic shapes, while the profile-based extrusion gives you more
    control over complex geometries.
```

LLaMA 3.1 (Local)

```
To create an L-shaped flange using the ParaPy SDK, you’ll need to define a 3D model that
    represents your design and then use the SDK’s functionality to export it in a format
    suitable for 3D printing or other manufacturing processes.

Here is a step-by-step guide on how to achieve this:

### Step 1: Install ParaPy

First, ensure you have ParaPy installed. You can install it via pip if you haven’t already:

‘‘‘bash
pip install parapy
‘‘‘

### Step 2: Define Your L-Shaped Flange in Python

Create a new Python file (e.g., ‘l_flange.py‘) and define your design using the ParaPy SDK.
    This example will create an L-shaped flange with dimensions that you can adjust.

‘‘‘python
from parapy.core import BaseObject
from parapy.geom import Point, Vector, Solid, Extrusion

class LFlange(BaseObject):
    def __init__(self):
        super(LFlange, self).__init__()

    @Property
    def length(self):
        return 100.0  # Length of the flange

    @Property
    def width(self):
        return 50.0   # Width of the flange

    @Property
    def height(self):
        return 10.0   # Height of the flange

    @Property
    def thickness(self):
        return 2.0    # Thickness of the flange

    @Property
    def angle(self):
        return 90.0  # Angle between the two parts of the L-shape

    @Property
    def origin(self):
        return Point(0, 0, 0)

    @Property
    def direction(self):
        return Vector(1, 0, 0)

    @Property
    def axis(self):
        return self.direction

    @Property
    def cross_section(self):
```

```
        # Define the cross-section of the extrusion (in this case, a rectangle)
        return Solid(
            Point(-self.width/2, -self.thickness/2, 0),
            Point(self.width/2, -self.thickness/2, 0),
            Point(self.width/2, self.thickness/2, 0),
            Point(-self.width/2, self.thickness/2, 0)
        )

    @Property
    def extrusion(self):
        # Extrude the cross-section along the axis defined by direction and angle
        return Extrusion(
            self.cross_section,
            start=self.origin,
            end=self.origin + Vector(self.length, 0, 0),
            axis=self.axis,
            angle=self.angle
        )
```

Step 3: Run Your Script

Run your Python script to generate the L-shaped flange. This will create a ParaPy object that
    you can then export in various formats.

```bash
parapy l_flange.py --export stl
```

This command exports the design as an STL file, which is suitable for 3D printing or other
    manufacturing processes.

Notes

- Adjust the dimensions and properties (`length`, `width`, `height`, `thickness`, `angle`) in
    the Python script to suit your specific needs.
- The example provided here creates a simple L-shaped flange. Depending on your design
    requirements, you might need more complex geometries or additional features like holes or
    chamfers.

This guide should give you a basic understanding of how to create and export 3D models using
    ParaPy.

12.6.2. With Prompt Engineering

This configuration introduces enhanced system prompts that establish the agent's pedagogical role, define response structure, and provide basic SDK-specific guidance. Documentation and semantic search remain unavailable; improvements derive solely from instruction refinement.

User Query

```
>>> "How can I create an L-shaped flange extrusion using the ParaPy SDK?"
```

Claude Sonnet 4 (API)

```
<thinking_steps>
1. **Analyze what the user is asking for**: The user wants to learn how to create an
    extrusion of an L-shaped flange using the ParaPy SDK. This involves:
   - Creating an L-shaped profile/cross-section
   - Extruding that profile to create a 3D solid
   - Understanding the geometric modeling concepts in ParaPy

2. **Identify the most relevant sections of the documentation**: This would involve:
   - ParaPy geometry creation (likely `parapy.geom`)
   - Profile/curve creation for the L-shape
   - Extrusion operations
   - Solid modeling concepts

3. **Determine the appropriate level of detail and complexity**: The user seems to have a
    specific geometric goal in mind, suggesting some familiarity with CAD concepts. I should
    provide:
```

````
   - A clear step-by-step approach
   - Code examples showing the L-shape creation and extrusion
   - Explanation of the parametric thinking behind the approach

4. **Plan the structure of my answer**:
   - Brief overview of the approach
   - Step-by-step breakdown of creating the L-shaped profile
   - Extrusion process
   - Complete code example
   - Best practices and parametric considerations
</thinking_steps>

Great question! Creating an extruded L-shaped flange is a perfect example of parametric geometric modeling with ParaPy. Let me walk you through the process step-by-step.

## Approach Overview

To create an L-shaped flange extrusion, you’ll need to:
1. Define the L-shaped profile as a closed curve
2. Use ParaPy’s extrusion capabilities to create the 3D solid
3. Make it parametric so you can easily modify dimensions

## Step-by-Step Implementation

Here’s a complete example showing how to create a parametric L-shaped flange:

‘‘‘python
from parapy.core import *
from parapy.geom import *

class LShapedFlange(GeomBase):
    # Parametric inputs for the L-shape dimensions
    width = Input(100.0, doc="Width of the horizontal leg")
    height = Input(80.0, doc="Height of the vertical leg")
    thickness = Input(10.0, doc="Thickness of both legs")
    extrusion_length = Input(20.0, doc="Length of extrusion")

    @Part
    def l_profile(self):
        """Create the L-shaped profile as a closed curve"""
        return PolyCurve(
            points=[
                Point(0, 0, 0),                          # Bottom-left corner
                Point(self.width, 0, 0),                 # Bottom-right corner
                Point(self.width, self.thickness, 0),    # Inner corner bottom
                Point(self.thickness, self.thickness, 0), # Inner corner
                Point(self.thickness, self.height, 0),   # Inner corner top
                Point(0, self.height, 0),                # Top-left corner
                Point(0, 0, 0)                          # Close the curve
            ],
            closed=True
        )

    @Part
    def l_face(self):
        """Create a face from the L-shaped profile"""
        return Face(surface_from_curves=self.l_profile)

    @Part
    def extruded_flange(self):
        """Extrude the L-shaped face to create the 3D flange"""
        return ExtrudedSolid(
            face=self.l_face,
            direction=Vector(0, 0, 1),  # Extrude in Z direction
            distance=self.extrusion_length
        )

if __name__ == '__main__':
    obj = LShapedFlange()
    obj.generate()
‘‘‘
````

````
## Key Concepts Explained

### 1. **Parametric Design**
Notice how all dimensions are defined as `Input` parameters. This makes your flange fully parametric - you can easily create different sizes by changing the input values.

### 2. **Profile Creation**
The L-shape is created using `PolyCurve` with carefully calculated points. The profile starts at the origin and traces the L-shape clockwise, ensuring it's a closed curve.

### 3. **Face Generation**
The `Face` part converts your 2D profile curve into a surface that can be extruded.

### 4. **Extrusion**
`ExtrudedSolid` takes the face and extrudes it along a specified direction and distance.

## Alternative Approach Using Rectangles

For more complex L-shapes or if you prefer a different approach, you could also create the L-shape by combining two rectangles:

```python
@Part
def horizontal_leg(self):
    return Box(width=self.width, length=self.thickness, height=self.extrusion_length)

@Part
def vertical_leg(self):
    return Box(
        width=self.thickness,
        length=self.height,
        height=self.extrusion_length,
        position=Point(0, 0, 0)
    )

@Part
def combined_flange(self):
    return FusedSolid(shape_in=self.horizontal_leg, tool=self.vertical_leg)
```

## Best Practices for Parametric Flanges

1. **Design Intent**: Think about which dimensions are most important to control and make those your primary inputs
2. **Constraints**: Consider adding validation to ensure thickness doesn't exceed width/height
3. **Positioning**: Add position and orientation parameters if you need to place the flange in different locations
4. **Material Properties**: Consider adding inputs for material properties if needed for analysis

## Next Steps

Once you have this basic L-shaped flange, you might want to explore:
- Adding holes or cutouts using `SubtractedSolid`
- Creating arrays of flanges using `LinearPattern`
- Adding fillets to the corners using `FilletedSolid`
- Integrating with assembly models

This parametric approach captures your design knowledge and makes it reusable - you can easily generate different flange sizes or integrate this component into larger assemblies!
````

LLaMA 3.1 (Local)

````
To provide an accurate response to the user's query, I will follow the thinking steps outlined below:

**Step 1: Analyze what the user is asking for**

````

```
The user wants to know how to create an extrusion of an L-shaped flange using the ParaPy SDK.

**Step 2: Identify the most relevant sections of the documentation**

Based on the user's query, I will focus on the "Geometry" and "Extrusions" sections of the ParaPy documentation. Specifically, I will look at the ‘geom‘ library, which provides functions for creating and manipulating geometric shapes.

**Step 3: Determine the appropriate level of detail and complexity for my response**

Given that the user is likely a beginner or intermediate developer, I will provide a step-by-step guide to creating an extrusion of an L-shaped flange using ParaPy. I will use simple language and focus on providing practical advice with specific code examples.

**Step 4: Plan the structure of my answer**

Here is the outline for my response:

I. Introduction to Extrusions in ParaPy

* Brief overview of extrusions and their importance in CAD design
* Explanation of how to create an extrusion using the ‘geom‘ library

II. Creating an L-Shaped Flange

* Step-by-step guide to creating a 2D profile for the flange
* Explanation of how to use the ‘geom.extrude()‘ function to create the extrusion

III. Example Code and Next Steps

* Provide a code example demonstrating how to create an extrusion of an L-shaped flange using ParaPy
* Suggest logical next steps for continued learning, such as exploring other geometry creation functions or learning about more advanced topics like parametric design.

Here is my final response:

**Creating an Extrusion of an L-Shaped Flange with ParaPy**

To create an extrusion of an L-shaped flange using the ParaPy SDK, follow these steps:

### Step 1: Create a 2D Profile for the Flange

First, you need to create a 2D profile for the flange. You can use the ‘geom‘ library to create a simple rectangle with two parallel lines:
‘‘‘python
import geom

# Define the dimensions of the flange
width = 10
height = 20

# Create a rectangle with two parallel lines
profile = geom.rectangle(width, height)
‘‘‘
### Step 2: Extrude the Profile

Next, use the ‘geom.extrude()‘ function to create an extrusion from the profile:
‘‘‘python
extrusion = geom.extrude(profile, direction=[0, 1])
‘‘‘
This will create a simple extrusion of the L-shaped flange.

### Example Code and Next Steps

Here is the complete code example:
‘‘‘python
import geom

# Define the dimensions of the flange
width = 10
```

````
height = 20

# Create a rectangle with two parallel lines
profile = geom.rectangle(width, height)

# Extrude the profile
extrusion = geom.extrude(profile, direction=[0, 1])

print(extrusion)
```
This code will output the extruded L-shaped flange.

For next steps, I recommend exploring other geometry creation functions in the `geom` library
 or learning about more advanced topics like parametric design. You can find more
 information on these topics in the ParaPy documentation.

I hope this helps! Let me know if you have any further questions.
````

12.6.3. Full Implementation

The full implementation integrates semantic search infrastructure, enabling retrieval from indexed ParaPy documentation and API references. Tools provide access to domain-specific knowledge, supporting accurate, documentation-grounded responses.

User Query

```
>>> "How can I create an L-shaped flange extrusion using the ParaPy SDK?"
```

Claude Sonnet 4 (API)

```
To create an extrusion of an L-shaped flange using the ParaPy SDK, you'll need to follow a
 two-step process: first create the L-shaped profile as a Wire, then extrude it using
 ExtrudedSolid. Here's how to approach this:

**Step 1: Create the L-shaped Profile**
The L-shape can be created by connecting multiple LineSegments into a Wire. You'll define the
 corner points of your L-shape and connect them with line segments.

**Step 2: Extrude the Profile**
Use the ExtrudedSolid class to extrude your L-shaped profile along a specified direction and
 distance.

The key classes you'll use are:
- `Point`: To define corner coordinates of your L-shape
- `LineSegment`: To create the individual edges of the L-shape
- `Wire`: To connect the line segments into a closed profile
- `ExtrudedSolid`: To perform the extrusion operation

This approach leverages ParaPy's parametric capabilities, allowing you to easily modify the L
 -shape dimensions, extrusion distance, and direction by changing input parameters.


STEP-BY-STEP GUIDE

1. Import the necessary ParaPy geometry classes: Point, LineSegment, Wire, and ExtrudedSolid

2. Define the corner points of your L-shaped profile using Point objects with appropriate
 coordinates

3. Create LineSegment objects connecting consecutive points to form the L-shape outline

4. Combine the line segments into a Wire object to create a closed profile

5. Use ExtrudedSolid with your Wire as the 'island' parameter and specify the extrusion
 distance

6. Optionally add holes to your flange by creating additional Wire objects and passing them
 as the 'holes' parameter


```
```

```
CODE EXAMPLES

Example 1: Basic L-shaped flange extrusion

# Basic L-shaped flange extrusion
from parapy.geom import Point, LineSegment, Wire, ExtrudedSolid

# Define L-shape corner points (example dimensions)
points = [
    Point(0, 0, 0),    # Bottom-left corner
    Point(3, 0, 0),    # Bottom-right of horizontal leg
    Point(3, 1, 0),    # Inner corner of L
    Point(1, 1, 0),    # Inner corner continued
    Point(1, 4, 0),    # Top of vertical leg
    Point(0, 4, 0),    # Top-left corner
    Point(0, 0, 0)     # Close the profile
]

# Create line segments
segments = []
for i in range(len(points)-1):
    segments.append(LineSegment(start=points[i], end=points[i+1]))

# Create the L-shaped profile wire
l_profile = Wire(segments)

# Extrude the L-shape
flange = ExtrudedSolid(island=l_profile, distance=0.5)


Example 2: L-shaped flange with mounting holes

# L-shaped flange with mounting holes
from parapy.geom import Point, LineSegment, Wire, ExtrudedSolid, Circle

# Create L-profile (same as above)
l_profile = Wire(segments)

# Add circular holes for mounting
hole1 = Circle(radius=0.2, position=Position(0.5, 0.5, 0))
hole2 = Circle(radius=0.2, position=Position(2.5, 0.5, 0))
hole3 = Circle(radius=0.2, position=Position(0.5, 3.5, 0))

# Extrude with holes
flange_with_holes = ExtrudedSolid(
    island=l_profile,
    distance=0.5,
    holes=[hole1, hole2, hole3]
)


NEXT STEPS

After creating your basic L-shaped flange extrusion, consider exploring these advanced topics
    :

1. **Parametric Design**: Wrap your flange creation in a ParaPy Part class with Input
    parameters for dimensions, making it easily configurable

2. **Alternative Extrusion Methods**: Explore ExtrudedShell for thin-walled flanges or
    SweptSolid for more complex sweep operations

3. **Boolean Operations**: Learn about Cut, Fuse, and Common operations to add features like
    chamfers, fillets, or additional mounting tabs

4. **Positioning and Orientation**: Study the Position class to place your flange at specific
     locations and orientations in 3D space

5. **Material Properties and Analysis**: Investigate how to add material properties and
    integrate with analysis tools for structural validation

```

```

I'm here to help you with any follow-up questions about ParaPy geometry creation, parametric
    modeling, or specific implementation details for your L-shaped flange project!
```

LLaMA 3.1 (Local)

```
Unfortunately, the search results do not provide a direct answer to how to create an
    extrusion of an L-shaped flange using ParaPy SDK. However, based on the provided
    documentation, it appears that you can use the ‘BSplineSurface‘ class to create a surface
     and then use the ‘extrude‘ method to create an extrusion.

Here is an example code snippet that demonstrates how to create an extrusion of a rectangular
     surface:

‘‘‘
from parapy.geom import BSplineSurface

# Create a rectangular surface
rect_surface = BSplineSurface(
    [0, 1], [0, 1],
    [[(0, 0), (1, 0)], [(0, 1), (1, 1)]]
)

# Extrude the surface
extruded_surface = rect_surface.extrude(10)
‘‘‘

You can modify this code to create an L-shaped flange by adjusting the control points of the
    ‘BSplineSurface‘ class.

Please note that this is just a basic example and you may need to adjust it according to your
     specific requirements. Additionally, you will need to have ParaPy SDK installed in your
    Python environment to run this code.
```

12.7. Tool Design & Architecture

To provide deeper insight into the custom tools developed, this section presents a detailed walkthrough of two to three representative tools, with a particular focus on those that integrate semantic search functionality. A step-by-step development approach is employed to support reproducibility, followed by illustrative (preliminary) results. Interactive examples for all functions are available in `tool_examples.py`.

12.7.1. ParaPy API Reference Search

The `suggest_apis()` [DT01] function interfaces with the `APISemanticSearcher` (section 6.4), which holds the indexed API primitives (subsection 12.3.2). It implements a dispatch strategy that selects specific search methods based on the provided filters. Without filters, it performs a general semantic search using `search_by_functionality()`. If a library filter is provided, it invokes `search_by_module()` to search within a specific library's namespace. A type filter triggers `search_by_type()`, retrieving primitive types (functions, classes, methods). When both filters are present, the function calls the generic `search()` method and applies filtering client-side.

Raw results from the `APISemanticSearcher` are transformed into structured `APISuggestion` objects. For each result, the function extracts the first line of the docstring as a preview (truncated to 100 characters), infers the library name from the module path, and compiles relevant metadata (name, signature, type, module) along with the similarity score. Low-relevance results are filtered using the `min_score` threshold, and the output is limited to a maximum of `max_results` suggestions. This transformation layer bridges the gap between the raw index entries and the structured output expected by downstream functions.

The `suggest_apis_text()` wrapper calls `suggest_apis()` and formats the resulting `APISuggestion` objects into text via `format_for_agent()`. This formatted string includes API names, similarity scores, library information, and docstring previews, and is structured for prompt injection into the agent's context. The separation between `suggest_apis()` (which returns structured data) and `suggest_apis_text()` (which returns formatted text) mirrors the design pattern used in the documentation search interface—core logic generates structured results, while wrappers handle presentation. As outlined in subsection 5.2.4, this approach enables both programmatic access to suggestion data and convenient formatted output for direct agent use.

A typical interaction with the `suggest_apis_text()` function (invoked by the agent) appears as follows:

```
>>> suggest_apis_text(ctx, "create yellow box")
...
...
>>> """
=== API text suggestions for: 'create yellow box' ===
API Suggestions:
==================================================

1. Cube (Score: 0.282)
   Library: parapy.geom.occ.primitives
   Description: A :class:‘Box‘ with equal dimensions.

2. Box (Score: 0.282)
   Library: parapy.geom.occ.primitives
   Description: A box with dimensions :attr:‘width‘, :attr:‘length‘ and :attr:‘height‘.

3. Compound (Score: 0.225)
   Library: parapy.geom.occ.compound
   Description: Make a compound from a list of shapes, :attr:‘built_from‘.
"""
```

12.7.2. Code Runtime Verification

The runtime validation system executes Python code in isolated subprocesses to verify correctness. Instead of using Python's `exec()` function, it writes the code to temporary files and runs them using new Python interpreter instances via `subprocess.run()`. This ensures that decorator validation and import-time checks behave identically to direct file execution, avoiding subtle semantic differences associated with `exec()`. This subprocess-based approach also enables timeout-based protection against infinite loops (default: 120 seconds) and captures `stdout`, `stderr`, and exit codes for analysis. However, since it allows arbitrary code execution on the host machine, it poses a security risk. It must therefore be used with caution and strictly within the current framework.

The `run_code()` [DT07] wrapper formats execution results into structured markdown and implements failure-as-exception behaviour using `ModelRetry`. When execution fails or produces `stderr` output (with `raise_for_warnings=True`), the function raises a `ModelRetry` exception containing the error type, message, traceback, executed code, and a summary. This ensures the AI agent revises faulty code rather than proceeding with errors. The separation between the core execution logic (`_run_code`) and formatted error reporting (`run_code`) aligns with the design pattern seen throughout the tool modules—decoupling mechanics from presentation.

A ParaPy-specific filtering step addresses a known issue: repeated `display()` calls used for 3D visualization may fail during runtime. The `filter_parapy_code()` function uses AST parsing to identify ParaPy classes decorated with `@Attribute` and `@Part`, removes all `display()` calls, and adds code to access all decorated members. This triggers lazy evaluation of the object graph without requiring GUI operations. The filtering logic extracts class structures through AST traversal, identifies instantiated objects in the `if __name__ == "__main__"` block, and synthesizes member access statements that force evaluation. This preserves the logical semantics of the original code while eliminating repeated operations, thus enabling validation of ParaPy logic in automated workflows, such as those used in performance evaluation (section 7.4).

A typical interaction with the `run_code()` function (invoked by the agent) appears as follows:

```
from src.tools import run_code

>>> run_code("print('hello world!')")
...
"""
## Runtime Execution Analysis

```

````
### Execution Status
**Result**: SUCCESS - Code executed without errors
"""
>>> run_code(
    """
def foo(bar: int)
    x = 1 + 2
    return x
    """
)
...
"""
### Execution Status
**Result**: FAILED - Warnings/errors detected in stderr
**Error Type**: `SyntaxError`
**Error Message**: expected ':'

### Error Output
```
  File "C:\Users\ernes\AppData\Local\Temp\tmp11r1b5io.py", line 2
    def foo(bar: int)
                     ^
SyntaxError: expected ':'
```
...
...
"""
````

12.7.3. ParaPy Documentation Search

This function underpins tools ET01–ET05. The `search_educational()` function provides an interface between AI agents and the `SphinxDocSearcher` (section 6.4), which contains the indexed Sphinx documentation (subsection 12.3.2). The function retrieves all indexed entries using the searcher's `search()` method and applies domain-specific filtering and scoring logic.

The architecture separates low-level semantic search (handled by `SphinxDocSearcher`) from application-level filtering and result formatting (handled by `search_educational`). The function supports two primary filters: *content-type filtering* restricts results to specific documentation categories (e.g., tutorial, guide, API, example, reference), allowing agents to request pedagogically relevant content; *score boosting* increases the relevance score (by 1.2x) for entries containing code examples, under the assumption that example-driven content has higher instructional value. Keyword-based exclusion is also supported to remove irrelevant matches.

After filtering, results are sorted by adjusted scores, and the top-k entries are returned. The formatting layer provides two presentation options. The `search_educational_full()` wrapper calls `search_educational()` and uses `format_search_results()` to generate formatted output with content previews (up to 500 characters), code examples (first two per entry, up to 300 characters each), API references, breadcrumbs, and metadata. The `search_educational_compact()` wrapper uses `format_search_results_compact()` to produce single-line summaries with minimal previews.

This dual-formatting strategy addresses token budget constraints: comprehensive results are used when space permits, while compact results maximize coverage under stricter limits. Both wrappers return text ready for prompt injection into the agent's context. A typical interaction with the `search_educational()` function (invoked by the agent) appears as follows:

```
>>> search_educational_compact(ctx, "aircraft wing")
...
"""
=== Compact search: 'aircraft wing' (type: None) ===
('Relevant Documentation:\n'
 '1. primiplane.py (example) | Path: Documentation | Has 1 code examples | '
 'References: Cylinder, primiplane.zip\n'
 '   primiplane.py >>NoteAPIs linksBoxCylinderDownload this '
 'example:primiplane.zipfrommathimportradiansfrom...\n'
 ...
 ...
```

```
"""
```

12.7.4. Tool Table

Table 12.1: Tool Specifications and Implementation Details (Part 1)

Tool ID	Tool Name	Purpose	Category	Input Schema
DT01	`suggest_apis`	Get API suggestions of ParaPy libraries, based on a natural language query.	Data Retrieval	`ctx: RunContext[DevDependencies]` `query: str`
DT02	`suggest_with_context`	Get API suggestions with additional context for AI agents.	Data Retrieval	`ctx: RunContext[DevDependencies]` `query: str` `context: Optional[Dict[str, any]]`
DT03	`get_indexed_api_libraries`	Get all indexed libraries used within the API semantic search engine.	Data Retrieval	`ctx: RunContext[DevDependencies]`
DT04	`get_parapy_examples`	Get relevant examples in ParaPy documentation.	Data Retrieval	`ctx: RunContext[DevDependencies]` `query: str`
DT05	`_describe_symbol_safe`	Import `module_name`, resolve `symbol_name`, and return a structured description of its inputs/parameters.	Code Improvement	`module_name: str` `symbol_name: str`
DT06	`_suggest_imports`	Suggest import statements for names that are used but not defined/imported.	Code Improvement	`code: str`
DT07	`_run_code`	Execute a Python snippet to validate runtime correctness.	Code Verification	`code: str`
DT08	`_check_syntax`	Check syntactic correctness of a Python snippet without executing it	Code Verification	`code: str`
ET01	`_get_learning_path`	Get a structured learning path for a topic	Data Retrieval	`ctx: RunContext[EduDependencies]` `topic: str`
ET02	`_get_quickstart_content`	Get quickstart and getting started content for ParaPy development.	Data Retrieval	`ctx: RunContext[EduDependencies]`
ET03	`search_educational_compact`	Search for educational content with filtering options. Return a compacted format.	Data Retrieval	`ctx: RunContext[EduDependencies]` `query: str`
ET04	`search_educational_full`	Search for educational content with filtering options and return full results.	Data Retrieval	`ctx: RunContext[EduDependencies]` `query: str`
ET05	`find_examples`	Find content with relevant code examples.	Data Retrieval	`ctx: RunContext[EduDependencies]` `query: str`

Table 12.2: Tool Specifications and Implementation Details (Part 2)

Tool ID	Output Type	Parser Func*	Verification (`root_dir='tests/tools'`)	GitHub Ref (`root_dir='src/tools'`)
DT01	`List[APISuggestion]`	`suggest_apis_text`	`./test_api_inspector.py::TestSuggestApis`	`./api_inspector.py#L74`
DT02	`Dict[str, any]`	NA	`./test_api_inspector.py::TestSuggestWithContext`	`./api_inspector.py#L154`
DT03	`str`	NA	NA	`./api_inspector.py#L24`
DT04	`str`	NA	`./test_doc_searcher.py`	`./doc_searcher.py#L9`
DT05	`SignatureInfo`	`describe_symbol`	`./test_export_signature.py`	`./export_signature.py#L56`
DT06	`SuggestImportsResult`	`suggest_imports`	`./test_import_suggestor.py`	`./import_suggestor.py#L195`
DT07	`RuntimeCheckResult`	`run_code`	`./test_runtime_correctness.py`	`./runtime_correctness.py#L115`
DT08	`SyntaxCheckResult`	`check_syntax`	`./test_syntax_correctness.py`	`./syntax_correctness.py#L86`
ET01	`Dict[str, List[Tuple[SphinxDocEntry, float]]]`	`get_learning_path`	`./educational/test_get_learning_path.py`	`./educational/get_learning_path.py#L32`
ET02	`List[Tuple[SphinxDocEntry, float]]`	`get_quickstart_content`	`./educational/test_get_quickstart_guide.py`	`./educational/get_quickstart_guide.py#L43`
ET03	`List[Tuple[SphinxDocEntry, float]]`	`format_search_results_compact`	`./educational/test_search_educational.py::TestSearchEducationalCompact`	`./educational/search_educational.py#L9`
ET04	`List[Tuple[SphinxDocEntry, float]]`	`format_search_results`	`./educational/test_search_educational.py::TestSearchEducationalFull`	`./educational/search_educational.py#L35`
ET05	`List[Tuple[SphinxDocEntry, float]]`	`format_search_results`	`./educational/test_search_educational.py::TestFindExamples`	`./educational/search_educational.py#L63`

12.8. Agent Deployment Details

This section provides detailed specifications for agent output structures, runtime instantiation, and validation mechanisms that complement the integration overview in section 6.7.

12.8.1. Structured Output Schemas

The Developer Agent outputs conform to the `DevOutput` structure, which enforces three required fields:

- `description`: Descriptive explanation of the implementation approach, design decisions, and any assumptions made.
- `completed_code`: Syntactically and runtime-correct ParaPy code when the query involves code generation. Empty string for non-coding queries.
- `problems_solved`: Documentation of problems encountered during code execution and their resolutions, providing transparency about the iterative refinement process.

The Educational Agent outputs follow the `EduOutput` structure, designed to support pedagogical completeness:

- `direct_answer`: Concise yet comprehensive response to the user's primary question, grounded in retrieved documentation.
- `step_by_step_guide`: Structured procedural guidance when queries require sequential implementation steps. Optional field populated based on query type.
- `code_examples`: Relevant code snippets with documentation references, illustrating concepts discussed in the answer. May be intentionally incomplete to focus on specific patterns.
- `next_steps`: Suggestions for continued learning, related topics to explore, or follow-up actions to reinforce understanding.
- `end_offer`: Invitation to ask follow-up questions, maintaining engagement and encouraging iterative learning.

These structures are implemented as Pydantic models, enabling automatic validation and ensuring consistent output formatting across all agent interactions.

12.8.2. Runtime Agent Instantiation

Agent prompts are dynamically constructed at runtime by populating the templates described in section 6.5 with user input and retrieved documentation. The populated prompt is passed via the `instructions` argument to the `Agent.run()` function. This function requires dependency instances (`DevDependencies` or `EduDependencies`) containing initialized semantic search engines, as described in section 6.4.

The following pseudocode illustrates the agent instantiation pattern and execution flow:

```
agent = Agent(
    model=model,
    model_settings=model_settings,
    system_prompt=DEVELOPER_SYSTEM_PROMPT | EDUCATIONAL_SYSTEM_PROMPT,  # depending on agent
    output_type=DevOutput | EduOutput,  # depending on agent
    deps_type=DevDependencies | EduDependencies,  # depending on the agent
    tools=DevTools | EduTools,  # depending on the agent
)

...
...

dev_agent.run(
    user_prompt=create_developer_prompt(),
    deps=DevDependencies(),
)

edu_agent.run(
    user_prompt=create_educational_prompt(),
    deps=EduDependencies(),
```

```
)
```

The `Agent` class wraps the Pydantic AI framework, providing unified access to both local (Ollama) and API-based (Anthropic, Groq) model deployments. The `system_prompt` defines agent role and operational guidelines, while `tools` specifies the functions accessible during execution. The `result_type` enforces structured output schemas through Pydantic validation.

12.8.3. Output Validation Mechanism

The Developer Agent employs an `output_validator` that re-validates generated code before finalizing responses. This validator invokes `run_code()` to verify runtime correctness of the `completed_code` field. If execution succeeds without errors, the output is returned to the user. If runtime errors occur, a `ModelRetry` exception is raised, prompting the agent to revise its implementation. This mechanism operates within the same retry limit configured for the agent's general error recovery behaviour.

This validation pipeline ensures that even when the Developer Agent fails to proactively invoke `run_code()` during its reasoning process, runtime correctness is guaranteed before code delivery to users. The Educational Agent does not employ output validation, as its responses prioritize pedagogical clarity over executable completeness.

12.9. Command Line Interface Application Design

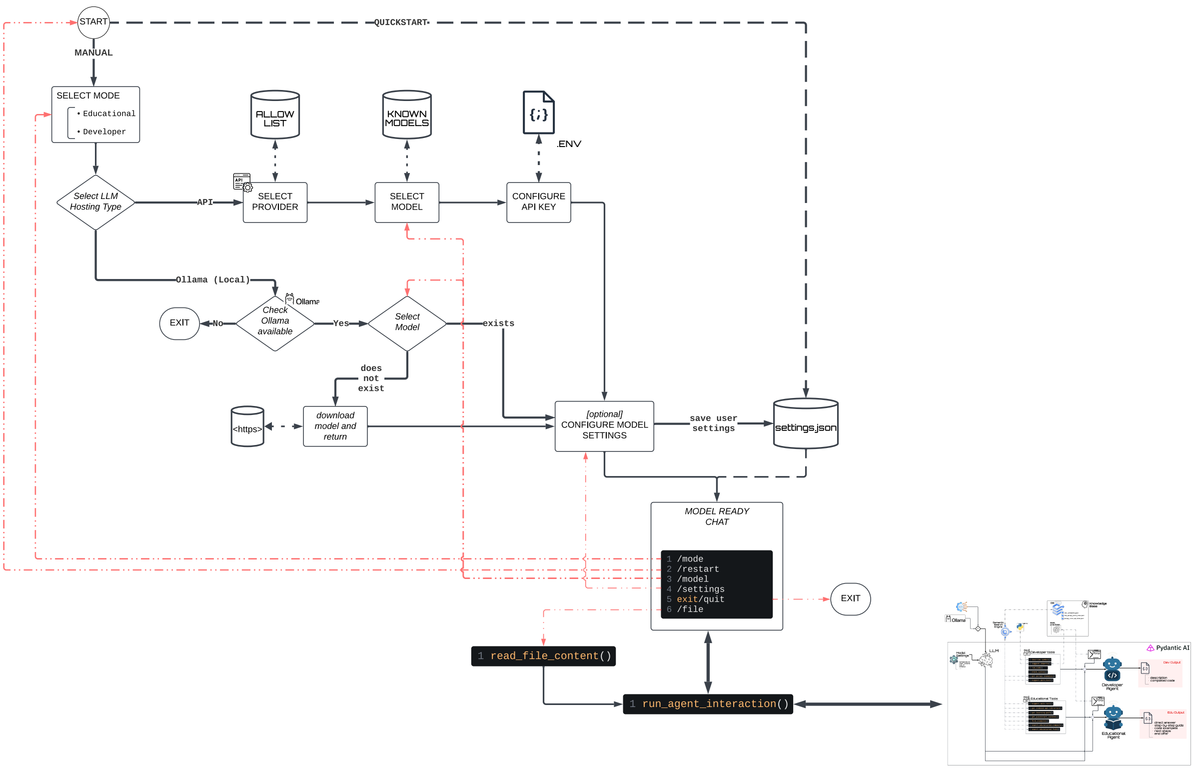


Figure 12.3: Overview of the logical flow and feedback mechanisms, as well as the quick-start pathway of the Command Line Interface (CLI) application developed for the agentic framework. The diagram also illustrates the communication flow between the CLI and the agentic framework.

Figure 12.3 illustrates the functional flow of the application. The CLI abstracts away the otherwise tedious setup steps involved in configuring the framework, thereby preventing runtime errors and reducing initialization time. The user begins by selecting the operational mode appropriate for the current stage of development. This mode can later be switched at any time during a session using the `/mode` command.

After selecting the mode, the user proceeds to configure the model provider—either locally through Ollama or remotely via an approved API provider (Anthropic or Groq, for versions <=`0.4.0`). The user can then select from a list of known models or specify a custom model name. This flexibility allows newly released models to be used immediately without requiring framework updates, though such models are used at the user's own risk, as compatibility is not guaranteed and may result in runtime errors. For Ollama-based deployments, users must ensure that the chosen model supports tool calling; failure to do so may also trigger runtime errors.

The application verifies whether Ollama is installed locally and checks for the availability of the selected model. If the model is not found on the machine but exists in the Ollama model hub, it is automatically downloaded. For API-based deployments, the user is prompted to enter an API key if one is not already stored in the `.env` file. The final step before initiating interaction involves optionally adjusting model parameters such as temperature, sampling rate, and maximum output tokens. Once setup is complete, the configuration is saved to a local `settings` file, allowing a streamlined quick-start in future sessions.

During an interactive session, several commands are available: users can switch the current mode (`/mode`), restart the application (`/restart`), change the model (`/model`), adjust model settings (`/settings`), or exit the program. The `/file` command allows users to append the contents of a selected file to the agent's context. This is achieved by reading the file via the `read_file_content()` function and appending its raw string content to the active user query. File contents persist across the session and can be manually cleared using the `/clear` command if no longer needed.

Interaction with the underlying agent framework—the main focus of Chapter 6—is managed by the `run_agent_interaction()` function. This function handles error propagation and formats the structured outputs of the agents into human-readable text for display in the console. While the interface resembles a chatbot, all interactions are single-turn: previous messages are neither recorded nor added to the agent's context for subsequent runs.

13

Evaluation Framework Design

13.1. Unit Testing Implementation Details

This section provides detailed implementation specifications for the unit testing approach employed in the framework verification process. The testing suite is implemented using `pytest`, with the structure mirroring the hierarchy of the `src` folder. This one-to-one mapping between source code and tests allows for efficient execution of targeted tests in response to changes within specific sub-packages.

Each unit test is designed to isolate the functionality of the feature under test, making use of *patching* where applicable. This involves replacing external dependencies with controlled test substitutes. For example, if a function reads input from a user file, the test patches this behaviour by supplying consistent, mock input, thereby removing uncontrolled variability from the test scope and ensuring reproducibility. In addition, *snapshot testing* is employed wherever possible to verify that the output of a function or component exactly matches an expected reference output (the snapshot). This approach allows for strict one-to-one comparisons of output structure and content, making it easier to detect even minor regressions or unintended changes in behaviour when the codebase evolves.

While achieving 100% test coverage is desirable in theory, the testing suite focuses on covering all critical components and includes integration tests where feasible. However, because the framework relies heavily on large language models (LLMs), traditional integration testing becomes increasingly impractical due to the non-deterministic nature of model outputs. To address this, the `Evals` package from Pydantic AI is employed, as it is specifically designed for evaluating LLM-based systems.

```
Start: What component changed?
 │
 ├─▷ Agent Core (High Risk)
 │    ├▷ Structural change?
 │    │    ├▷ Yes: Full suite
 │    │    └▷ No: Core unit tests
 │    └▷ ``run`` method changed?
 │         └▷ Run Evals
 │
 ├─▷ Tools (High Risk)
 │    ├▷ New tool added?
 │    │    ├▷ Yes: Write tests (unit & integration)
 │    │    └▷ No: Run tool unit tests only
 │    └▷ Affects agent interaction?
 │         └▷ Adapt integration tests and run full suite
 │
 ├─▷ Models (High Risk)
 │    └▷ Rerun full test suite
 │
 ├─▷ Semantic Search Engine (Medium Risk)
 │    ├▷ New functionality added?
 │    │    ├▷ Yes: (Re)Write & run tests:
 │    │    │       unit & integration & affected tools
 │    │    └▷ No: Run engine unit tests only
 │    └▷ Affects agent interaction?
 │         └▷ Adapt integration tests and run full suite
 │
 ├─▷ Prompt Templates (Medium Risk)
 │    ├▷ Prompt structure changed?
 │    │    └▷ Re-run evals
 │    └▷ Output structure changed?
 │         └▷ Run full suite
 │
 └─▷ UI (Low Risk)
      ├▷ UI-isolated?
      │    └▷ Adapt tests and run UI suite
      └▷ Interaction with framework?
           └▷ Run full suite
```

Figure 13.1: Test selection decision tree mapping code changes to required test suites based on component risk assessment.

Concrete Test Examples

One important unit test focuses on the API semantic searcher and reflects the architectural testing principles adopted throughout the framework. Critical functionalities that are not directly targeted by the unit under test are patched out. Mock data is employed to maximize reproducibility and to ensure that the `search_by_functionality` capability of the semantic search engine behaves as expected. Because the mock data is well-structured and its content is known in advance, the expected search results can be clearly defined, enabling precise validation of the engine's behaviour against a synthetic but controlled index.

```python
@patch("src.semantic_search_engine.base.SentenceTransformer")
@pytest.mark.asyncio
async def test_search_by_functionality(self, mock_transformer, temp_api_index_file):
    """Test search by functionality."""
    mock_model_instance = Mock()
    mock_model_instance.encode.return_value = np.array([[0.1, 0.2, 0.3, 0.4]])
    mock_transformer.return_value = mock_model_instance

    searcher = APISemanticSearcher(temp_api_index_file)
    results = await searcher.search_by_functionality("create array", k=2)

    assert len(results) == 2
    for entry, score in results:
        assert isinstance(entry, APIEntry)
        assert isinstance(score, float)
        assert 0.0 <= score <= 1.0
```

Another example concerns the execution of ParaPy code through the `run_code` function, which includes an optional decoding step to handle raw string-formatted code, an output that may occasionally be produced by smaller models. In these cases, snapshot testing is employed to verify the output one-to-one, ensuring that even subtle changes are immediately flagged. This is particularly important given the critical role this functionality plays in the correct operation of the Developer Agent.

```python
def test_parapy_execution(self, snapshot):
    code = """
from parapy.core import Part, Input
from parapy.geom import GeomBase, Box


class SimpleBox(GeomBase):
    width: float = Input()
    depth: float = Input()
    height: float = Input()

    @Part
    def box(self) -> Box:
        return Box(width=self.width, length=self.depth, height=self.height)

if __name__ == '__main__':
    from parapy.gui import display
    obj = SimpleBox(width=1, depth=1, height=1)
    display(obj)
    """

    result = run_code(code)

    snapshot.assert_match(result, "parapy_execution.md")
```

```python
def test_decoding_escaped_newlines_in_code(self, snapshot):
    """Test that _run_code decodes \\n escape sequences in code structure."""
    # This simulates code from an LLM with literal \n for line breaks
    escaped_code = "print('Line 1')\\nprint('Line 2')"
    result = _run_code(escaped_code)

    snapshot.assert_match(str(result), "decoding_escaped_newlines_in_code.md")
```

Test Model

A quick sanity check used during development, and serving as a basis for more elaborate testing such as the evaluations presented in section 7.4, is the `TestModel` feature of the Pydantic AI framework. When a developed agent is configured to use the test model as a mock LLM, the framework automatically attempts to invoke all tools available to the agent and returns either plain text or structured responses, depending on the agent's expected output format.

This test model was consistently employed prior to the release of new features and before formal test coverage analysis was conducted, ensuring that the core implementation of newly developed tools remained compatible with the agent framework. A typical run of the Developer Agent using the test model is shown in Figure 13.2.

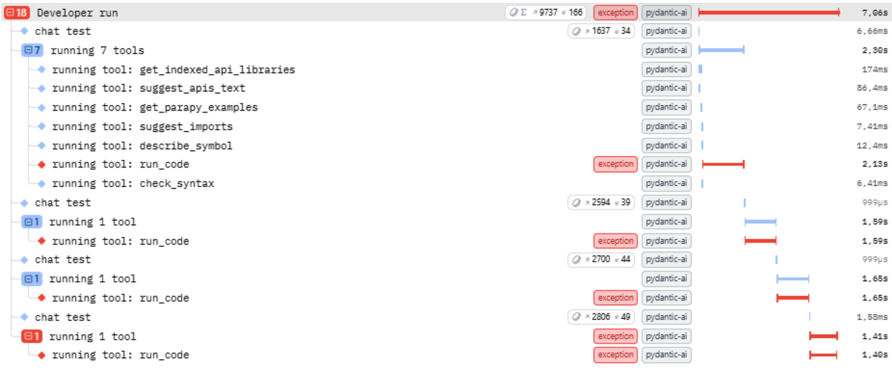


Figure 13.2: Example of a Developer Agent run with the Pydantic AI `TestModel` in place as mock LLM. The test mode architecture attempts to execute all tools available to the agent and return valid responses. Note the triple execution of the `run_code` function at the end, which represents the final output validation of the agent with a maximum retry count of three.

13.2. Large Language Model as Judge Framework

This section provides detailed specifications for the LLM Judge evaluation framework, including model selection rationale, temperature configuration, and complete criterion descriptions with requirement mappings.

13.2.1. Evaluation Protocol

The evaluation methodology follows a systematic protocol: each agent (Developer and Educational) is executed once per evaluation case at a fixed temperature setting ($T = 0.0$) to maximize reproducibility. The resulting outputs are then evaluated by multiple LLM judges, including the agent's own model and two to three alternative models, using dedicated rubrics for developer- and education-specific tasks. By holding agent outputs constant across all judge evaluations, the methodology isolates variance in scoring behaviour and removes randomness introduced by repeated agent executions. All LLM judges operate deterministically at $T = 0.0$ to ensure consistent, reproducible scoring across runs.

13.2.2. Model Selection Rationale

The evaluation methodology employs distinct models for agent execution and judge evaluation to ensure scientific rigour. Using separate models for judging mitigates self-assessment bias, where a model may favourably rate its own outputs, akin to human self-evaluation, and promotes objectivity by introducing independent evaluation perspectives. This strategy more accurately reflects real-world scenarios in which outputs are reviewed by diverse stakeholders, thereby enhancing the external validity of the findings.

The selected judge models, *GPT-OSS-120B*, *LLaMa 3.3 – 70B*, and *Qwen3 – 32B*, represent the most suitable options available under the constraint of using models with sufficient reasoning capacity (i.e., large parameter counts) from the set of permitted model providers. All three were selected from Groq's list of publicly available models. Although *Qwen3 – 32B* falls below the preferred parameter threshold, it was included due to the absence of higher-capacity alternatives within the same environment. For similar reasons, no self-assessment is performed using the *LLaMa 3.1 – 8B* model. Its limited parameter count renders it unsuitable for the extended reasoning tasks required in self-evaluation. Including it would likely compromise result quality, though its exclusion introduces some methodological incompleteness.

Consistency is maintained by using the same judge model for all evaluations of a given agent, while multiple judge models are employed to assess inter-rater reliability and identify potential bias patterns. To evaluate the extent of self-bias, the agent model is also used, alongside the independent judge models, to assess its own outputs. As previously noted, models must be sufficiently large (i.e., greater than 50B parameters) to ensure adequate reasoning capabilities for this task.

13.2.3. Temperature Configuration

While model selection plays a central role in evaluation quality, temperature settings are equally critical for ensuring reproducibility and consistency. All judge models operate at a fixed temperature of $T = 0.0$, eliminating variance due to stochastic sampling and ensuring that identical inputs yield identical evaluations. Similarly, all agent executions are conducted at $T = 0.0$ to maintain consistent outputs across different judge assessments. This deterministic configuration guarantees that observed differences in judge scores reflect genuine quality differences rather than random variation, thereby strengthening the validity of the requirement compliance assessments.

13.2.4. Developer Agent Evaluation Criteria

The Developer Agent's code generation outputs are evaluated by the LLM judge using six weighted criteria, each rated on a 1–5 scale with clearly defined thresholds. A score of 3 or higher indicates acceptable (i.e., passing) quality. The judge has access to the same ParaPy documentation and best practices as the Developer Agent, ensuring consistency in evaluation against established standards.

While the LLM judge framework is primarily designed to evaluate REQ-2-3, it also provides partial or indirect coverage of several other requirements, as detailed below:

1. **Functional Correctness (25%)** – Directly addresses REQ-2-3 by assessing whether the generated solution correctly implements the requested functionality using appropriate ParaPy constructs and patterns. This criterion also provides soft coverage of REQ-1-2 and REQ-1-3, relating to skeleton completion and standalone generation capabilities.
2. **Completeness (20%)** – Evaluates whether the solution comprehensively addresses all aspects of the user request. This indirectly supports REQ-1-2 and REQ-1-3, ensuring that the copilot produces production-ready, holistic responses rather than partial outputs.
3. **Code Quality & Standards (20%)** – Directly supports REQ-3-3 by evaluating adherence to PEP-8 conventions, semantic correctness, maintainability, code structure, and documentation quality, albeit through less rigorous analysis than the code quality framework.
4. **Practical Applicability (15%)** – Assesses whether the generated code is usable in real-world ParaPy projects beyond mere functional correctness. This contributes to evaluating adoption potential and time savings, as described in REQ-4-1, and tests the agent's contextual understanding of engineering workflows.
5. **Error Handling & Robustness (10%)** – Evaluates production-readiness through the handling of edge cases and input validation. This is particularly relevant to REQ-4-3, which emphasizes support for novice users who may lack the expertise to implement robust error handling themselves.
6. **Explanation Quality (10%)** – Assesses the educational value of the agent's responses. This supports knowledge transfer and learning, particularly for novice users, in alignment with REQ-4-3.

13.2.5. Educational Agent Evaluation Criteria

The Educational Agent's outputs are evaluated by the LLM judge using six criteria specifically designed to assess pedagogical effectiveness and learning support. The judge has access to the best practices defined in the Educational Agent's system prompt, ensuring alignment with its instructional objectives. Notably, these best practices include explicit guidance on Knowledge-Based Engineering principles, emphasizing declarative design capture, dependency tracking, and lazy evaluation as core ParaPy concepts (see subsection 12.5.2). The judge evaluates whether the Educational Agent effectively communicates these principles in its pedagogical responses.

While the LLM judge framework is primarily focused on evaluating REQ-2-3, it also provides partial or indirect coverage of additional learning-related requirements, as outlined below:

1. **Pedagogical Effectiveness (25%) –** The primary criterion, directly supporting REQ-4-3, assesses how well the agent scaffolds learning and facilitates progressive understanding.
2. **Content Accuracy (20%) –** Ensures that educational explanations are factually correct and aligned with current ParaPy practices, helping to prevent misconceptions and build user trust.
3. **Clarity & Comprehensibility (20%) –** Evaluates whether explanations are appropriately tailored to the user's experience level, using clear and accessible language. This is particularly important for novice users, as emphasized in REQ-4-3.
4. **Learning Path Coherence (15%) –** Assesses the logical structure and progression of the learning guidance, ensuring appropriate scaffolding that supports efficient skill acquisition and time savings (REQ-4-1).
5. **Resource Integration (10%) –** Evaluates the agent's ability to guide users toward relevant external resources and documentation, promoting self-sufficiency rather than replacing official materials.
6. **Actionability (10%) –** Assesses whether the learning guidance results in clear, concrete next steps rather than vague advice. This helps accelerate learning and reduce user frustration, particularly for novices (REQ-4-3).

13.3. Code Quality Framework

This section provides detailed specifications for the automated code quality assessment framework, including tool configurations, scoring formulas, and ParaPy-specific adjustments.

The quality framework evaluates three key dimensions of code quality: semantic correctness, maintainability, and PEP-8 compliance, each contributing to a weighted final score in line with REQ-3-3. These dimensions collectively reflect whether the generated code is logically sound, easy to understand and modify, and adheres to standard Python coding conventions. Each metric is computed using well-established Python libraries commonly adopted in both industry and research.

Dimension	Sub-metric	Tool	Scoring Rule	Weight
Semantic Correctness	Type Errors (N_{type})	`mypy`	Maximum: 5 points Deduction: -0.5 pts per error	40%
	Logical Errors (N_{logic})	`pylint`	Maximum: 5 points Deduction: -1.0 pts per error	
Maintainability	Maintainability Index (MI)	`radon`	Maximum: 100 points (MI)	35%
	Cyclomatic Complexity (CC)	`radon`	Maximum: 3 points Tiered Scoring: $\max(0, 3 - \lfloor CC/5 \rfloor)$	
PEP-8 Compliance	Style Violations per 100 LOC (v_{100})	`pycodestyle`	Maximum: 10 points	25%

LOC: Lines of Code

Table 13.1: Code Quality Scoring Framework

13.3.1. Semantic Correctness

Semantic correctness is assessed using two widely adopted static analysis tools: `mypy` for type checking and `pylint` for logical correctness.

- **`mypy`** is a static type checker for Python. It verifies type consistency based on optional type hints, enabling early detection of mismatched data types or incorrect function signatures. In this framework, several custom arguments are passed to `mypy` to reduce false positives caused by the dynamic or unsupported features of ParaPy:

 – `-ignore-missing-imports`: Ignores missing type stubs for third-party libraries (e.g., ParaPy), preventing errors when type information is unavailable.
 – `-allow-untyped-decorators`: Permits the use of decorators (such as `@Attribute`) without explicit type annotations, avoiding unnecessary type-checking errors.
 – `-allow-subclassing-any`: Allows subclassing from types of unknown origin (such as `GeomBase`), which may otherwise raise errors due to incomplete type information.

- **`pylint`** is a Python linter that performs a broad range of static code checks, including detection of unreachable code, undefined variables, and other logical flaws. Only critical errors, specifically those in the error (E) and fatal (F) categories, are considered for scoring. Error codes identified as false positives (E0401, E0633, E1130, and E1133[1]), typically caused by unsupported ParaPy constructs, are excluded from the evaluation.

The semantic correctness subscore is capped at 10 points, with 5 points allocated to type correctness and 5 points to logical correctness. For each error reported by `mypy`, 0.5 points are deducted from the type correctness score (minimum 0). For each error reported by `pylint`, 1 point is deducted from the logical correctness score (minimum 0). The total score is computed as shown in Equation 13.1.

$$score_{sem} = \max\left\{0, (10 - 0.5 \cdot N_{type} - 1.0 \cdot N_{logic})\right\} \tag{13.1}$$

13.3.2. Maintainability

Maintainability is evaluated using the `radon` library, a Python tool for analyzing code complexity and maintainability. The reader is referred to [36, 37] for dedicated literature on software maintainability and the principles applied by `radon`.

- **Maintainability Index (MI)** is a composite metric derived from lines of code, cyclomatic complexity, and Halstead volume[2]. It provides a general indication of how easy a piece of code is to understand and modify.
- **Cyclomatic Complexity (CC)** quantifies the number of independent decision paths in the code (e.g., loops and conditionals), with higher values indicating more complex and potentially harder-to-maintain structures.

The maintainability subscore has a maximum of 10 points, weighted 70% on the Maintainability Index (MI) and 30% on the Cyclomatic Complexity (CC). Since the MI produced by `radon` is already scaled from 0 to 100, it only needs to be linearly rescaled to fit the 0–10 scoring range. For the CC component, a tiered scoring system is applied as defined in Table 7.3. The final maintainability score is computed using Equation 13.2.

$$score_{maint} = 7.0 \cdot \frac{MI}{100} + score(CC) \tag{13.2}$$

13.3.3. PEP-8 Compliance

Style consistency is evaluated using `pycodestyle`, a tool that checks Python code against the PEP-8 style guide, the official convention for Python formatting [35]. `pycodestyle` is also used by other popular tools such as `flake8`, making it a well-established and reliable choice for linting.

The framework considers all PEP-8 violation categories reported by `pycodestyle`, including but not limited to whitespace issues, line length violations, indentation inconsistencies, naming convention violations, and import ordering. No violation types are excluded from scoring. Common violations

[1] `E0401`: *"Used when pylint has been unable to import a module."*
`E0633`: *"Used when something which is not a sequence is used in an unpack assignment."*
`E1130`: *"Emitted when a unary operand is used on an object which does not support this type of operation."*
`E1133`: *"Used when a non-iterable value is used in place where iterable is expected."*
From https://pylint.readthedocs.io [accessed 18-10-2025]

[2] *"Halstead's Software Metrics - Software Engineering"*, from https://www.geeksforgeeks.org/software-engineering/software-engineering-halsteads-software-metrics/ [accessed 14-10-2025]

observed in Developer Agent outputs include line length exceedances, inconsistent whitespace around operators, and occasional naming convention deviations. However, all detected violations contribute equally to the final compliance score regardless of type or severity.

The subscore for PEP-8 compliance is based on the number of violations per 100 lines of code, using a tiered system as defined in Equation 13.3. In the edge case of an empty file, a perfect score of 10 points is awarded. For code with $0 < v_{100} \leq 20$ violations, a staggered penalty system is applied. For violation counts exceeding 20, a linear penalty is used.

$$score_{pep} = \begin{cases} 10, & \text{if } v_{100} = 0 \\ 10 - 2 \cdot \min\left(3, 1 + \left\lfloor \frac{v_{100}}{5} \right\rfloor\right), & \text{if } 0 < v_{100} \leq 20 \\ \max\left(0, 3 - 0.1(v_{100} - 20)\right), & \text{if } v_{100} > 20 \end{cases} \tag{13.3}$$

13.3.4. Final Score Computation

By combining these three evaluation dimensions into a single weighted score, as defined by MSC-3-3, the framework offers a scalable and repeatable method for assessing code quality. While it is not a replacement for expert review, it facilitates efficient screening of large volumes of generated code. Random sampling is used to validate results and flag anomalies for more in-depth inspection. The final quality score is computed as a weighted average of the three dimensions, as shown in Equation 13.4:

$$QS = 0.4 \cdot \text{score}_{\text{sem}} + 0.35 \cdot \text{score}_{\text{maint}} + 0.25 \cdot \text{score}_{\text{pep}} \tag{13.4}$$

13.4. Developer Agent: Evaluation Cases

The evaluation cases presented below were designed to cover real-world ParaPy development scenarios, ranging from simple code completion to complex class generation from natural language specifications. Cases are stratified by task type (skeleton code completion versus generation from scratch) and complexity level, spanning debugging tasks, feature implementation, and edge cases that test the agent's robustness under atypical conditions.

Each case provides either a natural language specification or skeleton code with implementation requirements. Agent responses were generated once at temperature $T = 0.0$ using both Claude Sonnet 4 and LLaMa 3.1 8B configurations, then evaluated across four dimensions: syntactic correctness, runtime correctness, functional correctness, and code quality. Results and detailed metrics are presented in section 7.4.

Both the empty and completed skeleton codes of the evaluation cases can be consulted in the `skeleton_code_completion` directory on GitHub.

Table 13.2: Developer Test Cases for ParaPy Assistant Evaluation

ID	Name	Input Description	Difficulty	Category	Domain	Edge Case Type
D-1	Basic Geometry Creation	Help me create a ParaPy class to generate a rectangular profile with length, width, and thickness parameters. Include proper input validation.	Easy	Code Generation	Geometry	N/A
D-2	Complex Assembly	I need to assemble multiple components (fuselage, wing, empennage) into a complete aircraft configuration. Show me how to manage the positioning and connections.	Hard	Code Generation	Assembly	N/A
D-3	Parameterization Debug	My ParaPy model is failing when I change the curvature parameter. The error mentions 'invalid geometry'. Can you help me fix this issue?	Medium	Debugging	Parameterization	N/A

Continued on next page

Table 13.2 – *Continued from previous page*

ID	Name	Input Description	Difficulty	Category	Domain	Edge Case Type
D-4	Performance Optimization	My ParaPy script is running slowly with large datasets. How can I optimize the geometry generation and reduce computation time?	Medium	Optimization	Performance	N/A
D-5	File I/O Integration	I want to import hull profile coordinates from a .csv file and use them to create a ship hull section in ParaPy. Show me the complete implementation.	Medium	Code Generation	File I/O	N/A
D-6	Constraint Handling	Help me implement design constraints in my ParaPy model where the solar panel array area must stay within specific bounds while maintaining structural requirements and optimal sun exposure angle.	Hard	Code Generation	Constraints	N/A
D-7	Advanced Transformations	I need to apply complex transformations (rotation, scaling, translation) to a shunt transformer connector geometry.	Hard	Code Generation	Transformations	N/A
D-8	Integration with External Tools	Show me how to export my ParaPy generated model to CATIA and set up automated analysis workflows.	Hard	Integration	External Tools	N/A
D-9	Error Recovery	My ParaPy application crashes when users input extreme parameter values. Help me implement robust error handling and user feedback.	Medium	Error Handling	Robustness	N/A
D-10	Custom Algorithms	I want to implement a custom algorithm for generating organic-shaped components (like automotive body panels) within the ParaPy framework.	Expert	Code Generation	Algorithms	N/A
D-11	Multi-disciplinary Optimization	Help me set up a ParaPy model that interfaces with thermal and structural solvers for simultaneous optimization of a heat exchanger design.	Expert	Integration	MDO	N/A
D-12	Version Control Integration	I'm working in a team and need to manage ParaPy model versions and dependencies. What's the best approach for collaborative development?	Medium	Workflow	Version Control	N/A
D-12	Version Control Integration	I'm working in a team and need to manage ParaPy model versions and dependencies. What's the best approach for collaborative development?	Medium	Workflow	Version Control	N/A
D-13	MBSE Translation Completion - Simple	MBSE-generated skeleton code for a Fuselage class requiring completion of position, quantify, and radius parameters for circular profiles, plus fixing a typo in the radius calculation.	Hard	Code Completion	MBSE Integration	N/A
D-14	MBSE Translation Completion - Extensive (w/o context)	MBSE-generated skeleton code for a complete Aircraft assembly with incomplete implementations for wing positioning, tail sizing, engine placement, and the determine_engine_position method. Missing imports for LiftingSurface, Engine, and Fuselage classes.	Expert	Code Completion	MBSE Integration	N/A
D-15	MBSE Translation Completion - Extensive (with context)	MBSE-generated skeleton code for a complete Aircraft assembly with incomplete implementations. Includes reference implementations of LiftingSurface, Engine, and Fuselage classes to provide context for completing the Aircraft class.	Expert	Code Completion	MBSE Integration	N/A
D-16	Ambiguous Requirements	Make my ParaPy model faster and better. It needs to handle more complex geometries efficiently.	Medium	Requirements Clarification	Optimization	vague requirements

Continued on next page

Table 13.2 – *Continued from previous page*

ID	Name	Input Description	Difficulty	Category	Domain	Edge Case Type
D-17	Contradictory Constraints	I need a ParaPy model that generates a bottle design with maximum possible volume while keeping the height under 20cm and maintaining a minimum slenderness ratio of 8. Also, it should be optimized for minimum material usage.	Hard	Constraint Resolution	Optimization	impossible constraints
D-17	Contradictory Constraints	I need a ParaPy model that generates a bottle design with maximum possible volume while keeping the height under 20cm and maintaining a minimum slenderness ratio of 8. Also, it should be optimized for minimum material usage.	Hard	Constraint Resolution	Optimization	impossible constraints
D-18	Legacy Code Modernization	I have old ParaPy code from 2015 using deprecated methods (old-style class definitions, create_box function). Can you modernize it to current ParaPy standards?	Hard	Code Refactoring	API Migration	deprecated code
D-19	Poorly Written Code Debug	Minimal, poorly formatted code snippet attempting to create a gear with unclear variable names and missing class definitions. Reports AttributeError with excessive punctuation.	Medium	Debugging	Code Quality	poor code quality
D-20	Security Critical Implementation	I need to implement a ParaPy web service that accepts user input for aircraft parameters and generates models. How do I ensure the input is safe and prevent code injection attacks?	Expert	Security	Web Services	security concern
D-21	Out of Domain Request	Can you show me how to use ParaPy to create a machine learning model for predicting aircraft performance based on historical flight data?	Medium	Scope Clarification	Out of Scope	wrong tool
D-22	Unrealistic Performance Expectations	I need my ParaPy model to generate 10,000 unique aircraft configurations per second with full CFD analysis for real-time optimization. How do I achieve this?	Expert	Performance	Optimization	unrealistic requirements
D-23	Minimal Context Request	Fix this error: 'NoneType' object has no attribute 'position'	Hard	Debugging	Error Resolution	insufficient information
D-24	Multiple Unrelated Concerns	I need help with: 1) Creating a wing geometry, 2) Setting up version control for my team, 3) Exporting to STEP format, 4) Understanding Python decorators, 5) Installing ParaPy on Linux, and 6) Optimizing render performance. Where do I start?	Medium	Prioritization	Mixed Topics	scattered concerns
D-25	Non-Existent Feature Request	How do I use ParaPy's built-in quantum computing module to optimize my wing design using quantum annealing algorithms?	Easy	Feature Clarification	Feature Set	imaginary features
D-26	Copy-Paste Programming	Code copied from Stack Overflow using numpy and matplotlib for 3D surface plotting (terrain surface) that is incompatible with ParaPy's geometry system. Requests direct conversion.	Medium	Code Translation	Framework Migration	wrong framework
D-27	Simple Box with Color	Skeleton code with missing inputs for dimensions and part placeholder for box. Tests use of correct ParaPy primitives.	Easy	Code Completion	Basic Geometry	N/A
D-28	Cylinder Positioning	Skeleton code with predefined inputs and placeholder for cylinder part. Cylinder created and translation should be performed. Tests use of ParaPy primitives and positioning logic.	Easy	Code Completion	Transformations	N/A

Continued on next page

Table 13.2 – *Continued from previous page*

ID	Name	Input Description	Difficulty	Category	Domain	Edge Case Type
D-29	Multiple Spheres with Quantify	Skeleton code to create multiple spheres using `quantify` and position these spheres next to each other using positioning logic and `child`. Tests whether `quantify` is correctly applied, in tandem with `child` without prior knowledge.	Medium	Code Completion	Quantify Pattern	N/A
D-30	Attribute Calculation	Skeleton code with existing box part. Calculate surface area and volume attributes and create second box with different dimensions but identical volume. Tests basic mathematical knowledge and applying it on ParaPy primitives.	Easy	Code Completion	Attributes	N/A
D-31	Rotated Shape	Skeleton code with existing box part. Should create new solid based on box part, rotated around z-axis by `rotation_angle`. Tests correct use of ParaPy primitives (`RotatedShape`) and positioning logic.	Medium	Code Completion	Transformations	N/A
D-32	Derived Input	Skeleton code with normal `Input` and box `Part`. Add a derived input for the box height. Tests whether `Input` is correctly used as decorator.	Easy	Code Completion	Derived Inputs	N/A
D-33	Mirrored Shape	Skeleton with pre-defined input and box part. Create a second solid, mirrored across the YZ plane. Tests correct use of ParaPy primitives (`MirroredShape`) and positioning logic.	Medium	Code Completion	Mirroring	N/A
D-34	Conditional Geometry	Skeleton code for placeholders of conditional shape (either a `Sphere` or `Box`) and attribute placeholders for the surface area and volume of the present shape. Tests whether `DynamicType` is correctly applied and `Part` grammar adhered as opposed to the `Attribute` placeholders.	Medium	Code Completion	Conditional Logic	N/A
D-35	Scaled Shape	Skeleton code with pre-defined inputs and base cylinder `Part`. Cylinder should be uniformly and non-uniformly scaled. Tests whether correct ParaPy primitive is used (`ScaledShape`) and primitive signature is known (scalar scaling factor vs list).	Medium	Code Completion	Scaling	N/A
D-36	List Processing	Skeleton code with pre-defined input for height values and placeholder for `boxes` part. Should create multiple boxes within part based on height values. Check whether `quantify` and `child` are correctly applied and positioning logic is sound.	Medium	Code Completion	List Processing	N/A

13.5. Educational Agent: Evaluation Cases

The evaluation cases presented below were designed to simulate common learning scenarios encountered in ParaPy and Knowledge-Based Engineering development, covering queries from both novice and expert users. Cases are stratified by complexity level (basic, intermediate, advanced) and knowledge domain, ranging from conceptual questions about ParaPy principles to specific implementation guidance, troubleshooting support, and best practices. Edge cases are included to assess the agent's ability to recognize limitations and redirect users appropriately.

Each case consists of a natural language query representing authentic user questions. Agent responses were generated once at temperature $T = 0.0$ using both Claude Sonnet 4 and LLaMa 3.1 8B configurations, then evaluated for functional correctness using the LLM Judge methodology and code quality for included examples. Results and detailed assessments are presented in section 7.4.

Table 13.3: Educational Test Cases for ParaPy Assistant Evaluation

ID	Name	Input Description	Difficulty	Category	Edge Case Type
E-1	Complete Beginner Orientation	I'm completely new to ParaPy and knowledge-based engineering. Can you create a learning path to get me started from zero?	Beginner	Learning Path	N/A
E-2	Concept Clarification	I don't understand the difference between 'attributes' and 'inputs' in ParaPy. Can you explain with practical examples?	Beginner	Concept Explanation	N/A
E-3	Architecture Understanding	How does ParaPy's class inheritance system work? I'm confused about when to use composition vs inheritance.	Intermediate	Architecture	N/A
E-4	Workflow Guidance	What's the recommended workflow for developing a complex ParaPy application from requirements to deployment?	Intermediate	Workflow	N/A
E-5	Best Practices Inquiry	I've written my first ParaPy model but it feels messy. What are the coding standards and best practices I should follow?	Intermediate	Best Practices	N/A
E-6	Troubleshooting Guidance	My ParaPy GUI isn't displaying correctly. Can you guide me through the debugging process step by step?	Intermediate	Troubleshooting	N/A
E-7	Advanced Feature Exploration	I'm comfortable with basic ParaPy but want to learn about advanced features like custom GUIs and web deployment. Where should I start?	Advanced	Advanced Features	N/A
E-8	Industry-Specific Application	I work in the automotive industry. How can ParaPy be applied to car design, and what specific modules should I focus on learning?	Intermediate	Domain Application	N/A
E-9	Integration Learning Path	I need to integrate ParaPy with our existing CAD pipeline (SolidWorks/ANSYS). What do I need to learn and in what order?	Advanced	Integration	N/A
E-10	Documentation Navigation	The ParaPy documentation is overwhelming. Can you help me find information about mesh generation and guide me to the right resources?	Beginner	Resource Navigation	N/A
E-11	Project Planning Assistance	I want to build a parametric drone design tool. Can you break this down into learning milestones and suggest a development timeline?	Advanced	Project Planning	N/A
E-12	Certification Preparation	Are there ParaPy certifications or competency assessments? How should I prepare and validate my skills?	Intermediate	Certification	N/A
E-13	Frustrated Beginner	I've been trying to learn ParaPy for 3 weeks and I STILL can't get a simple box to appear in the viewer. This is impossible! Everything I try fails. Should I just give up? What am I doing wrong???	Beginner	Motivational Support	Emotional distress
E-14	Fundamental Misconception	I read that ParaPy is object-oriented, so I assume each part I create is stored in a database and I can query it with SQL. How do I write SELECT statements to retrieve my wing components?	Beginner	Concept Correction	Major misconception
E-15	Skill Level Mismatch - Too Advanced	I just started learning ParaPy yesterday. Can you explain how to implement custom B-spline surface tessellation algorithms with adaptive refinement based on curvature analysis?	Beginner	Level Adjustment	Unrealistic progression
E-16	Skill Level Mismatch - Too Basic	I've been developing ParaPy applications professionally for 5 years. Can you tell me what 'import' means in Python?	Advanced	Level Adjustment	Skill inconsistency
E-17	Wants to Skip Fundamentals	I don't want to waste time learning Python basics or OOP concepts. Just tell me exactly which buttons to click to make a plane in ParaPy. Can you give me a step-by-step recipe I can follow?	Beginner	Learning Philosophy	Methodology conflict
E-18	Imaginary Features Question	I heard ParaPy has an AI assistant that can automatically generate entire aircraft from a simple sketch. Where do I find this feature in the interface?	Beginner	Feature Clarification	Nonexistent features
E-19	Wrong Tool Comparison	I'm trying to decide between ParaPy and Photoshop for my engineering project. Which one is better for parametric design? How do they compare?	Beginner	Tool Selection	Category confusion
E-20	Unrealistic Timeline	I have a job interview in 3 days where I need to demonstrate expert ParaPy skills. I've never used it before. Can you create a crash course that will make me an expert by Friday?	Beginner	Timeline Planning	Impossible timeline

Continued on next page

Table 13.3 – *Continued from previous page*

ID	Name	Input Description	Difficulty	Category	Edge Case Type
E-21	Overwhelmed and Lost	I've read 500 pages of documentation, watched 20 tutorials, and I still don't know where to begin. Everything seems connected to everything else. I need to create a wing but I don't understand classes, or geometry, or inputs, or parts, or... anything. Help?	Beginner	Orientation	Information overload
E-22	Meta Learning Question	Is learning ParaPy even worth it? What if the company switches to a different tool next year? Should I invest time in this or learn something more universal like general CAD programming?	Beginner	Career Guidance	Existential question
E-23	Vague Multi-Topic Question	Tell me everything about ParaPy - how it works, what I can build, all the features, best practices, common mistakes, and career opportunities. I need a complete overview.	Beginner	Information Request	Overly broad

13.6. Developer Agent: LLM Judge Rubric

The LLM Judge is a separate evaluation agent used exclusively for assessment purposes and is not part of the dual-agent framework. As described in subsection 7.2.3, this methodology employs independent large language models to evaluate Developer Agent responses against defined rubrics, simulating how an experienced ParaPy developer would assess code quality and correctness. The judge assesses functional correctness, adherence to ParaPy SDK conventions, and code quality but does not interact with users or contribute to the framework's operational behaviour.

The prompt below defines the evaluation rubric provided to judge models when assessing Developer Agent outputs to the evaluation cases of section 13.4. Multiple judge models (including Claude Sonnet 4, GPT-OSS-120B, LLaMa-3.3-70B-Versatile and Qwen3, as detailed in Table 7.2) independently evaluate each agent response using this rubric at temperature T=0.0 to ensure consistent, reproducible scoring. This enables systematic assessment of whether the Developer Agent generates well-structured ParaPy code that follows SDK best practices. The respective ParaPy principles and best practises given to the Developer Agent, are also given to the LLM Judge by inserting them at the `documentation` and `$best_practises$`, respectively.

```
DEVELOPER_JUDGE_PROMPT: Template = Template(
    """
You are an expert evaluator assessing the quality of responses from an agent designed to
    assist engineers with ParaPy SDK development. You will evaluate agent responses to user
    prompts requesting code completion, generation, debugging assistance, and technical
    guidance.

### Agent Best Practices Reference

The agent being evaluated has been instructed to follow these best practices and guidelines:

<best_practices>
$best_practises
</best_practises>

The agent also had access to the following description of the ParaPy paradigm:
<parapy_docs>
$documentation
</parapy_docs>

Use these best practices and documentation as reference criteria when evaluating the agent's
    responses. The agent should demonstrate adherence to these standards in its code
    generation, explanations, and technical guidance.

---

### Evaluation Framework

**Input:** You will receive:
1. **User Prompt:** The original agent's request
2. **Agent Response:** The agent's complete response
```

```

**Task:** Rate the agent response on the following criteria using a 1-5 scale where:
- 1 = Poor/Inadequate
- 2 = Below Average
- 3 = Average/Acceptable
- 4 = Good/Above Average
- 5 = Excellent/Outstanding

### Evaluation Criteria

#### 1. Functional Correctness (Weight: 25%)
**Rate 1-5:** Does the solution correctly implement the requested functionality using appropriate ParaPy constructs and patterns?
- **5:** Fully achieves all functional requirements, uses ParaPy best practices, produces correct outputs for all cases
- **4:** Achieves main functionality with appropriate ParaPy usage, minor issues in edge cases or secondary features
- **3:** Achieves core functionality but has notable gaps, suboptimal ParaPy usage, or incorrect behavior in some scenarios
- **2:** Partially functional with significant incorrect behaviors, poor ParaPy implementation, or missing key functionality
- **1:** Does not achieve intended functionality or uses ParaPy incorrectly

**Note:** Syntactic and runtime correctness are evaluated separately. Focus on whether the solution does what was requested and whether it follows ParaPy SDK patterns and best practices.

#### 2. Completeness (Weight: 20%)
**Rate 1-5:** How complete is the solution relative to the user's request?
- **5:** Fully addresses all aspects of the request with comprehensive solution
- **4:** Addresses most aspects with minor gaps
- **3:** Addresses core request but misses some important elements
- **2:** Partially addresses request with significant gaps
- **1:** Incomplete or fails to address main request

#### 3. Code Quality & Standards (Weight: 20%)
**Rate 1-5:** How well does the code adhere to PEP-8 standards, maintain semantic correctness, and demonstrate maintainability?
- **5:** Excellent structure, follows PEP-8 conventions, clear naming, proper documentation, highly maintainable
- **4:** Good structure with minor PEP-8 deviations or documentation gaps
- **3:** Acceptable structure but multiple PEP-8 violations or maintainability issues
- **2:** Poor structure with significant PEP-8 violations or difficult to maintain
- **1:** Very poor code quality, major PEP-8 violations, unmaintainable code

#### 4. Practical Applicability (Weight: 15%)
**Rate 1-5:** How practical and usable is the solution in real development scenarios?
- **5:** Immediately usable, considers real-world constraints and edge cases
- **4:** Mostly practical with minor considerations needed
- **3:** Practical but requires some adaptation
- **2:** Limited practical value, needs significant modification
- **1:** Not practically applicable

#### 5. Error Handling & Robustness (Weight: 10%)
**Rate 1-5:** How well does the solution handle potential errors and edge cases?
- **5:** Comprehensive error handling, considers multiple edge cases
- **4:** Good error handling with minor gaps
- **3:** Basic error handling present
- **2:** Limited error handling
- **1:** No error handling or robustness considerations

#### 6. Explanation Quality (Weight: 10%)
**Rate 1-5:** How clear and helpful are the explanations accompanying the code?
- **5:** Clear, comprehensive explanations that enhance understanding
- **4:** Good explanations with minor clarity issues
- **3:** Adequate explanations
- **2:** Limited or unclear explanations
- **1:** Poor or missing explanations

### Output Format
Provide your evaluation in this exact JSON format:
```

```

```json
{
  "reason": "Comprehensive explanation of the evaluation covering key strengths, weaknesses, and justification for the score and pass/fail decision. Include specific examples from the response.",
  "pass": true,
  "score": 4.2
}
```

Score Calculation: Calculate the weighted average score based on the criteria above:
`(Functional Correctness   0.25) + (Completeness   0.20) + (Code Quality   0.20) + (Practical Applicability   0.15) + (Error Handling   0.10) + (Explanation Quality   0.10)`

Pass Threshold: Set `"pass": true` if the weighted score is   3.0, otherwise `"pass": false`

Reason Field: Provide a comprehensive 3-5 sentence explanation that:
- Individual criterion scores breakdown (e.g., "Technical Accuracy: 4/5, Completeness: 3/5, ...")
- Summarizes overall quality and key evaluation findings
- Highlights main strengths and critical weaknesses
- Justifies the numerical score and pass/fail decision
- References specific aspects of the agent's response

JSON Output Requirements
- **Valid JSON:** Ensure the output is properly formatted JSON that can be parsed programmatically
- **Score Range:** Scores must be between 1.0 and 5.0 (inclusive)
- **Boolean Pass:** Must be exactly `true` or `false` (lowercase)
- **Reason Length:** Keep explanations concise but comprehensive (150-300 words)

Additional Instructions
- Be objective and consistent across evaluations
- Provide constructive feedback that could improve agent performance
- Flag any responses that could be harmful or misleading
- If a response is incomplete or truncated, note this in your assessment
"""
)
```

## 13.7. Educational Agent: LLM Judge Rubric

The LLM Judge is a separate evaluation agent used exclusively for assessment purposes and is not part of the dual-agent framework. As described in subsection 7.2.3, this methodology employs independent large language models to evaluate Educational Agent responses against defined rubrics, simulating how an instructor would assess student work. The judge assesses functional correctness and pedagogical quality but does not interact with users or contribute to the framework's operational behaviour.

The prompt below defines the evaluation rubric provided to judge models when assessing Educational Agent outputs to the evaluation cases of section 13.5. Multiple judge models (including Claude Sonnet 4, GPT-OSS-120B, LLaMa-3.3-70B-Versatile and Qwen3, as detailed in Table 7.2) independently evaluate each agent response using this rubric at temperature $T = 0.0$ to ensure consistent, reproducible scoring. This enables systematic assessment of whether the Educational Agent provides accurate, helpful, and pedagogically sound responses to user queries. The best practises given to the Educational Agent are also given to the LLM Judge by inserting them at `best_practises`.

```
EDUCATIONAL_JUDGE_PROMPT: Template = Template(
    """You are an expert evaluator assessing the quality of responses from an agent designed to guide users through ParaPy SDK learning and documentation. You will evaluate agent responses to learning-oriented queries, guidance requests, and educational support interactions.

### Agent Best Practices Reference

The agent being evaluated has been instructed to follow these pedagogical principles and
```

```
    guidelines:

<best_practices>
$best_practises
</best_practises>

Use these best practices as reference criteria when evaluating the agent's responses. The
    agent should demonstrate adherence to these pedagogical standards in its learning
    guidance, explanations, and resource recommendations.

---

### Evaluation Framework

**Input:** You will receive:
1. **User Prompt:** The original learning/guidance request
2. **Agent Response:** The agent's complete response

**Task:** Rate the agent response on the following criteria using a 1-5 scale where:
- 1 = Poor/Inadequate
- 2 = Below Average
- 3 = Average/Acceptable
- 4 = Good/Above Average
- 5 = Excellent/Outstanding

### Evaluation Criteria

#### 1. Pedagogical Effectiveness (Weight: 25%)
**Rate 1-5:** How effective is the response as a learning tool?
- **5:** Excellent learning design, clear progression, engaging and motivating
- **4:** Good learning structure with minor pedagogical issues
- **3:** Adequate learning approach but could be more effective
- **2:** Poor learning design, confusing or demotivating
- **1:** Ineffective as a learning tool

#### 2. Content Accuracy (Weight: 20%)
**Rate 1-5:** How accurate and up-to-date is the information provided?
- **5:** Completely accurate, current, and reliable information
- **4:** Mostly accurate with minor inaccuracies
- **3:** Generally accurate but some questionable information
- **2:** Contains significant inaccuracies
- **1:** Fundamentally incorrect or misleading information

#### 3. Clarity & Comprehensibility (Weight: 20%)
**Rate 1-5:** How clear and understandable is the explanation for the target audience?
- **5:** Crystal clear, perfectly adapted to user level, easy to follow
- **4:** Very clear with minor complexity issues
- **3:** Generally clear but some confusing elements
- **2:** Difficult to understand, inappropriate complexity
- **1:** Unclear, confusing, or incomprehensible

#### 4. Learning Path Coherence (Weight: 15%)
**Rate 1-5:** How well-structured and logical is the learning progression?
- **5:** Excellent logical flow, clear milestones, well-sequenced
- **4:** Good structure with minor sequencing issues
- **3:** Adequate structure but could be more logical
- **2:** Poor structure, confusing progression
- **1:** No clear structure or illogical sequence

#### 5. Resource Integration (Weight: 10%)
**Rate 1-5:** How well does the response integrate and reference relevant resources?
- **5:** Excellent resource integration, highly relevant and accessible
- **4:** Good resource use with minor relevance issues
- **3:** Adequate resource integration
- **2:** Limited or poorly integrated resources
- **1:** No resource integration or irrelevant resources

#### 6. Actionability (Weight: 10%)
**Rate 1-5:** How actionable and practical are the learning recommendations?
- **5:** Highly actionable with clear next steps and practical exercises
- **4:** Mostly actionable with minor gaps
```

```
- **3:** Somewhat actionable but could be more specific
- **2:** Limited actionability, vague recommendations
- **1:** Not actionable, no clear next steps

### Output Format
Provide your evaluation in this exact JSON format:

‘‘‘json
{
  "reason": "Comprehensive explanation of the evaluation covering pedagogical effectiveness,
      content accuracy, and overall learning value. Include specific examples from the
      response and justification for the score and pass/fail decision.",
  "pass": true,
  "score": 3.8
}
‘‘‘

**Score Calculation:** Calculate the weighted average score based on the criteria above:
‘(Pedagogical Effectiveness    0.25) + (Content Accuracy    0.20) + (Clarity    0.20) + (
    Learning Path Coherence    0.15) + (Resource Integration    0.10) + (Actionability
    0.10)‘

**Pass Threshold:** Set ‘"pass": true‘ if the weighted score is     3.0, otherwise ‘"pass":
    false‘


**Reason Field:** Provide a comprehensive 3-4 sentence explanation that:
- Individual criterion scores breakdown (e.g., "Pedagogical Effectiveness: 4/5, Content
    Accuracy: 5/5, ...")
- Summarizes the educational value and learning effectiveness
- Highlights pedagogical strengths and areas for improvement
- Justifies the numerical score and pass/fail decision
- References specific aspects of the agent’s guidance approach

### JSON Output Requirements
- **Valid JSON:** Ensure the output is properly formatted JSON that can be parsed
    programmatically
- **Score Range:** Scores must be between 1.0 and 5.0 (inclusive)
- **Boolean Pass:** Must be exactly ‘true‘ or ‘false‘ (lowercase)
- **Reason Length:** Keep explanations concise but comprehensive (150-300 words)

### Additional Instructions
- Be objective and consistent across evaluations
- Provide constructive feedback that could improve agent performance
- Flag any responses that could be harmful or misleading
- If a response is incomplete or truncated, note this in your assessment
"""
)
```

13.8. TRS Case Study: Skeleton Code

```python
# -*- coding: utf-8 -*-
#
# Copyright (C) 2016-2023 ParaPy Holding B.V.
#
# You may use the contents of this file in your application code.
#
# THIS CODE AND INFORMATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY
# KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
# IMPLIED WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR
# PURPOSE.

from parapy.core import Input, Part, Attribute
from parapy.geom import GeomBase


class TurbineStage(GeomBase):
    number_vanes: int = Input(13)

```

```python
    # TODO add inputs

    @Part
    def outer_case(self):
        """TODO implement hollow outer casing"""
        pass

    @Part
    def inner_case(self):
        """TODO implement hollow inner casing"""
        pass

    @Part
    def vane_assembly(self):
        """TODO implement vane assembly with inner and outer cases and vanes inbetween,
        The vanes have oval shapes and should run (spanwise) form inner to outer casing.
        """
        pass

    # ------- FEM shape building -------

    @Attribute
    def primitives(self):
        """TODO geometric subshapes to merge, make sure these are solids only"""
        pass

    @Part
    def solid(self):
        """TODO create one solid of assembly"""
        pass

    # ------- meshing -------

    @Part
    def mesh(self):
        """TODO mesh solid with necessary arguments"""
        pass


if __name__ == '__main__':
    from parapy.gui import display

    obj = TurbineStage()
    display(obj)
```

Bibliography

[1] R. Morrar, H. Arman, and S. Mousa. The Fourth Industrial Revolution (Industry 4.0): A Social Innovation Perspective. *Technology Innovation Management Review*, 7(11):12–20, 2017.

[2] R. Sawyer and D. Henriksen. *Explaining Creativity: The Science of Human Innovation*. Oxford University Press, New York, USA, 3rd edition, 2024.

[3] U.D.o. Economic and S. Affairs. *World Social Report 2024: Social Development in Times of Converging Crises: A Call for Global Action*. United Nations, 2024.

[4] U.D.o. Economic and S. Affairs. *The Sustainability Development Goals Report 2024*. United Nations, New York, USA, 2024.

[5] U.D.o. Economic and S. Affairs. *World Economic Situation and Prospects 2025*. United Nations, New York, USA, 2025.

[6] S. Thomke and T. Fujimoto. The Effect of "Front-Loading" Problem-Solving on Product Development Performance. *Journal of Product Innovation Management*, 17:128–142, 2000.

[7] A. Yassine and D. Braha. Complex Concurrent Engineering and the Design Structure Matrix Method. *Concurrent Engineering*, 11(3):165–176, 2003.

[8] A. Kulkarni, G.L. Rocca, T.v.d. Berg, and R.v. Dijk. A knowledge based engineering tool to support front-loading and multidisciplinary design optimization of the fin-rudder interface. In *6th CEAS Conference*, 2017.

[9] R. Eastman. Engineering information release prior to final design freeze. *IEEE Transactions on Engineering Management*, EM-27(2):37–42, 1980.

[10] S.v.d. Elst and E. Moerland. *Full Project Proposal: IDEaliSM*. IDEaliSM, 2018.

[11] A. Kulkarni, D. Bansal, G.L. Rocca, and F. Fernandes. An MBSE approach to support Knowledge Based Engineering application development. In *Aerospace Europe Conference 2023 - 10th EUCASS - 9th CEAS*, 2023.

[12] A.v.d. Laan and T.v.d. Berg. Integrated multidisciplinary engineering solutions at Fokker Aerostructures. In *5th CEAS Air and Space Conference "Challenges in European Aerospace"*, 2015.

[13] G.L. Rocca. Knowledge based engineering: Between AI and CAD. Review of a language based technology to support engineering design. *Advanced Engineering Informatics*, 26:159–179, 2012.

[14] M. Consortium. *Managing engineering knowledge: MOKA: methodology for knowledge based engineering applications*. Professional Engineering Publishing, London, United Kingdom, 2001.

[15] G. Schreiber, B. Wielinga, R.d. Hoog, H. Akkermans, and W.v.d. Velde. CommonKADS: a comprehensive methodology for KBS development. *IEEE Expert*, 9(6):28–37, 1994.

[16] R. Curran, W. Verhagen, M.v. Tooren, and T.v.d. Laan. A multidisciplinary implementation methodology for knowledge based engineering: KNOMAD. *Expert Systems with Applications*, 37(11):7336–7350, 2010.

[17] W. Verhagen, P. Bernell-Garcia, R.v. Dijk, and R. Curran. A critical review of Knowledge-Based Engineering: An identification of research challenges. *Advanced Engineering Informatics*, 26(1):5–15, 2012.

[18] J. Koopman. Towards Generative AI-powered Engineering of Critical Systems: Reverse Engineering Tool for Knowledge Based Engineering Applications & Ideation Matrix for AI-powered Automation Systems. Master's thesis, Delft University of Technology, 2025.

[19] M. Baan. DEFAINE: Design Exploration Framework based on AI for froNT-loaded Engineering. https://itea4.org/project/defaine.html, 2024. Accessed: 2025-03-20.

[20] A. Kulkarni. *DEFAINE Report 3.1.1. Requirement-product-process Ontology*. Delft University of Technology, 2022.

[21] D. Bansal, A. Kulkarni, and G. la Rocca. *DEFAINE Report 3.2.2. ParaPy language ontology and mapping to Knowledge Model*. 2024. confidential.

[22] S. Friedenthal, A. Moore, and R. Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufman, Waltham, USA, 3rd edition, 2015.

[23] F. Fernandes. A Model-Based Systems Engineering Framework for developing Knowledge Based Engineering Applications. Master's thesis, Delft University of Technology, 2023.

[24] T. Heikkinen. Transparency of Design Automation Systems Using Visual Programming - within the Mechanical Manufacturing Industry. *Proceedings of the Design Society*, 1:3249–3258, 2021.

[25] A. Gomez. A team of three: The role of generative AI in development of design automation systems for complex products. *Proceedings of the Design Society*, 5, 2025.

[26] Z. Zheng, K. Ning, Q. Zhong, J. Chen, W. Chen, L. Guo, W. Wang, and Y. Wang. Towards an understanding of large language models in software engineering tasks. *Empirical Software Engineering*, 30(50), 2024.

[27] X. Bai, S. Huang, C. Wei, and R. Wang. Collaboration between intelligent agents and large language models: A novel approach for enhancing code generation capability. *Expert Systems with Applications*, 269:126–357, 2025.

[28] V. Murali, C. Maddila, I. Ahmad, M. Bolin, D. Cheng, N. Ghorbani, R. Fernandez, N. Nagappan, and P. Rigby. AI-assisted Code Authoring at Scale: Fine-tuning, deploying, and mixed methods evaluation, 2024. arXiv preprint.

[29] P.P. Senna, L.M.D.F. Ferreira, A.C. Barros, J.B. Roca and V. Magalhães. Prioritizing barriers for the adoption of Industry 4.0 technologies. *Computers Industrial Engineering*, 171:108428, 2022.

[30] D.f.C.N. Content and Technology. *AI Act | Shaping Europe's digital future*. European Commission, 2024.

[31] J. Weisz, J. He, M. Muller, G. Hoefer, R. Miles, and W. Geyer. Design Principles for Generative AI Applications. *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*, 378:1–22, 2024.

[32] T. Schmid, W. Hildesheim, T. Holoyad, and K. Schumacher. The AI Methods, Capabilities and Criticality Grid: A Three-Dimensional Classification Schema for Artificial Intelligence Applications. *Künstliche Intelligenz*, 35:425–440, 2021.

[33] N. Yüksel, H. Börklü, H. Sezer, and O. Canyurt. Review of artificial intelligence applications in engineering design perspective. *Engineering Applications of Artificial Intelligence*, 188:105–131, 2023.

[34] L. Blessing and A. Chakrabarti. *DRM, a Design Research Methodology*. Springer, London, United Kingdom, 1st edition, 2009.

[35] G.v. Rossum, B. Warsaw, and A. Coghlan. PEP 8 – Style Guide for Python Code. https://peps.python.org/pep-0008/, 2013. Accessed: 28-07-2025.

[36] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *Proceedings Conference on Software Maintenance 1992*, pages 337–344, 1992.

[37] Coleman, Don and Oman, Paul and Ash, Dan and Lowther, Bruce. Using Metrics to Evaluate Software System Maintainability. *Computer*, 27(08):44–49, August 1994.

[38] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 4th edition, 2021.

[39] J. McCarthy, M. Minsky, N. Rochester, and C. Shannon. *A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence*. Dartmouth College, August 1955.

[40] M. Roser. The brief history of artificial intelligence: the world has changed fast — what might be next? *Our World in Data*, 2022. https://ourworldindata.org/brief-history-of-ai.

[41] J. Cullen and A. Bryman. The Knowledge Acquisition Bottleneck: Time for Reassessment? *Expert Systems*, 5(3):216–225, 1988.

[42] A. Barr and E.A. Feigenbaum. The Handbook Of Artificial Intelligence. *Robotica*, 1(2), 1983.

[43] S. C. Kwasny and K.A. Faisal. Overcoming Limitations of Rule-Based Systems: An Example of a Hybrid Deterministic Parser. In *Konnektionismus in Artificial Intelligence und Kognitionsforschung*, pages 48–57, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[44] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016.

[45] T. Fu, J. Zhang, and R. Sun et al. Optical neural networks: progress and challenges. *Light: Science & Applications*, 13(263), 2024.

[46] D. Rumelhart, G. Hinton, and R. Williams. Learning Representations by Back-Propagating Errors. *Nature*, 323(6088):533–536, 1986.

[47] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, and L. Kaiser. Attention Is All You Need. In *31st Conference on Neural Information Processing Systems*, 2017.

[48] T. Brown, B. Mann, N. Ryder, and e. al. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:1877–1901, 2020.

[49] OpenAI. GPT-4 Technical Report, 2023.

[50] W. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, and e. al. A Survey of Large Language Models. *arXiv preprint arXiv:2303.18223*, 2023.

[51] J. Devlin, M. Chang, K. Lee, and K. Toutanova. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding, 2019.

[52] A. Radford and K. Narasimhan. Improving Language Understanding by Generative Pre-Training. 2018.

[53] R. Bommasani, D. Hudson, E. Adeli, and e. al. On the Opportunities and Risks of Foundation Models, 2021.

[54] M. Chen, J. Tworek, H. Jun, and e. al. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*, 2021.

[55] Y. Li, D. Choi, J. Chung, and e. al. Competition-Level Code Generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022.

[56] N. Jain, R. Kwiatkowski, B. Ray, M. Ramanathan, and V. Kumar. On Mitigating Code LLM Hallucinations with API Documentation. *arXiv preprint arXiv:2407.09726*, 2024. arXiv preprint.

[57] Y. Wang, Y. Wen, G. Niu, G. Na, J. Lei, M. Cong, T. Jin, H. Yu, and J. Su. Advancing Code Generation for Multiple Unseen APIs via Retrieval-Guided Example Optimization. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (ACL)*, 2025. Long Paper.

[58] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[59] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, L. Zettlemoyer, N. Cancedda, and T. Scialom. Toolformer: Language Models Can Teach Themselves to Use Tools. *arXiv preprint arXiv:2302.04761*, 2023.

[60] J. Spracklen, R. Wijewickrama, A. Sakib, A. Maiti, B. Viswanath, and M. Jadliwala. We Have a Package for You! A Comprehensive Analysis of Package Hallucinations by Code Generating LLMs. In *USENIX Security Symposium*, 2025. prepublication.

[61] Z. Zhang, Y. Wang, C. Wang, J. Chen, and Z. Zheng. LLM Hallucinations in Practical Code Generation: Phenomena, Mechanism, and Mitigation. *arXiv preprint arXiv:2409.20550*, 2025. arXiv preprint.

[62] Z. Ji, N. Lee, R. Frieske, T. Yu, D. Su, Y. Xu, E. Ishii, Y. Bang, A. Madotto, and P. Fung. Survey of Hallucination in Natural Language Generation. *ACM Computing Surveys*, 55(12):Article 248, 2023.

[63] A. Agrawal, M. Suzgun, L. Mackey, and A. Kalai. Do Language Models Know When They're Hallucinating References? In *Findings of the Association for Computational Linguistics: EACL 2024*, pages 912–928, 2024.

[64] S. Farquhar, J. Kossen, L. Kuhn, Y. Gal, and e. al. Detecting hallucinations in large language models using semantic entropy. *Nature*, 630:625–630, 2024.

[65] Z. Wang, A. Asai, X. Yu, F. Xu, Y. Xie, G. Neubig, and D. Fried. CodeRAG-Bench: Can Retrieval Augment Code Generation? *arXiv preprint arXiv:2406.14497*, 2024. arXiv preprint.

[66] W. Jules, F. Quchen, H. Sam, S. Michael, O. Carlos, G. Henry, E. Ashraf, S. Jesse, and S.D. C. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. *arXiv preprint*, 2023.

[67] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, and F. Xia et al. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[68] L. Guohao, Y. Zheng, Z. Zhiwei, Q. Chen, L. Zihan, D. Yutong, S. Jing, and E. Stefano. CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society. *arXiv preprint*, 2023.

[69] W. Lei, M. Chen, F. Xueyang, Z. Zeyu, Y. Hao, Z. Jingsen, C. Zhiyuan, T. Jiakai, C. Xu, L. Yankai, Z.W. Xin, W. Zhewei, and W. Ji-Rong. A Survey on Large Language Model based Autonomous Agents. *arXiv preprint*, 2023.

[70] Z. Xia, W. Chen, L. Jing, et al. Comparison of Prompt Engineering and Fine-Tuning Strategies in Large Language Models in the Classification of Clinical Notes. *medRxiv*, 2024.

[71] S. Yao, D. Yang, E. Bosma, Q. Wang, and E.C.e. al. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*, 2023.

[72] L. Gao, Z. Dai, and J.C.e. al. Retrieval-Augmented Generation for Large Language Models: A Survey. *arXiv preprint*, 2023.

[73] H. Xinyi, Z. Yanjie, W. Shenao, and W. Haoyu. Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions. *arXiv preprint*, 2025.

[74] A. Gomez, M. Panarotto, and O. Isaksson. Evaluation of Different Large Language Model Agent Frameworks for Design Engineering Tasks. In *NordDesign 2024*, 2024.

[75] W. Thomas, D. Lysandre, S. Victor, et al. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, 2020.

[76] P.J. Sung, O. Carrie, C.C. J., M.M. Ringel, L. Percy, and B.M. S. Generative Agents: Interactive Simulacra of Human Behavior. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST)*, 2023.

[77] V. Braun and V. Clarke. *Thematic Analysis: A Practical Guide*. SAGE Publications, 2021.

[78] S. Barke, M. James, and N. Polikarpova. Grounded Copilot: How Programmers Interact with Code-Generating Models. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):85–111, 2023.

[79] Anthropic. Prompt engineering overview. `https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/overview`. Accessed: 03-09-2025.

[80] C. Shannon. Prediction and Entropy of Printed English. *Bell System Technical Journal*, 30(1):50–64, 1951.

[81] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.

[82] T. Mikolov, K. Chen, G. Corrado, and J. Dean. Efficient Estimation of Word Representations in Vector Space, 2013.

[83] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 1997.

[84] J. Gehring, M. Auli, D. Grangier, and Y. Dauphin. Convolutional Sequence to Sequence Learning. In *Proceedings of the 34th International Conference on Machine Learning (ICML)*, pages 1243–1252, 2017.

[85] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, and e. al. Training Language Models to Follow Instructions with Human Feedback. *arXiv preprint arXiv:2203.02155*, 2022.

[86] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. Liu. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

[87] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi. The Curious Case of Neural Text Degeneration. In *International Conference on Learning Representations (ICLR)*, 2020.

[88] GitHub Copilot. https://github.com/features/copilot, 2025. Accessed: 2025-08-20.

[89] D. Guo, Q. Zhu, D. Yang, et al. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[90] Pydantic Services Inc. Pydantic AI - GenAI Agent Framework, the Pydantic way. https://ai.pydantic.dev/. Accessed: 02-10-2025.

[91] LLama Team, AI Meta. The Llama 3 Herd of Models, 7 2024.

[92] Anthropic. *System Card: Claude Opus 4 & Claude Sonnet 4*. 5 2025.

A

Extended Analysis: Industrial Study GKN

This appendix presents the complete interview data and analysis from the industrial study conducted at GKN Aerospace Sweden, as summarized in Chapter 4. The study aimed to validate the research gaps identified in Chapter 1 and Chapter 3 by examining, from an industrial perspective, whether KBE adoption is indeed time-intensive and knowledge-intensive, and by identifying the specific challenges users encounter during this process.

Semi-structured interviews were conducted with five domain experts at GKN Aerospace Sweden between May 4th, 2025, and May 16th, 2025. Participants were selected for their involvement in software development, engineering automation, and/or AI-KBE integration initiatives, representing a diverse range of technical competencies in KBE application development using the ParaPy framework. This diversity in roles and expertise was strategically chosen to capture both the entry barriers faced by novices and the sustained adoption challenges experienced by more experienced practitioners.

Each interview lasted between 60 and 90 minutes and followed a semi-structured format that balanced consistency across sessions with the flexibility to explore individual perspectives. Participants were first introduced to the concepts of KBE and a demonstrator application (a ParaPy-based Turbine Rear Structure model), followed by a guided live coding session. This hands-on approach enabled participants to gain practical exposure to the ParaPy SDK and its application development workflow. Subsequently, they were asked to reflect on their experience with the demonstration, their existing knowledge of KBE principles and automation practices, and their perceptions of how Generative AI or LLM tools might support similar development tasks in the future. No AI-based development tools were demonstrated or used during the interviews.

The interview guide consisted of open-ended questions focusing on AI-KBE integration challenges, perceived benefits, organizational readiness, and technical barriers. All sessions were audio-recorded with participant consent and subsequently transcribed verbatim. The consent form is included in section A.4, and the full interview script is provided in section A.5.

The analysis employed reflexive thematic analysis following the six-phase approach outlined by [77], with particular focus on two themes most relevant to the proposed support system: *Skill Gap & Learning Curve* and *AI/LLM Assistance – Potential & Trust Concerns*. These themes, which emerged universally across all five participants, directly informed the design of the dual-agent architecture discussed in Chapter 6. The complete thematic analysis, including all eight identified themes, individual interview summaries, and representative quotes, is presented in the sections that follow.

A.1. Methodology

The analysis was based on reflexive thematic analysis, following the six-phase approach outlined by [77]:

1. Familiarization with the data through repeated reading of the transcripts.

2. Initial coding, using both manual annotation and local AI-assisted suggestions.
3. Theme development by grouping related codes.
4. Reviewing and refining themes to ensure internal coherence and distinctiveness.
5. Defining and naming themes in relation to the research objectives.
6. Producing a thematic narrative, supported by illustrative quotes from the interviews.

To enhance efficiency and consistency during initial coding, a local large language model (LLM) was employed to generate interview summaries and identify candidate themes. Prompts were developed to extract both high-level themes and representative quotes. While AI was used to support the coding process, all outputs were manually reviewed, validated, and revised by the researcher to ensure interpretative rigour.

Following the individual analysis of each interview, a second round of synthesis was conducted to identify patterns across participants. This cross-case analysis focused on consolidating recurring themes, highlighting contradictions, and exploring relationships between the identified challenges. The purpose of this synthesis was to distill the collective meaning of participants' accounts, uncover shared challenges and opportunities, and generate analytically robust findings suitable for presentation in the current chapter. All participants reflected on the practicalities of using AI-driven KBE tools—primarily ParaPy, Siemens NX [1] with a Copilot, and Knowledge Fusion [2] —within existing design and analysis workflows.

A.2. Findings

Although the distribution of experience has been addressed at a general level above, it is useful to further contextualize the participants' backgrounds. Interviewee 5 is the most senior member of the group, both in terms of chronological age and professional experience. In contrast, Interviewees 2 and 4 are junior employees, each holding a PhD and relatively early in their professional careers. Interviewees 1 and 3 represent an intermediate category, with experience levels situated between these two extremes.

A.2.1. Nomenclature

Table A.1: Terms and abbreviations used by the Domain Experts (DEs) during the interviews.

Abbreviation	Definition
AML	Adaptive Modelling Language (KBE modelling framework)
ANSA	CAE pre-processing tool developed by BETA CAE Systems [3]
ANSYS	Comprehensive suite of CAE software developed by ANSYS, Inc. [4]
CFD	Computational Fluid Dynamics
CFX	CFD software developed by ANSYS
DC	Design Convention(s)
DP	Design Practise(s)
NX	Siemens NX[1]
PLM	Product Lifecycle Management
ROI	Return On Investment
SAP	Systems, Applications, and Products
TRS	Turbine Rear Structure

A.2.2. Common Themes Across Interviews

Inadequate Knowledge Capture & Formalisation

All five interviewees mention this theme.

Knowledge about design rules, standards, and past analyses is stored in disparate PLM documents, scripts, reports, or "ask-an-expert" conversations. No unified, query-able knowledge base exists, making reuse error-prone.

Table A.2: Overview of common themes identified in the interviews and their relative urgency, based on the number of Domain Experts (DEs) referencing each theme.

Nr	Theme	DE responses ($n = 5$)
1	Inadequate knowledge capture & formalisation	5
2	Integration & tool-chain heterogeneity	5
3	Skill gap & learning curve	5
4	AI/LLM assistance – potential & trust concerns	5
5	Perceived value & benefit-effort trade-off of KBE	5
6	Usability & documentation shortcomings	4
7	Organizational & cultural barriers	5
8	Trust, reliability & validation of automated/AI solutions	3

- "Right, so we have documents that basically describe them [design rules, lessons learned and/or best practises] and they're stored somewhere on some sort of PLM platform." - Interviewee 1
- "There are certain documents and standards and some scripts which kind of incorporate our knowledge... also on top of that, we have DPs and best practices in certain teams... It is kind of spread out in multiple ways." - Interviewee 3
- "We have the system with the DPs, the design practices, the DC post to fulfil that function... there is also mentoring, and you have the more senior people helping the engineers." - Interviewee 4
- "We have three Single Sources of Truth now... Teamcenter, we have SAP, and we have GitHub." - Interviewee 5

All participants noted the scattering of knowledge across the organization, though the form it takes varies. Interviewee 1 emphasized reliance on informal peer support; Interviewee 2 highlighted the absence of a formalized knowledge base; Interviewee 3 pointed to the use of scripts and standards; Interviewee 4 referred to distributed design practices (DPs, DC); and Interviewee 5 described the existence of multiple, competing "single sources of truth."

Integration & Tool-Chain Heterogeneity

All five interviewees mention this theme.

KBE tools (ParaPy, AML, Knowledge Fusion, [Siemens] NX-journaling) must interoperate with CAD (NX, ANSA), PLM (Teamcenter), CFD (ANSYS/CFX), optimisation, and version-control systems. Persistent friction arises from data-exchange formats, proprietary APIs, and the need to bridge legacy and modern platforms.

- "My concerns are what exactly is the benefit of this [ParaPy] application over a well-parametrized NX model?" - Interviewee 2
- "We had an attempt that we call 'Engineering Work Bench'... it has not been used so much." - Interviewee 5
- "A lot of the time when we run parametric studies or something like that, you kind of have to just repeat the same thing six times on your own, manually." -Interviewee 1

The perceived sources of difficulty varied across participants. Interviewee 1 identified the absence of a formal automation framework; Interviewee 2 expressed concern about inadequate data flow to the PLM system; Interviewee 3 noted that integration is significantly easier when all components are developed in Python; Interviewee 4 highlighted the persistence of manual post-processing; and Interviewee 5 described platform fragmentation as a strategic barrier to progress.

Skill Gap & Learning Curve

All five interviewees mention this theme.

Successful KBE requires engineers to be proficient both in domain knowledge and in software development (Python/ParaPy, CAD APIs). The steep learning curve deters adoption, especially for designers without coding experience.

- *[What do you see as the biggest technical challenges to adopting this kind of KBE in your day-to-day engineering work?]* "For me as an employee, there is the learning curve." - Interviewee 2
- "[For] a designer who has no Python experience.. [extending an KBE appi] would be pretty tall requirement." - Interviewee 3
- "The entire ParaPy way of thinking with the parts and attributes... I'm not very familiar with developing in this framework... the main thing is... knowing what the clever or the best approach is" - Interviewee 4
- "We need a couple [engineers] that will be able to work in this, so it's difficult [to integrate]." - Interviewee 5
- "If you're fully unaware of something like this [an existing ParaPy codebase], LLMs are incredibly helpful." - Interviewee 1

While all participants acknowledged the existence of a skill gap, their perceptions of its severity varied. Interviewee 1 considered their own coding skills sufficient but expressed concern about colleagues without programming experience. Interviewees 2 and 3 emphasized the need for broader organizational training. Interviewee 4 admitted to a "lack of deep understanding," yet demonstrated the ability to quickly adapt and modify existing code. In contrast, Interviewee 5 viewed the presence of a stable pool of NX-proficient engineers as a mitigating factor.

AI/LLM Assistance – Potential & Trust Concerns

All five interviewees mention this theme.

Participants use ChatGPT, Copilot, Gemini, or locally hosted LLMs for code snippets, documentation, and troubleshooting. Expectations range from generic code-completion to domain-specific chatbots that can explain ParaPy APIs. Simultaneously, there is scepticism about reliability, "black-box" behaviour, and the need for validation.

- "I've used large language models exclusively since they launched." - Interviewee 1
- "If you are a non-programmer... asking an AI... would have been probably faster, *if* it would have been a reliable answer." - Interviewee 2
- "A chatbot could help everywhere... explain what is happening... mostly from a ParaPy documentation perspective." - Interviewee 3
- "I would like a chatbot with a deep knowledge of both the API and a wealth of examples." - Interviewee 4

Views on the role of AI assistants varied among participants. Interviewee 1 expressed interest in a ParaPy-specific conversational assistant. Interviewee 2 remained sceptical of such tools unless the underlying model is explicitly trained on domain-specific knowledge. Interviewee 3 regarded large language models (LLMs) as a potential remedy for the steep learning curve but acknowledged that the technology is not yet sufficiently mature. Interviewee 4 prioritized the development of a knowledge-rich chatbot over generic code-completion tools. In contrast, Interviewee 5 saw limited value in AI for deep CAD scripting and emphasized the importance of proofreading of design documents over design automation.

Perceived Value & Benefit-Effort Trade-off of KBE

All five interviewees mention this theme.

The TRS app demo demonstrated the promise of end-to-end automation (geometry → mesh → CFD → post-processing). However, participants differ on when the payoff justifies the upfront development cost.

- "The level of integration [of the TRS app] ... is not seen at GKN currently." - Interviewee 1

- "It is really powerful [the TRS app]... but I think that a lot of KBE applications, they focus on the creation of design information, while not really representing the actual context of the information, which would turn it into actual knowledge." - Interviewee 2
- "It opens up a lot of possibility, but also it has this curve, like the effort to benefit curve, which we need to know where it will cross." - Interviewee 3
- "This is really powerful and potentially super useful... but the biggest fear... is that we cannot get the support that we need in the future." - Interviewee 4
- "We need a clear business case . . . otherwise we stick with NX." - Interviewee 5

Participants expressed differing views on the practical value and strategic implications of KBE tools. Interviewee 1 was enthusiastic about the potential for rapid visualization and real-time geometry updates. Interviewee 2 raised concerns about the opacity of embedded knowledge structures, despite acknowledging the technical capabilities of such systems. Interviewee 3 characterized KBE as most suitable for early-stage conceptual work, where the benefit must outweigh the required effort. Interviewee 4 recognized the value of automation but expressed concern over potential vendor lock-in. Finally, Interviewee 5 argued that existing NX workflows already address most engineering needs, framing the adoption of new KBE tools as a strategic rather than a technical decision.

Usability & Documentation Shortcomings

Four out of five interviewees mention this theme.

Code bases are large, partly documented, and engineers struggle to locate relevant objects, understand parameter interactions, or visualise geometry without sketches. Desired UI features include bi-directional mapping of GUI actions to code.

- "The biggest hurdle . . . is understanding the existing code and the parameterisation." - Interviewee 2
- "If you click on an object . . . the tool would point you to the code... that would be incredible." - Interviewee 1
- "It's just easier for humans... to read geometry or drawings than to read code." - Interviewee 2
- "Having a web GUI would just be a game-changer for us in Solid Mechanics." - Interviewee 1

Several participants raised concerns related to code transparency and documentation. Interviewee 1 emphasized the importance of visual-to-code traceability. Interviewee 2 pointed to inadequate documentation, noting that code comments alone are insufficient. Interviewee 3 highlighted the absence of systematic testing, which was viewed as a related documentation and maintainability issue. Interviewee 4 reported difficulty navigating the codebase, further underscoring the need for improved structural clarity.

Organizational & Cultural Barriers

All five interviewees mention this theme.

Adoption is constrained by management commitment, business-case justification, fear of vendor lock-in, and the need to up-skill a large engineering workforce.

- "If it still requires a programming background, it'll alienate some engineers." - Interviewee 1
- "Spreading that approach through the company would require quite a bit of education and training." - Interviewee 2
- "Cultural tension – you need to be really good at Python and also you should be really good at engineering and what knowledge to encode and how you can encode that efficiently using Python." - Interviewee 3
- "Chicken-and-egg problem. . . you want to know it can solve future problems before you invest." - Interviewee 4
- "I mean, you need to show why ParaPy is... that way of doing parametric models and so on is better than NX." - Interviewee 5

A common thread across the interviews is that technical feasibility alone is insufficient without strategic alignment. Interviewee 1 highlighted that programming requirements risk alienating engineers without coding backgrounds, creating a barrier to widespread adoption. Interviewee 2 emphasized that scaling the approach would require substantial organizational investment in education and training. Interviewee 3 identified a cultural tension inherent in KBE: engineers must excel both in their domain expertise and in Python programming, while also understanding how to efficiently encode knowledge. Interviewee 4 articulated a chicken-and-egg problem wherein management requires evidence that the technology can solve future problems before committing resources. Finally, Interviewee 5 argued that adoption ultimately depends on demonstrating a clear business case showing why ParaPy offers advantages over existing tools like NX.

Trust, Reliability & Validation of Automated/AI Solutions

A moderate three out of five interviewees mention this theme.

Participants express concern over the lack of systematic testing, the "black-box" nature of AI suggestions, and the risk of hidden errors in safety-critical aerospace applications.

- "If you use an AI that understands the code for me, I don't understand the code." - Interviewee 2
- "There's a lack of automated tests. . . yet I assume the CFD results are reliable." - Interviewee 3

Trust emerged as a cross-cutting concern, closely linked to both skill gaps and perceptions of AI. The lack of systematic testing was often attributed to limited expertise, while concerns about AI-generated code reflected apprehension toward black-box systems. Interviewee 2 expressed explicit distrust of AI-generated code, emphasizing the need for transparency and control. In contrast, Interviewee 3 was more optimistic, suggesting that LLMs could assist by explaining code behaviour. This contrast highlights a broader tension between confidence in automation and the need for validation and interpretability.

A.2.3. Diverging or Contradictory Findings

- **AI Usage vs. Perceived Need**

  Interviewee 1 reports exclusive daily ChatGPT use yet still asks for a dedicated ParaPy chatbot; Interviewee 4 uses Copilot extensively but downplays its importance, preferring a knowledge-rich chatbot.

- **Value of KBE vs. Opacity**

  Interviewee 1 sees the integrated TRS workflow as a game-changer; Interviewee 2 praises the demo's power but repeatedly stresses that knowledge is hidden, making the tool feel like a black box.

- **Scope of KBE Applicability**

  Interviewee 1 envisions KBE for non-conformance modelling and rapid geometry updates; Interviewee 2 questions added value over a well-parameterised NX model; Interviewee 3 limits KBE to early-stage, CFD-driven studies; Interviewee 5 argues NX already covers most needs.

- **Trust in AI-Generated Code**

  Interviewee 2 warns against blind reliance on AI without domain-specific training; Interviewee 3 believes an LLM can explain code and thus mitigate risk; Interviewee 5 finds AI helpful only for proofreading, not for deep CAD scripting.

- **Automation Tools Existence**

  Interviewee 1 states "we have in-house Python utilities" yet also claims no automation tools exist in his department (00:06:30), indicating a semantic split between ad-hoc scripts and formalised automation frameworks.

- **Vendor Dependency vs. Openness**

  Interviewee 4 worries about future vendor support for ParaPy; Interviewee 5 is comfortable with vendor-specific LLMs (e.g., ANSYS) and sees little need for internal AI development.

A.2.4. Inter-Theme Relationships

- **Skill Gap // Integration Challenges**

  Without sufficient programming expertise, engineers cannot create the adapters needed for seamless data exchange between CAD/PLM and KBE tools, exacerbating the integration problem.

- **Knowledge Capture Deficiency // Trust & Validation**

  Scattered, undocumented knowledge leads to black-box perceptions, reducing confidence in automated outputs and making validation harder.

- **AI/LLM Assistance // Skill Gap**

  AI chatbots are explicitly envisioned as learning aids to bridge the skill gap; however, the trust issue (Theme 8) tempers this benefit—engineers may rely on AI without fully understanding the underlying code.

- **Organizational Barriers // Perceived ROI** Management's demand for a clear business case (Theme 7) directly influences whether the perceived value (Theme 5) of KBE is acted upon; lack of ROI evidence stalls investment in training or tooling.

- **Usability & Documentation // Adoption** Poor documentation and unintuitive UI increase the learning curve, reinforcing the skill gap and organizational resistance. Conversely, a bi-directional UI (Theme 6) could lower barriers and improve trust. While such an interface is not implemented in the current work, the MBSE-driven KBE approach in [23] presents a promising direction: by enabling traceability from SysML-modeled rules directly to corresponding ParaPy code snippets, it takes an important step toward realizing bi-directional integration.

- **Effort-Benefit Trade-off // Trust & Validation** When developers perceive the initial effort as high (Theme 1, 3), they demand robust testing and transparent knowledge to justify investment—thus trust and validation become prerequisites for perceived benefit.

- **Vendor Dependency // Integration & Heterogeneity** Fear of being locked into a vendor-specific KBE platform (Theme 4) heightens concerns about tool heterogeneity and the cost of integration with existing PLM/CAD ecosystems.

A.2.5. Overall Insights

- *[IS-Insight-I]*

  Fragmented knowledge management is a systemic barrier. Across all interviewees, expertise lives in scattered scripts, PLM documents, and informal mentoring, preventing reliable reuse and eroding trust in automated tools.

- *[IS-Insight-II]*

  A persistent skill gap—especially in Python/ParaPy programming—drives both the learning-curve barrier and the demand for AI-driven learning support. Yet, trust in AI outputs remains low without domain-specific training and systematic validation.

- *[IS-Insight-III]*

  The perceived value of AI-enhanced KBE hinges on a clear benefit-effort threshold. Early-stage, concept-generation tasks are widely seen as the sweet spot, whereas detailed solid-mechanics or certification-critical work is still considered unsuitable.

- *[IS-Insight-IV]*

  Organisational and cultural constraints (business case, management endorsement, vendor lock-in) dominate over pure technical feasibility. Even when the technology works, adoption stalls without strategic alignment and dedicated up-skilling programmes.

- *[IS-Insight-V]*

  Usability, documentation, and bi-directional UI are decisive for adoption. Participants repeatedly called for a web-based front-end that maps GUI actions to code snippets, reducing reliance on deep code inspection and fostering confidence.

These insights suggest that successful deployment of AI-augmented KBE at GKN (and similar industrial contexts) requires a co-evolution of knowledge capture frameworks, skill-development pathways, trustworthy AI assistants, and tightly-integrated, user-centred tooling—all underpinned by a demonstrable ROI that convinces senior management to invest in the required cultural shift.

A.3. Summary & Conclusion

Table A.3: Overview of how the proposed dual-agent system—comprising an Educational Agent and a Coding Agent—addresses the common themes identified in the GKN interviews.

Nr	Theme	Addressed?	*Rationale*
1	Inadequate knowledge capture & formalisation	No	*Organisational issue related to the implementation of KBE. Also partially covered already by the approach in [23]*
2	Integration & tool-chain heterogeneity	Partly	*Incorporation of KBE tools (and ParaPy) can be sped up by use of Coding Agent*
3	Skill gap & learning curve	Yes	*Learning curve can be significantly flattened by both the Educational and Coding Agent*
4	AI/LLM assistance – potential & trust concerns	Yes	*Perfect example of how the Educational Agent can be employed [The notion of needing a model fine-tuned on domain specific knowledge (Interviewee 2) can be rejected by the findings in subsection 3.3.2]*
5	Perceived value & benefit-effort trade-off of KBE	Partly	*Agent framework can lower perceived effort and provide "passive" support*
6	Usability & documentation shortcomings	Partly	*Documentation issue can be solved with Educational Agent but development of (G)UIs is outside of scope.*
7	Organizational & cultural barriers	Partly	*Agent framework can lower pressure of effort on management decisions*
8	Trust, reliability & validation of automated/AI solutions	Partly	*Valid point. Can be partly addressed by keeping to a supportive role and providing transparency w.r.t the agent context*

This appendix presented the complete interview data from five semi-structured sessions conducted at GKN Aerospace Sweden, Trollhättan, between May 4th and May 16th, 2025. The participants, representing diverse backgrounds and expertise levels, provided insights into (i) current knowledge capture and automation practices, (ii) experiences with the ParaPy-based Turbine Rear Structure (TRS) demonstrator, (iii) the use of LLMs for coding and learning, and (iv) organizational, technical, and cultural factors influencing KBE adoption.

The thematic analysis identified eight common themes, with two emerging as most directly relevant to the proposed support system: *Skill Gap & Learning Curve* and *AI/LLM Assistance – Potential & Trust Concerns*. These themes were universally mentioned by all five participants and revealed a fundamental tension: while AI assistance is seen as necessary to flatten the steep learning curve associated with ParaPy development, participants expressed significant concerns about reliability, transparency, and the risk of engineers becoming dependent on tools they do not fully understand.

A technical assessment of current state-of-the-art LLMs, presented in section 4.3, provided empirical validation of these concerns by demonstrating the systematic failure of even advanced commercial models to generate functional ParaPy code. This assessment confirmed that the trust concerns raised during interviews are grounded in genuine limitations of current AI tools when applied to specialized frameworks.

Based on these findings, a dual-agent architecture was proposed to address both the skill gap and trust concerns. The *Educational Agent* focuses on guiding users in understanding ParaPy's architectural patterns, explaining framework conventions, and helping navigate documentation, thereby building the foundational knowledge necessary for engineers to validate AI-generated outputs. The *Developer Agent* operates in parallel to provide programmatic assistance for users with basic framework understanding, incorporating retrieval-augmented generation, explicit knowledge injection from ParaPy documentation, and structured validation workflows.

Table A.3 illustrates how this dual-agent system addresses the eight common themes identified during the interviews. The two themes most directly relevant to the research are fully addressed, while other themes are partially addressed or remain outside the scope of this work due to their organizational or infrastructure-related nature.

A.4. Consent Form

You are being invited to participate in a research study as part of the project *"Large Language Model Supported Coding Assistant for Knowlegde Based Engineering (KBE) Application Development"*. This study is carried out by Ernesto Hof (ehof@tudelft.nl) for a MSc. thesis project at the Delft University of Technology -situated in Delft, the Netherlands- under supervision of Gianfranco La Rocca (g.larocca@tudelft.nl) and Alejandro Pradas Gomez (alejandro.pradas@chalmers.se).

The purpose of this study is to capture the requirements and current challenges involved in the further implementation of KBE, specifically using the ParaPy SDKi, in combination with GenAIii/LLMiii-based tools that can support this process within the company GKN. To this end, a series of interviews will be conducted with GKN employees from various technical backgrounds. The findings of this research will be used to validate and/or refine the preliminary requirements identified in the literature review of the MSc thesis, thereby ensuring better alignment with the current needs of GKN.

The data collected during the interviews will be used for the researcher's MSc thesis project and will be anonymized in any resulting publications (e.g., the thesis report). Publications will primarily present aggregated results. Specific, anonymized quotations may be included; however, full transcripts or recordings will not be published.

To the best of our ability, your responses will be kept confidential. We will minimize any risks through the following measures:

– Interviews will be conducted on-site using offline recording tools.

– Recordings and transcripts will be processed exclusively on local devices.

– Recordings and unprocessed (i.e., identifiable) transcripts will be shared only with supervising researchers. They will not be transferred to any third parties, including web- or cloud-based services.

– Recordings and unprocessed transcripts will be permanently deleted at the conclusion of the research project (Dec 2025).

Your participation in this interview is entirely voluntary, and you may withdraw at any time. You are also free to decline to answer any individual questions. Furthermore, until 1 Sep 2025, you may: 1) request access to your data in order to (partially) rectify or delete it; 2) submit a written request to be excluded from the study, upon which all associated data (both processed and unprocessed) will be removed. After this date, preliminary results may have already been submitted, and exclusion from the study cannot be guaranteed.

Please sign below to consent to the terms of this study:

Name of participant	Date	Signature

A.5. Interview Script

KBE & LLMs in Aerospace (Engine) Design – GKN Aeroengines AB (Trollhättan)

*The following script was used to conduct the interviews at GKN Aeroengines AB in Trollhättan, Sweden between May 4th and May 16th, 2025. Text enclosed in square brackets (e.g., "[...])" denotes roles within the interview, while text enclosed in asterisks (e.g., "***...***") and marked bold, indicates actions.*

1. Introduction and Background

Q1.0. [Interviewer] Welcome and thank you for taking the time to conduct this interview. In short, this interview is aimed at getting a better understanding of the current challenges and requirements involved in the further implementation of KBE within GKN, specifically using the ParaPy Software Development Kit (which we will get to later), in combination with GenAI/LLMs-based tools that can support this process. I will also introduce myself: I am Ernesto, currently doing my MSc thesis at the Delft University of Technology (in the track of Flight Performance and Propulsion) in cooperation with GKN and/or Alex. I am in Trollhättan for 2 weeks now to gather as much information as possible and to co-develop the tool with Alex.

turn on recorder

Q1.1. [Interviewer] Have you read and do you agree to the consent form that was sent to your prior to this interview?

[Interviewee] ...

Q1.2. [Interviewer] Can you briefly introduce yourself and describe your role and involvement within the design process here at GKN?

[Interviewee] ...

Q1.3. [Interviewer] Do you have any experience with programming and/or scripting? If so, in what language(s)?

[Interviewee] ...

Q1.4. [Interviewer] Have you heard of or worked with Knowledge Based Engineering (KBE) before? If yes, how would you define or describe KBE based on your understanding?

[Interviewee] ...

Q1.5. [Interviewer] Have you used AI assistants like ChatGPT, GitHub Copilot, or similar tools before, both professionally or personally?

[Interviewee] ...

Q1.o1 *(Optional)* [Interviewer] What tools or systems do you regularly use to support your engineering work?

[Interviewee] ...

Q1.o2 *(Optional)* [Interviewer] How is expert knowledge – such as design rules, lessons learned, or best practices – typically shared or reused in your team?

[Interviewee] ...

o1. (Optional) Explanation of KBE and ParaPy

at this point, you may provide a brief introduction to KBE and ParaPy as context for the next questions [All by Interviewer]

Knowledge based engineering (KBE) is engineering using domain specific product and process knowledge stored into software applications, called KBE applications, developed using a high level programming language, called KBE language, provided by specialized software tools, called KBE.

KBE aims at improving the quality and reducing time and cost of (complex) product development by:

- Automation of repetitive and non-creative design tasks
- Enabling multidisciplinary integration in all phases of the design process
- Formalization and preservation for re-use of valuable domain specific knowledge
- ParaPy is a software platform that enables engineers to create, integrate and deploy smart KBE applications without needing proficiency in software engineering.

2. TRS App Demonstration

Q2.1. [Interviewer] Do you have any question or need further clarification on the concepts of KBE and ParaPy?

[Interviewee] ...

Q2.2. [Interviewer] What follows is a short description of a KBE application, developed using ParaPy, for the design and analysis of the turbine rear structure. A short app demonstration will follow. Based on this demonstration, you will answer questions related to the implementation of KBE and ParaPy.

[Interviewee] ...

Q2.3. [Interviewer] The TRS app is a KBE application to:

- Generate parametrized geometry from 2D blade profiles to a full 3D turbine rear structure, based on common inputs encountered in industry:
  - NACA codes, CST parametrization or coordinate files for airfoil sections;
  - Chord, sweep, twist, thickness, lean angle for 3D blade geometry;
  - The complexity of the blades can be increased by adding more blade sections along the span
- Automatically mesh the geometry for analysis in CFX. Meshing settings can be set from within the app.
- Perform RANS analysis with turbulence model of choice (selectable from within the app) using CFX from the app interface.
- Automatic processing of CFX output data, available for inspection both within CFX and by internal analysis. Analyses include:
  - The pressure, temperature, velocity and swirl profiles
  - Mass- and area-averaged pressure coefficients of the TRS
  - Decrease of swirl angle

...what follows is a short video demonstration of the app.

****show video demonstration****

Q2.4. [Interviewer] Do you have any questions regarding the app, both its design and usage?

[Interviewee] ...

3. Perception of KBE and AI

Q3.1. [Interviewer] Based on the TRS app demonstration, do you think tools like these could be useful in your current or future work? Why or why not?

[Interviewee] ...

Q3.2. [Interviewer] You will now see an isolated code snippet from the TRS app, representing the blade geometry.

****show code of Blade class and viewer next to each other****

Q3.3. [Interviewer] Given the task of adding a blade twist to the ‘Blade‘ class, i.e. rotating the whole blade about its spanwise axis, how would you do this? Please talk through your thought process as much as possible.

[Interviewee] ...

interviewee performs task

Q3.4. [Interviewer] Looking back at the task you just performed w.r.t KBE application development, would you say an AI assistant would have been helpful during the development? If yes, what kind of support would you find most useful to aid in your coding (e.g. chatbot window, code completion, documentation explanation)?

[Interviewee] ...

Q3.5. [Interviewer] What would you say was the biggest hurdle/challenge in the previous task?

[Interviewee] ...

Q3.6. [Interviewer] What do you see as the biggest technical challenges to adopting KBE in your day-to-day engineering work? Both personally and within the company?

[Interviewee] ...

Q3.o1. *(Optional)* [Interviewer] What advantages do you see in using tools that formalize or automate engineering knowledge?

[Interviewee] ...

Q3.o2. *(Optional)* [Interviewer] Do you believe AI tools could realistically support tasks such as rule creation, documentation, or design decision-making?

[Interviewee] ...

4. Barriers to Adoption and Trust

The following questions are bonus if time allows.

Q4.1. [Interviewer] What are the main challenges you foresee in adopting KBE or AI-based tools in your team or organization?

[Interviewee] ...

Q4.2. [Interviewer] How difficult do you think it is to formalize your domain expertise into structured rules or logic?

[Interviewee] ...

Q4.3. [Interviewer] Do you consider trust or validation a key issue when using AI-generated outputs in engineering? What would help increase your trust in such tools?

[Interviewee] ...

Q4.4. [Interviewer] Are there any organizational, cultural, or regulatory constraints (e.g., resistance to change, safety certification, standards compliance) that may hinder adoption of these technologies?

[Interviewee] ...

5. Closure

Q5.1. [Interviewer] Those were all the questions. If would like to thank you for your time and letting me conduct the interview. Do you have any further questions or comments?

[Interviewee] ...

turn off recorder

A.6. Prompt Example and AI-Generated ParaPy Code

All of the following code and prompts were produced at the start of the thesis project, the 26th of February, 2025. The produced output of the models was a response to the following prompt:

```
>>> In Python, I want to generate a parametric wing model representing the wing outer body
    and inner structure. Can you write a Wing class for me in Python using the ParaPy SDK?
```

Anthropic - Claude

```python
from parapy.core import *
from parapy.geom import *
import numpy as np


class WingSpar(GeomBase):
    """A class representing a wing spar structure"""

    # Position parameters
    position = Input(0.3)  # Position along chord (0.0-1.0)

    # Parent reference
    wing = Input()

    # Structural parameters
    thickness = Input(0.01)  # Spar thickness
    height_ratio = Input(0.8)  # Height ratio compared to airfoil height

    @Attribute
    def path(self):
        """The path of the spar along the wing span"""
        return self.wing.planform_curve.scale_u(self.position)

    @Part
    def solid(self):
        """The 3D representation of the spar"""
        return LoftedSolid(
            profiles=[
                Rectangle(
                    width=self.thickness,
                    height=self.wing.airfoil_height(u) * self.height_ratio,
                    position=pos,
                )
                for u, pos in zip(
                    np.linspace(0, 1, 20), self.path.equidistant_points(20)
                )
            ],
            mesh_deflection=1e-4,
        )


class WingRib(GeomBase):
    """A class representing a wing rib structure"""

    # Position parameters
    position = Input(0.5)  # Position along span (0.0-1.0)

    # Parent reference
    wing = Input()

    # Structural parameters
    thickness = Input(0.008)  # Rib thickness

    @Attribute
    def airfoil_curve(self):
        """The airfoil curve at this rib position"""
        return self.wing.airfoil_curves[
            int(self.position * (len(self.wing.airfoil_curves) - 1))
        ]

```

```
    @Part
    def solid(self):
        """The 3D representation of the rib"""
        return ExtrudedSolid(
            profile=self.airfoil_curve,
            direction=self.wing.chord_vector(self.position) * self.thickness,
            mesh_deflection=1e-4,
        )


class Wing(GeomBase):
    """A parametric wing model with internal structure"""

    # Basic wing parameters
    span = Input(36.0)  # Wing span in meters
    root_chord = Input(6.0)  # Root chord length in meters
    tip_chord = Input(1.5)  # Tip chord length in meters
    sweep_angle = Input(30.0)  # Sweep angle in degrees
    dihedral_angle = Input(5.0)  # Dihedral angle in degrees

    # Airfoil parameters
    root_airfoil = Input("NACA0015")  # Airfoil at root
    tip_airfoil = Input("NACA0012")  # Airfoil at tip
    num_sections = Input(10)  # Number of sections for internal representation

    # Structural parameters
    num_spars = Input(3)  # Number of spars
    num_ribs = Input(12)  # Number of ribs

    @Attribute
    def spar_positions(self):
        """Positions of spars along chord"""
        return np.linspace(0.15, 0.7, self.num_spars)

    @Attribute
    def rib_positions(self):
        """Positions of ribs along span"""
        return np.linspace(0, 1, self.num_ribs)

    @Attribute
    def root_position(self):
        """The position of the wing root"""
        return Position(0, 0, 0)

    @Attribute
    def tip_position(self):
        """The position of the wing tip"""
        x_offset = self.span * np.tan(np.radians(self.sweep_angle))
        z_offset = self.span * np.tan(np.radians(self.dihedral_angle))
        return Position(x_offset, self.span, z_offset)

    @Attribute
    def planform_curve(self):
        """The curve representing the wing planform (leading edge)"""
        return BSplineCurve(control_points=[self.root_position, self.tip_position])

    @Attribute
    def chord_vector(self):
        """Function returning chord vector at given span position (0-1)"""

        def chord_vector_at_position(u):
            direction = Vector(1, 0, 0)
            return direction.rotate(
                angle=np.radians(self.sweep_angle), axis=Vector(0, 0, 1)
            )

        return chord_vector_at_position

    @Attribute
    def chord_length(self):
        """Function returning chord length at given span position (0-1)"""
```

```

        def chord_at_position(u):
            return self.root_chord + u * (self.tip_chord - self.root_chord)

        return chord_at_position

    @Attribute
    def airfoil_curves(self):
        """List of airfoil curves at different span positions"""
        curves = []
        for i in range(self.num_sections):
            u = i / (self.num_sections - 1)
            # Interpolate between root and tip airfoil
            if i == 0:
                airfoil_name = self.root_airfoil
            elif i == self.num_sections - 1:
                airfoil_name = self.tip_airfoil
            else:
                # Simple linear interpolation for demonstration
                # Real implementation would interpolate airfoil coordinates
                airfoil_name = self.root_airfoil

            # Position of this section along span
            section_position = self.planform_curve.point_at(u)

            # Current chord length
            chord = self.chord_length(u)

            # Create a scaled airfoil curve at the right position
            curves.append(
                self.create_airfoil_curve(airfoil_name, section_position, chord, u)
            )

        return curves

    def create_airfoil_curve(self, airfoil_name, position, chord, u):
        """Create an airfoil curve from a NACA code"""
        # Simplified - in a real implementation, would read airfoil data from database
        # This example just creates a basic airfoil shape for demonstration

        if airfoil_name.startswith("NACA"):
            thickness = int(airfoil_name[4:]) / 100.0
        else:
            thickness = 0.12  # Default

        # Very simplified airfoil for demonstration
        points = []
        for x in np.linspace(0, 1, 50):
            # Simple thickness formula (not accurate NACA)
            t = (
                5
                * thickness
                * (
                    0.2969 * np.sqrt(x)
                    - 0.1260 * x
                    - 0.3516 * x**2
                    + 0.2843 * x**3
                    - 0.1015 * x**4
                )
            )
            if x < 0.01 or x > 0.99:  # Force sharp leading edge and trailing edge
                t = 0
            points.append(Point(x, 0, t))
            points.insert(0, Point(x, 0, -t))

        # Create a closed curve
        curve = BSplineCurve(points, degree=2)

        # Scale and position
        curve = curve.scale(chord)
        curve = curve.translate(position.vector)
```

```
        curve = curve.rotate(
            angle=np.radians(self.sweep_angle), axis=Vector(0, 0, 1), center=position
        )

        return curve

    @Attribute
    def airfoil_height(self):
        """Function returning airfoil height at given span position (0-1)"""

        def height_at_position(u):
            # Simplified approximation - would be more accurate with actual airfoil data
            thickness_root = (
                int(self.root_airfoil[4:]) / 100.0
                if self.root_airfoil.startswith("NACA")
                else 0.12
            )
            thickness_tip = (
                int(self.tip_airfoil[4:]) / 100.0
                if self.tip_airfoil.startswith("NACA")
                else 0.12
            )
            thickness = thickness_root + u * (thickness_tip - thickness_root)
            return self.chord_length(u) * thickness

        return height_at_position

    @Part
    def spars(self):
        """The wing spars"""
        return WingSpar(
            wing=self,
            position=position,
            quantify=len(self.spar_positions),
            position_index=lambda i: self.spar_positions[i],
        )

    @Part
    def ribs(self):
        """The wing ribs"""
        return WingRib(
            wing=self,
            position=position,
            quantify=len(self.rib_positions),
            position_index=lambda i: self.rib_positions[i],
        )

    @Part
    def skin(self):
        """The wing skin (outer surface)"""
        return LoftedSolid(profiles=self.airfoil_curves, mesh_deflection=1e-4)

    # Optional visualization helpers
    @Part
    def wing_box(self):
        """Wing box (space between front and rear spar)"""
        return LoftedSolid(
            profiles=[
                section.solid.intersection(
                    self.spars[0].solid.union(self.spars[-1].solid)
                )
                for section in self.ribs
            ],
            mesh_deflection=1e-4,
            hidden=True,  # Hidden by default for cleaner visualization
        )


if __name__ == "__main__":
    from parapy.gui import display

```

```
    # Create a wing with default parameters
    wing = Wing()

    # Display the wing
    display(wing)
```

Deepseek

```
from parapy.geom import *
from parapy.core import *


class Wing(GeomBase):
    # Wing parameters
    span = Input(10.0)  # Wing span
    root_chord = Input(2.0)  # Root chord length
    tip_chord = Input(1.0)  # Tip chord length
    sweep_angle = Input(20.0)  # Sweep angle in degrees
    dihedral_angle = Input(5.0)  # Dihedral angle in degrees
    thickness_ratio = Input(0.12)  # Thickness ratio of the airfoil

    # Inner structure parameters
    num_spars = Input(2)  # Number of spars
    num_ribs = Input(5)  # Number of ribs
    spar_thickness = Input(0.02)  # Thickness of the spars
    rib_thickness = Input(0.01)  # Thickness of the ribs

    @Attribute
    def airfoil_points(self):
        # Define a simple symmetric airfoil (NACA 0012)
        # This is a simplified representation for demonstration purposes
        return [
            Point(0, 0, 0),
            Point(0.5, 0.06, 0),
            Point(1, 0, 0),
            Point(0.5, -0.06, 0),
            Point(0, 0, 0),
        ]

    @Part
    def outer_surface(self):
        return LoftedShell(
            profiles=[
                Translated(
                    Scaled(
                        Rotated(
                            PolygonalFace(self.airfoil_points),
                            angle=self.sweep_angle,
                            vector=Vector(1, 0, 0),
                        ),
                        factor=(self.root_chord, self.thickness_ratio, 1),
                    ),
                    vector=Vector(0, 0, 0),
                ),
                Translated(
                    Scaled(
                        Rotated(
                            PolygonalFace(self.airfoil_points),
                            angle=self.sweep_angle,
                            vector=Vector(1, 0, 0),
                        ),
                        factor=(self.tip_chord, self.thickness_ratio, 1),
                    ),
                    vector=Vector(
                        0, self.span, self.span * tan(radians(self.dihedral_angle))
                    ),
                ),
            ]
        )

    @Part
```

```
    def spars(self):
        return [
            ExtrudedSurface(
                profile=Polygon(
                    points=[
                        Point(
                            i * self.root_chord / (self.num_spars + 1),
                            0,
                            -self.thickness_ratio * self.root_chord / 2,
                        ),
                        Point(
                            i * self.tip_chord / (self.num_spars + 1),
                            self.span,
                            -self.thickness_ratio * self.tip_chord / 2,
                        ),
                        Point(
                            i * self.tip_chord / (self.num_spars + 1),
                            self.span,
                            self.thickness_ratio * self.tip_chord / 2,
                        ),
                        Point(
                            i * self.root_chord / (self.num_spars + 1),
                            0,
                            self.thickness_ratio * self.root_chord / 2,
                        ),
                    ]
                ),
                vector=Vector(0, 0, self.spar_thickness),
            )
            for i in range(1, self.num_spars + 1)
        ]

    @Part
    def ribs(self):
        return [
            ExtrudedSurface(
                profile=Polygon(
                    points=[
                        Point(
                            0,
                            j * self.span / (self.num_ribs + 1),
                            -self.thickness_ratio * self.root_chord / 2,
                        ),
                        Point(
                            self.root_chord,
                            j * self.span / (self.num_ribs + 1),
                            -self.thickness_ratio * self.root_chord / 2,
                        ),
                        Point(
                            self.root_chord,
                            j * self.span / (self.num_ribs + 1),
                            self.thickness_ratio * self.root_chord / 2,
                        ),
                        Point(
                            0,
                            j * self.span / (self.num_ribs + 1),
                            self.thickness_ratio * self.root_chord / 2,
                        ),
                    ]
                ),
                vector=Vector(0, 0, self.rib_thickness),
            )
            for j in range(1, self.num_ribs + 1)
        ]
```

GitHub - Copilot

```
from parapy.core import Base, Input, Part
from parapy.geom import Fused, LoftedSolid, Rectangle, RotatedShape, TranslatedShape


```

```python
class Wing(Base):
    span = Input(10.0)  # Wing span
    chord_root = Input(2.0)  # Root chord length
    chord_tip = Input(1.0)  # Tip chord length
    thickness = Input(0.1)  # Wing thickness

    @Part
    def root_airfoil(self):
        return Rectangle(width=self.chord_root, height=self.thickness)

    @Part
    def tip_airfoil(self):
        return TranslatedShape(
            shape_in=Rectangle(width=self.chord_tip, height=self.thickness),
            displacement=(self.span, 0, 0),
        )

    @Part
    def wing_body(self):
        return LoftedSolid(profiles=[self.root_airfoil, self.tip_airfoil])

    @Part
    def spar(self):
        return RotatedShape(
            shape_in=Rectangle(width=self.thickness, height=self.span),
            rotation_point=(0, 0, 0),
            vector=(0, 1, 0),
            angle=90,
        )

    @Part
    def wing_structure(self):
        return Fused([self.wing_body, self.spar])
```

Mistral AI - Le Chat

```python
from parapy.core import *
from parapy.geom import *
from parapy.exchange import *


class Wing(GeomBase):
    # Define the parameters for the wing
    span = Input(15.0, validate=positive)  # Wing span in meters
    root_chord = Input(3.0, validate=positive)  # Root chord length in meters
    tip_chord = Input(1.5, validate=positive)  # Tip chord length in meters
    sweep_angle = Input(20.0, validate=positive)  # Sweep angle in degrees
    dihedral_angle = Input(5.0)  # Dihedral angle in degrees
    airfoil_thickness = Input(0.12)  # Airfoil thickness as a fraction of the chord

    @Attribute
    def airfoil(self):
        # Define a simple airfoil shape using a NACA 4-digit series
        return NACA4(self.airfoil_thickness)

    @Part
    def wing_surface(self):
        # Create the wing surface using the defined parameters
        return LoftedSolid(
            profiles=[
                self.airfoil.scaled(self.root_chord),
                self.airfoil.scaled(self.tip_chord).translated(
                    x=self.span / 2, z=self.span / 2 * tan(radians(self.dihedral_angle))
                ),
            ],
            rulings=[
                LineSegment(
                    start=Point(0, 0, 0),
                    end=Point(
                        self.span / 2,
                        0,
```

```
                        self.span / 2 * tan(radians(self.dihedral_angle)),
                    ),
                ).rotated(self.sweep_angle)
            ],
        )

    @Part
    def spar(self):
        # Define a simple spar running along the span of the wing
        return Box(length=self.span, width=0.1, height=0.1).translated(
            x=self.root_chord / 4, z=0.05
        )

    @Part
    def ribs(self):
        # Define ribs along the span of the wing
        ribs = []
        num_ribs = 10
        for i in range(num_ribs):
            position = self.span * i / (num_ribs - 1)
            rib = self.airfoil.scaled(
                self.root_chord
                - (self.root_chord - self.tip_chord) * i / (num_ribs - 1)
            )
            ribs.append(
                rib.translated(
                    x=position, z=position * tan(radians(self.dihedral_angle))
                )
            )
        return ribs
```

OpenAI - ChatGPT

```
from parapy.core import Base, Input, Part
from parapy.geom import Fused, LoftedSolid, LineSegment, Plane
from parapy.exchange.step import STEPWriter


class Wing(Base):
    """A parametric wing model with an aerodynamic outer shape and internal structure."""

    span = Input(10.0)  # Wing span in meters
    root_chord = Input(2.0)  # Root chord length in meters
    tip_chord = Input(1.0)  # Tip chord length in meters
    airfoil_root = Input("NACA0012")  # Root airfoil profile
    airfoil_tip = Input("NACA0012")  # Tip airfoil profile

    @Part
    def wing_surface(self):
        """Defines the aerodynamic wing outer shape."""
        return LoftedSolid(
            profiles=[
                self.create_airfoil(self.airfoil_root, self.root_chord, (0, 0, 0)),
                self.create_airfoil(
                    self.airfoil_tip, self.tip_chord, (self.span, 0, 0)
                ),
            ]
        )

    @Part
    def front_spar(self):
        """Defines a front spar as an internal structure element."""
        return LineSegment(
            start=(0, 0.2 * self.root_chord, 0),
            end=(self.span, 0.2 * self.tip_chord, 0),
        )

    @Part
    def rear_spar(self):
        """Defines a rear spar as an internal structure element."""
        return LineSegment(
```

```
            start=(0, 0.7 * self.root_chord, 0),
            end=(self.span, 0.7 * self.tip_chord, 0),
        )

    @Part
    def wing_structure(self):
        """Combines all structural elements."""
        return Fused(children=[self.front_spar, self.rear_spar])

    def create_airfoil(self, airfoil_name, chord, position):
        """Creates an airfoil profile at a given position."""
        # For simplicity, this function could be expanded to read airfoil coordinates
        return Plane(position=position)

    def export_to_step(self, filename="wing_model.step"):
        """Exports the wing geometry to a STEP file."""
        writer = STEPWriter(self.wing_surface)
        writer.write(filename)
```

B

Supporting Material: User Testing

B.1. Data Collection and Processing

The data obtained from the user testing sessions consisted of two primary components: (1) the sessions themselves, which involved screen and audio recordings that were subsequently transcribed and analysed, and (2) the analysis of the (semi-)completed test cases to extract key performance metrics.

The transcribed recordings were examined to identify recurring themes and notable observations that could help interpret—or potentially challenge—the numerical results. This qualitative dataset includes observations from the sessions as well as participants' responses to structured interview questions. Each response was assigned one or more thematic codes, which are presented alongside the corresponding quantitative results to provide contextual explanation.

The quantitative data includes metrics extracted from the (semi-)completed test case files and participants' development processes. As outlined in Chapter 5, this data comprises both time-based measures—such as execution time and debugging time—and code productivity metrics. Productivity is evaluated based on the number of added lines of code and added features (where a feature is defined as a class subcomponent—e.g., method, attribute, or part—that executes without error, regardless of full functionality). Additional metrics include the number of ParaPy-specific errors encountered (e.g., part grammar violations), automated code quality scores (from the quality assessment framework), and functional correctness as compared to the expert solution. A detailed breakdown of the functional correctness scoring for both test cases is presented in the appendix in section B.4.

B.2. Consent Form

You are being invited to participate in testing sessions as part of the project *"Large Language Model Supported Coding Assistant for Knowlegde Based Engineering (KBE) Application Development"*. This study is carried out by Ernesto Hof (ehof@tudelft.nl) for a MSc. thesis project at the Delft University of Technology -situated in Delft, the Netherlands- under supervision of Gianfranco La Rocca (g.larocca@tudelft.nl) and Alejandro Pradas Gomez (alejandro.pradas@chalmers.se).

The purpose of the user testing is to assess whether the developed GenAI-assisted tool demonstrably improves performance in the development of KBE applications using the ParaPy SDK, for both novice and expert industry users. To this end, a series of user testing sessions will be conducted with employees from GKN Aerospace Sweden and ParaPy B.V., representing a range of technical backgrounds. The findings from these sessions will be used to evaluate whether the requirements outlined in the research introduction have been met, and to determine whether the thesis has (partially) addressed the identified research gap through the development of a successful product.

The data collected during the interviews will be used for the researcher's MSc thesis project and will be anonymized in any resulting publications (e.g., the thesis report). Publications will primarily present aggregated results. Specific, anonymized quotations may be included; however, full transcripts or recordings will not be published.

To the best of our ability, your responses will be kept confidential. We will minimize any risks through the following measures:

– Interviews will be conducted on-site using offline recording tools whenever possible but will take place via MS Teams when remote participation is the only option.

– Recordings and transcripts will be processed exclusively on local devices.

– Recordings and unprocessed (i.e., identifiable) transcripts will be shared only with supervising researchers if necessary. They will not be transferred to any third parties, including web- or cloud-based services.

– Recordings and unprocessed transcripts will be permanently deleted at the conclusion of the research project (Dec 2025).

Your participation in this interview is entirely voluntary, and you may withdraw at any time. You are also free to decline to answer any individual questions. Furthermore, until 24 Oct 2025, you may: 1) request access to your data in order to (partially) rectify or delete it; 2) submit a written request to be excluded from the study, upon which all associated data (both processed and unprocessed) will be removed. After this date, preliminary results may have already been submitted, and exclusion from the study cannot be guaranteed.

Please sign below to consent to the terms of this study:

Name of participant	Date	Signature

B.3. User Testing Script

LLM assisted KBE application development – ParaPy Copilot – GKN Aeroengines AB (Trollhättan, SE) | ParaPy B.V. (Delft, NL)

*The following script was used to conduct the user testing sessions at GKN Aeroengines AB and ParaPy B.V. between October 6th and October 10th, 2025. Text enclosed in square brackets (e.g., "[...])" denotes roles within the interview, while text enclosed in asterisks (e.g., "***...***") and marked bold, indicates actions.*

1. Introduction and Background

Q1.0. [Interviewer] Welcome and thank you for taking the time to participate in the user testing of my MSc thesis. In short, these user testing sessions are aimed at evaluating whether the goals set in the research, as well as the research gap identified, have been (partially) met by the developed copilot tool. As outlined in the research, the main goal of the copilot tool is to assist both novice and expert users in the development of KBE applications using the ParaPy SDK.

turn on recorder

Q1.1. [Interviewer] Have you read, and do you agree to the consent form that was sent to you prior to this interview?

[Interviewee] ...

Q1.2. [Interviewer] Can you briefly introduce yourself and describe your role within GKN/ParaPy?

[Interviewee] ...

Q1.3. [Interviewer] What is your experience with programming and/or scripting? Would you describe yourself more as a novice or expert when it comes to (KBE) programming? What languages do you have experience in? What experience do you have with ParaPy?

[Interviewee] ...

Q1.4. [Interviewer] Do you have experience with CAD and/or CAE programs? If so, to what extend?

[Interviewee] ...

Q1.5. [Interviewer] Have you used AI assistants like ChatGPT, GitHub Copilot, or similar tools before, both professionally and personally?

[Interviewee] ...

Q1.o1. *(For people who were already interviewed in May)* [Interviewer] Based on the above questions, have any of these aspects changed significantly since our last session in May? Have you adopted KBE more in your day-to-day engineering work?

[Interviewee] ...

2. Tool Introduction

Q2.1. [Interviewer] We will move on now to the introduction of the copilot tool. Firstly, have you followed the installation instructions that were sent to you and is the tool ready to use?

[Interviewee] ...

install tool if not done (see installation instructions)

Q2.2. [Interviewer] The developed copilot tool is an AI-assisted framework that is designed to assist developers in KBE application development using the ParaPy SDK. The framework is user accessible through a CLI interface and features two agent modes: a Developer agent tasked specifically with code suggestion/completion/debug requests, and an Educational agent aimed at guiding the user through the ParaPy documentation and providing more theoretical background on all things KBE/ParaPy development related. You will have access to the tool with pre-defined settings. Although it is not recommended, you are free to alter the model and model settings. Furthermore, you have several commands at your disposal (these will be repeated in the CLI as well):

- `/restart`: restart the application from scratch
- `/model`: change the model powering the agent
- `/mode`: switch mode (educational ↔ developer)
- `/settings`: change the model settings
- `/file` or `/f`: append file content to the prompt/context
- `/clear` or `/c`: clear the file content from the prompt; file contents are persisted during sessions
- `/help`: show available commands
- `‘exit‘` or `‘quit‘`: quit the application

Important to note is that the agent uses a one-time pipeline, so there is no message history saved within sessions. Do you have any questions regarding the tool and its usage?

[Interviewee] ...

Q2.3. [Interviewer] You will now get the task of completing two KBE application cases from minimal skeleton code. Based on which group you are in (A or B), you will either start with case 1 or case 2 without AI assistance, after which you will complete the remaining case with AI assistance. You will get 20 minutes for each task. The cases are not designed to be completed in 20 minutes, so do not worry if you cannot complete all features. Furthermore, I recommend having the ParaPy documentation open in your browser as well. During testing, I will be available for question not related to the implementation of the particular user case.

Do you have any questions about the test cases and/or the task at hand?

[Interviewee] ...

****Proceed with both test cases with a small break in between but without asking questions****

3. Core Testing

Q3.1. [Interviewer] In one sentence, how would you describe your experience using the copilot tool versus coding manually?

[Interviewee] ...

Q3.2. [Interviewer] Which test case did you find most challenging, and why?

[Interviewee] ...

Q3.3. [Interviewer] When using the copilot, did the suggestions align with what you intended to write? Can you give a specific example?

[Interviewee] ...

Q3.4. [Interviewer] Did you feel the need to verify or modify the AI suggestions? How often, would you say – rarely, sometimes, or frequently?

[Interviewee] ...

Q3.5. [Interviewer] On a scale from 1-10, and based on your experience with programming and ParaPy, what grade would you give the generated code of the agent?

[Interviewee] . . .

Q3.6. [Interviewer] For the (aerospace) components specifically, do you feel the AI understood the domain requirements? What worked well or poorly?

[Interviewee] ...

Q3.7. [Interviewer] Based on today’s experience, in what situations would you prefer using the copilot versus coding manually?

[Interviewee] ...

Q3.8. [Interviewer] As someone [new to/experienced with] ParaPy, did the AI help you learn new patterns or did it hinder your understanding?

[Interviewee] ...

Q3.9. [Interviewer] What is the one thing you would change about the copilot to make it more useful for (aerospace/your company) KBE development?

[Interviewee] ...

4. Closure

Q4.1. [Interviewer] Those were all the questions. I would like to thank you for your time and for letting me conduct this interview. Do you have any further questions or comments?

[Interviewee] ...

turn off recorder

B.4. Functional Breakdown of Test Cases

TC1 - Aircraft Mounting Bracket

Table B.1: Scoring Rubric for functional correctness of TC1. Each criterion is evaluated on a binary pass/fail basis, receiving either 0 or 1 point. The resulting scores are then weighted to calculate the total score.

Subcomponent	Criteria	Weight [%]
Base Flange	Correct use of geometrical primitives and operations	6.67
	Correctly dimensioned and/or positioned	6.67
	Correct application of 2.5D rule	6.67
Rib	Correct use of geometrical primitives and operations	6.67
	Correctly dimensioned and/or positioned	6.67
	Correct application of 2.5D rule	6.67
Mounting Holes	Correctly dimensioned and/or positioned	6.67
	Correct application of 2.5D rule	6.67
	Correctly implemented holes as tools for boolean operation	6.67
Lightning Holes	Correct dimension (derived)	6.67
	Correct positions	6.67
	Correct application of 2.5D rule	6.67
	Correctly implemented holes as tools for boolean operation	6.67
Boolean Solid	Correct base shape employed	10
	Correct tools employed	10
	Total Score	**100**

TC2 - Y-Pipe Connector

Table B.2: Scoring Rubric for functional correctness of TC2. Each criterion is evaluated on a binary pass/fail basis, receiving either 0 or 1 point. The resulting scores are then weighted to calculate the total score.

Subcomponent	Criteria	Weight [%]
Correct Outer Diameter (derived)		5
Thin Walled Pipe	Correct use of geometrical primitives and operations	9.5
	Correctly dimensioned	9.5
Inlet Pipe	Correctly dimensioned	9.5
	Correctly positioned	9.5
Outlet Pipe 1	Correctly dimensioned	9.5
	Correctly positioned	9.5
Outlet Pipe 2	Correctly dimensioned	9.5
	Correctly positioned	9.5
Boolean Solid	Correctly cut pipes into shape	6.3
	Correct base shape employed	6.3
	Correct tools employed	6.3
	Total Score	**100**

C
Expert Grading Rubric

Code Quality Grading Rubric for Generated ParaPy Code

Submission Size: 200–400 lines
Target Quality: 7.5/10 average
Evaluation Time: 10–15 minutes
Evaluator: ParaPy Domain Expert

Thank you for taking the time to evaluate these code snippets for my MSc thesis.
Your expertise in programming and familiarity with ParaPy will significantly contribute to the assessment of the agent framework's performance.

Instructions

You will be provided with the following materials:

1. An expert-developed ParaPy application (for reference);
2. Code generated solely from natural language prompts, based on the expert application (the original prompt is included in the file);
3. Code generated from a pre-defined skeleton, derived from the expert-developed code (you will receive the skeleton code as well).

For each code example:

1. **Run the code** and compare its output and behavior to the reference implementation;
2. **Score each section** by selecting the appropriate score within the 0–10 range;
3. **Document major issues** in the space provided;
4. **Record the final scores** in the summary table.

You don't need to document every minor issue — just assign scores based on overall impression within each category.

C.1. Semantic Correctness

Does the code work correctly and use ParaPy properly?

Functional Equivalence & Framework Usage

Circle your score:

Score	Criteria
9–10	✓ Produces identical results to reference solution ✓ Handles all parametrization correctly ✓ Correct ParaPy framework usage throughout ✓ No functional errors
7–8	✓ Correct for most components ✓ Minor issues with 1–2 (edge) cases ✓ Generally correct ParaPy usage with minor mistakes ✓ Overall behavior is correct
5–6	✓ Works for simple components ! Fails on several other components ! Some ParaPy framework misuse ! Some incorrect outputs or logic errors
3–4	! Incorrect outputs for many components ! Significant ParaPy framework misuse ! Significant functional problems ! Missing key functionality
0–2	× Does not work / crashes × Completely wrong behavior × Fundamental ParaPy misunderstandings × Missing most functionality

Section 1 Score: ____________ / 10

ParaPy Framework Check (if applicable)

Check correct usage of:

`@Input` — Defines configurable parameters
`@Attribute` — Used for computed/cached values
`@Part` — Creates child components properly
`Base` / `GeomBase` — Inherits appropriately
`quantify()` — Used correctly for collections
`child.` references — Proper parent-child relationships
`DynamicType` — Correct dynamic instantiation

Major issues:

C.2. Maintainability

Is the code readable, organized, and maintainable?

Code Quality & Structure

Circle your score:

Score	Criteria
9–10	✓ Clean, professional code ✓ Clear structure and naming ✓ Simple, elegant solutions ✓ Well-documented with docstrings ✓ Low complexity, easy to understand
7–8	✓ Generally clear and readable ✓ Good organization and naming ✓ Reasonable complexity ✓ Adequate documentation
5–6	! Somewhat unclear or messy ! Inconsistent naming or structure ! Higher complexity than needed ! Minimal documentation
3–4	! Difficult to understand ! Poor organization ! Very complex/convoluted ! Little to no documentation
0–2	× Incomprehensible × Chaotic structure × Unmaintainable complexity × No documentation

Section 2 Score: ___________ / 10

Red Flags (check if present)

Functions/methods > 50 lines
Nesting depth > 3 levels
Unclear variable names (`x`, `temp`, `data1`)
No docstrings on classes or key methods
Duplicate code blocks
Overly complex logic

Major issues:

C.3. PEP-8 Compliance

Does the code follow Python style conventions?

Style & Formatting

Circle your score:

Score	Typical Violations	Quality
9–10	0–5 violations	Excellent — Professional style
7–8	6–15 violations	Good — Minor style issues
5–6	16–30 violations	Adequate — Several style problems
3–4	31–50 violations	Poor — Many style issues
0–2	50+ violations	Unacceptable — No style adherence

Section 3 Score: ___________ / 10

Common Issues (check if widespread)

Wrong naming conventions (classes, functions)
Lines too long (>88 characters)
Inconsistent indentation or spacing
Missing blank lines between classes/functions
Poor import organization
Whitespace issues (`x=1+2` vs `x = 1 + 2`)

Major issues:

Score Summary

Section	Score (0–10)
Semantic Correctness	________________
Maintainability	________________
PEP-8 Compliance	________________

Overall Assessment

Key Strengths:

Key Weaknesses:

Would you use this code in production?

Yes, as-is
Yes, with minor fixes
No, needs significant revision

Submission Information

Submission ID: ________________________________

Evaluator Name: ________________________________

Date: ________________________________

Evaluation Time: __________ minutes

Appendix: Quick Reference

Semantic Correctness Red Flags

- Crashes or exceptions during execution
- Wrong outputs for test cases
- Missing required functionality
- Incorrect ParaPy framework usage (`@Attribute`, `@Part`)
- Undefined variables or attributes

Maintainability Red Flags

- Functions longer than 50 lines
- Nesting depth > 3 levels
- No docstrings
- Unclear variable names (`x`, `temp`, `data`)
- Duplicate code blocks
- "Magic numbers" without explanation

PEP-8 Red Flags

- Inconsistent indentation
- Lines > 100 characters
- Wrong naming conventions (e.g., `myFunction`, `MyVariable`)
- Missing spaces around operators
- Disorganized imports
- Trailing whitespace

Evaluation Tips

1. **Run the code first** — Does it work?
2. **Compare to reference** — Is it functionally equivalent?
3. **Skim for structure** — Is it organized?
4. **Check complexity** — Is it simple enough?
5. **Scan for style** — Does it look professional?

Time-Saving Approach

- Spend 40% of time on semantic correctness (most important)
- Spend 35% on maintainability (scan for readability)
- Spend 25% on PEP-8 (quick style check)

D

Extended Results of Automated Evaluations

D.1. Run Usage

Developer Agent

Table D.1: Performance metrics for Developer Agent across test cases.

Case ID	Model	Input Tokens	Output Tokens	Runtime (s)	Requests	Tool Calls
D-1	Claude Sonnet 4	152,360	3,530	102.00	19.0	18.0
D-2	Claude Sonnet 4	108,790	6,528	102.00	13.0	12.0
D-3	Claude Sonnet 4	109,648	6,695	120.00	12.0	10.0
D-4	Claude Sonnet 4	263,896	15,805	195.00	18.0	14.0
D-5	Claude Sonnet 4	240,996	12,470	185.00	17.0	14.0
D-6	Claude Sonnet 4	176,758	9,696	120.00	nan	17.0
D-7	Claude Sonnet 4	315,076	11,630	186.00	21.0	18.0
D-8	Claude Sonnet 4	191,484	18,887	222.00	15.0	12.0
D-9	Claude Sonnet 4	241,613	20,431	231.00	18.0	14.0
D-10	Claude Sonnet 4	271,205	16,374	235.00	21.0	17.0
D-11	Claude Sonnet 4	237,834	40,094	478.00	nan	15.0
D-12	Claude Sonnet 4	122,285	12,129	131.00	nan	11.0
D-13	Claude Sonnet 4	71,960	4,191	81.80	9.0	8.0
D-14	Claude Sonnet 4	172,317	17,870	218.00	nan	9.0
D-15	Claude Sonnet 4	199,167	17,366	228.00	nan	16.0
D-16	Claude Sonnet 4	411,401	20,789	303.00	nan	12.0
D-17	Claude Sonnet 4	198,092	15,141	186.00	17.0	13.0
D-18	Claude Sonnet 4	57,098	2,030	53.50	8.0	6.0
D-19	Claude Sonnet 4	336,693	13,353	218.00	25.0	21.0
D-20	Claude Sonnet 4	544,626	3,268	127.00	50.0*	50.0*
D-21	Claude Sonnet 4	126,064	14,242	155.00	10.0	7.0
D-22	Claude Sonnet 4	138,356	15,146	181.00	11.0	8.0
D-23	Claude Sonnet 4	132,182	7,581	136.00	11.0	8.0
D-24	Claude Sonnet 4	147,497	13,931	188.00	12.0	9.0
D-25	Claude Sonnet 4	22,995	3,456	65.70	4.0	3.0
D-26	Claude Sonnet 4	79,543	3,749	75.60	11.0	10.0
D-27	Claude Sonnet 4	39,087	1,449	44.80	6.0	5.0
D-28	Claude Sonnet 4	69,230	1,589	56.20	10.0	9.0
D-29	Claude Sonnet 4	55,272	1,668	48.80	8.0	7.0
D-30	Claude Sonnet 4	79,485	4,897	86.40	10.0	9.0

Continued on next page

Table D.1 – *continued from previous page*

Case ID	Model	Input Tokens	Output Tokens	Runtime (s)	Requests	Tool Calls
D-31	Claude Sonnet 4	40,640	1,576	49.50	6.0	5.0
D-32	Claude Sonnet 4	16,469	1,029	35.50	3.0	2.0
D-33	Claude Sonnet 4	100,028	2,023	69.70	12.0	11.0
D-34	Claude Sonnet 4	102,197	2,834	68.40	13.0	11.0
D-35	Claude Sonnet 4	30,182	1,651	41.80	5.0	4.0
D-36	Claude Sonnet 4	60,282	2,158	64.80	9.0	8.0
D-1	LLaMa 3.1 - 8B	46,180	948	24.00	7.0	4.0
D-2	LLaMa 3.1 - 8B	42,717	1,657	52.80	6.0	2.0
D-3	LLaMa 3.1 - 8B	22,581	566	21.30	nan	4.0
D-4	LLaMa 3.1 - 8B	28,919	735	32.20	5.0	3.0
D-5	LLaMa 3.1 - 8B	24,563	855	13.80	nan	4.0
D-6	LLaMa 3.1 - 8B	22,554	576	23.20	nan	4.0
D-7	LLaMa 3.1 - 8B	513,033	1,390	47.10	nan	50.0*
D-8	LLaMa 3.1 - 8B	15,902	526	11.30	3.0	2.0
D-9	LLaMa 3.1 - 8B	22,551	570	21.40	nan	4.0
D-10	LLaMa 3.1 - 8B	24,326	372	21.50	nan	4.0
D-11	LLaMa 3.1 - 8B	16,520	963	12.50	3.0	2.0
D-12	LLaMa 3.1 - 8B	903,703	1,888	197.00	nan	14.0
D-13	LLaMa 3.1 - 8B	11,831	513	9.05	nan	2.0
D-14	LLaMa 3.1 - 8B	64,273	4,553	42.80	nan	2.0
D-15	LLaMa 3.1 - 8B	0	0	3.10	nan	0.0
D-16	LLaMa 3.1 - 8B	28,857	794	35.30	5.0	3.0
D-17	LLaMa 3.1 - 8B	31,926	662	23.70	5.0	3.0
D-18	LLaMa 3.1 - 8B	16,157	445	14.80	3.0	2.0
D-19	LLaMa 3.1 - 8B	55,424	642	15.50	10.0	9.0
D-20	LLaMa 3.1 - 8B	10,503	280	17.50	nan	2.0
D-21	LLaMa 3.1 - 8B	10,512	334	7.81	nan	2.0
D-22	LLaMa 3.1 - 8B	42,259	1,316	136.00	7.0	2.0
D-23	LLaMa 3.1 - 8B	0	0	0.80	nan	0.0
D-24	LLaMa 3.1 - 8B	31,715	528	17.70	5.0	3.0
D-25	LLaMa 3.1 - 8B	10,402	122	5.78	nan	2.0
D-26	LLaMa 3.1 - 8B	10,706	263	6.25	nan	2.0
D-27	LLaMa 3.1 - 8B	42,892	800	31.40	7.0	4.0
D-28	LLaMa 3.1 - 8B	462,768	1,122	33.90	nan	50.0*
D-29	LLaMa 3.1 - 8B	85,487	1,138	38.70	nan	13.0
D-30	LLaMa 3.1 - 8B	47,594	840	29.50	8.0	7.0
D-31	LLaMa 3.1 - 8B	441,822	1,296	32.90	nan	50.0*
D-32	LLaMa 3.1 - 8B	420,887	896	64.70	14.0	12.0
D-33	LLaMa 3.1 - 8B	158,698	24,576	35.00	nan	nan
D-34	LLaMa 3.1 - 8B	97,155	2,228	34.30	nan	9.0
D-35	LLaMa 3.1 - 8B	88,870	2,106	34.20	nan	9.0
D-36	LLaMa 3.1 - 8B	66,662	1,523	28.70	nan	6.0

* – These extremities were results of failed runs and as such are excluded from the data processing.

Educational Agent

Table D.2: Performance metrics for Educational Agent across test cases.

Case ID	Model	Input Tokens	Output Tokens	Runtime (s)	Requests	Tool Calls
E-1	Claude Sonnet 4	12,884	1,426	30.10	3.0	4.0

Continued on next page

Table D.2 – *continued from previous page*

Case ID	Model	Input Tokens	Output Tokens	Runtime (s)	Requests	Tool Calls
E-2	Claude Sonnet 4	2,970	1,827	30.90	1.0	0.0
E-3	Claude Sonnet 4	19,984	1,591	32.10	3.0	4.0
E-4	Claude Sonnet 4	25,430	1,442	31.70	4.0	6.0
E-5	Claude Sonnet 4	18,561	1,829	36.60	3.0	4.0
E-6	Claude Sonnet 4	26,575	1,469	33.40	4.0	6.0
E-7	Claude Sonnet 4	20,572	1,236	29.90	4.0	4.0
E-8	Claude Sonnet 4	28,970	1,660	38.90	4.0	6.0
E-9	Claude Sonnet 4	17,379	1,450	30.20	3.0	4.0
E-10	Claude Sonnet 4	17,855	1,605	31.40	3.0	4.0
E-11	Claude Sonnet 4	19,749	1,473	32.40	3.0	4.0
E-12	Claude Sonnet 4	17,768	1,450	31.70	3.0	5.0
E-13	Claude Sonnet 4	16,655	1,152	23.90	3.0	4.0
E-14	Claude Sonnet 4	16,213	1,325	29.80	3.0	5.0
E-15	Claude Sonnet 4	16,098	1,577	31.50	3.0	5.0
E-16	Claude Sonnet 4	15,484	752	21.30	4.0	3.0
E-17	Claude Sonnet 4	22,740	1,722	33.90	4.0	5.0
E-18	Claude Sonnet 4	29,341	914	23.80	4.0	6.0
E-19	Claude Sonnet 4	11,190	1,111	24.20	3.0	3.0
E-20	Claude Sonnet 4	18,044	2,072	40.80	3.0	5.0
E-21	Claude Sonnet 4	23,228	1,240	29.20	4.0	6.0
E-22	Claude Sonnet 4	14,613	1,076	26.70	3.0	3.0
E-23	Claude Sonnet 4	20,907	1,968	41.40	3.0	6.0
E-1	LLaMa 3.1 - 8B	0	0	0.54	0.0	0.0
E-2	LLaMa 3.1 - 8B	7,354	233	1.28	2.0	1.0
E-3	LLaMa 3.1 - 8B	6,416	311	1.18	2.0	1.0
E-4	LLaMa 3.1 - 8B	3,073	13	1.10	nan	1.0
E-5	LLaMa 3.1 - 8B	6,249	126	1.03	2.0	1.0
E-6	LLaMa 3.1 - 8B	3,077	180	0.64	1.0	0.0
E-7	LLaMa 3.1 - 8B	7,176	285	1.37	2.0	1.0
E-8	LLaMa 3.1 - 8B	3,083	13	1.02	nan	1.0
E-9	LLaMa 3.1 - 8B	0	0	0.30	0.0	0.0
E-10	LLaMa 3.1 - 8B	3,080	18	1.25	nan	1.0
E-11	LLaMa 3.1 - 8B	0	0	0.47	0.0	0.0
E-12	LLaMa 3.1 - 8B	3,074	38	1.64	nan	1.0
E-13	LLaMa 3.1 - 8B	0	0	0.73	0.0	0.0
E-14	LLaMa 3.1 - 8B	6,281	150	1.09	2.0	1.0
E-15	LLaMa 3.1 - 8B	7,152	205	1.19	2.0	1.0
E-16	LLaMa 3.1 - 8B	6,432	224	1.15	2.0	1.0
E-17	LLaMa 3.1 - 8B	0	0	0.99	0.0	0.0
E-18	LLaMa 3.1 - 8B	3,085	13	0.94	nan	1.0
E-19	LLaMa 3.1 - 8B	3,085	40	1.45	nan	1.0
E-20	LLaMa 3.1 - 8B	7,261	194	1.29	2.0	1.0
E-21	LLaMa 3.1 - 8B	3,115	283	1.01	1.0	0.0
E-22	LLaMa 3.1 - 8B	0	0	1.01	0.0	0.0
E-23	LLaMa 3.1 - 8B	3,092	182	0.86	1.0	0.0

D.2. Agent Performance Across Evaluation Criteria

Developer Agent

Table D.3: Evaluation results for Developer Agent.

Case ID	Model	isinstance	Syntax	Runtime	Quality	GPT OSS	Llama 3.3	Qwen3
D-7	Claude Sonnet 4	✓	✓	✓	8.2	4.5	4.2	4.8
D-16	Claude Sonnet 4	×	×	×	4.0	2.8	4.2	4.0
D-30	Claude Sonnet 4	✓	✓	✓	8.3	4.6	4.2	4.8
D-1	Claude Sonnet 4	✓	✓	✓	8.0	4.7	4.2	4.9
D-2	Claude Sonnet 4	✓	✓	✓	8.6	4.2	4.5	4.9
D-34	Claude Sonnet 4	✓	✓	✓	9.0	3.6	4.1	4.9
D-6	Claude Sonnet 4	×	×	×	6.5	1.8	4.2	3.2
D-17	Claude Sonnet 4	✓	✓	×	7.9	3.8	4.5	4.8
D-26	Claude Sonnet 4	✓	✓	✓	8.5	4.2	4.2	4.8
D-10	Claude Sonnet 4	✓	✓	✓	8.0	4.4	4.2	4.6
D-28	Claude Sonnet 4	✓	✓	✓	9.3	4.5	4.0	4.4
D-32	Claude Sonnet 4	✓	✓	✓	8.1	4.7	4.2	4.6
D-9	Claude Sonnet 4	✓	✓	✓	6.3	4.6	4.5	5.0
D-5	Claude Sonnet 4	✓	✓	✓	7.6	4.3	4.5	4.9
D-8	Claude Sonnet 4	✓	✓	✓	8.0	4.3	4.2	4.1
D-18	Claude Sonnet 4	✓	✓	✓	9.4	4.7	4.3	4.8
D-36	Claude Sonnet 4	✓	✓	✓	8.0	3.4	4.2	4.9
D-15	Claude Sonnet 4	×	✓	✓	N/A	1.1	1.5	1.0
D-14	Claude Sonnet 4	×	×	×	6.0	3.2	4.2	4.8
D-13	Claude Sonnet 4	✓	✓	✓	8.8	4.4	4.2	4.8
D-23	Claude Sonnet 4	✓	✓	✓	9.5	4.6	4.2	5.0
D-33	Claude Sonnet 4	✓	✓	✓	9.7	4.5	4.2	4.9
D11	Claude Sonnet 4	×	×	×	4.0	3.5	4.2	4.8
D-29	Claude Sonnet 4	✓	✓	✓	9.2	4.5	4.2	4.9
D-24	Claude Sonnet 4	✓	✓	✓	7.8	3.9	4.2	3.7
D-25	Claude Sonnet 4	✓	✓	×	7.8	4.3	4.2	4.2
D-21	Claude Sonnet 4	✓	✓	×	8.1	4.7	4.2	4.9
D-3	Claude Sonnet 4	✓	✓	✓	8.2	4.4	4.2	5.0
D-4	Claude Sonnet 4	✓	✓	✓	7.4	3.2	4.2	4.8
D-19	Claude Sonnet 4	✓	✓	✓	8.6	3.9	4.2	4.8
D-31	Claude Sonnet 4	✓	✓	✓	9.6	4.5	4.2	4.3
D-35	Claude Sonnet 4	✓	✓	✓	9.1	4.5	4.2	4.9
D-20	Claude Sonnet 4	✓	✓	✓	N/A	1.0	1.0	1.0
D-27	Claude Sonnet 4	✓	✓	✓	8.7	4.7	4.9	4.8
D-22	Claude Sonnet 4	✓	✓	✓	7.4	3.5	4.2	4.3
D-12	Claude Sonnet 4	✓	×	×	4.0	2.1	4.0	4.1
D-7	LLaMa 3.1 - 8B	×	✓	✓	N/A	1.0	1.0	1.0
D-16	LLaMa 3.1 - 8B	✓	✓	✓	8.7	1.2	4.2	2.4
D-30	LLaMa 3.1 - 8B	✓	✓	✓	8.9	4.1	4.1	4.7
D-1	LLaMa 3.1 - 8B	✓	✓	✓	9.1	2.5	3.5	2.2
D-2	LLaMa 3.1 - 8B	✓	✓	×	8.2	2.4	4.2	3.4
D-34	LLaMa 3.1 - 8B	×	×	×	4.0	2.4	3.7	3.1
D-6	LLaMa 3.1 - 8B	×	×	×	4.0	1.2	2.1	2.9
D-17	LLaMa 3.1 - 8B	✓	✓	✓	9.1	1.6	4.2	2.5
D-26	LLaMa 3.1 - 8B	×	×	×	4.0	1.2	1.0	1.4
D-10	LLaMa 3.1 - 8B	×	×	×	4.0	1.2	3.5	2.0
D-28	LLaMa 3.1 - 8B	×	✓	✓	N/A	1.0	1.0	1.0
D-32	LLaMa 3.1 - 8B	✓	✓	✓	8.2	4.3	4.1	4.2
D-9	LLaMa 3.1 - 8B	×	×	×	4.0	1.2	2.5	1.6
D-5	LLaMa 3.1 - 8B	✓	×	×	4.0	1.8	2.4	1.4
D-8	LLaMa 3.1 - 8B	✓	✓	✓	9.7	2.1	3.1	2.0

Continued on next page

Table D.3 – *continued from previous page*

Case ID	Model	isinstance	Syntax	Runtime	Quality	GPT OSS	Llama 3.3	Qwen3
D-18	LLaMa 3.1 - 8B	✓	✓	✓	8.1	2.1	4.1	4.8
D-36	LLaMa 3.1 - 8B	×	×	×	4.0	1.9	4.2	2.0
D-15	LLaMa 3.1 - 8B	×	✓	✓	N/A	1.0	3.5	1.0
D-14	LLaMa 3.1 - 8B	×	×	×	4.0	1.3	2.5	2.2
D-13	LLaMa 3.1 - 8B	×	×	×	4.0	1.0	2.5	1.6
D-23	LLaMa 3.1 - 8B	×	✓	✓	N/A	1.0	1.0	1.0
D-33	LLaMa 3.1 - 8B	×	✓	✓	N/A	1.0	1.0	1.0
D11	LLaMa 3.1 - 8B	✓	✓	✓	8.9	2.0	4.2	3.0
D-29	LLaMa 3.1 - 8B	×	×	×	4.0	1.4	2.5	1.6
D-24	LLaMa 3.1 - 8B	✓	✓	✓	N/A	1.4	3.8	2.1
D-25	LLaMa 3.1 - 8B	×	×	×	4.0	1.2	1.4	1.4
D-21	LLaMa 3.1 - 8B	×	×	×	4.0	1.0	4.2	1.4
D-3	LLaMa 3.1 - 8B	×	×	×	4.0	1.0	2.2	2.2
D-4	LLaMa 3.1 - 8B	✓	✓	✓	9.2	1.2	4.2	2.2
D-19	LLaMa 3.1 - 8B	✓	✓	✓	N/A	2.1	4.2	2.9
D-31	LLaMa 3.1 - 8B	×	✓	✓	N/A	1.0	1.7	1.0
D-35	LLaMa 3.1 - 8B	×	×	×	4.0	1.3	3.9	3.1
D-20	LLaMa 3.1 - 8B	×	×	×	4.0	1.0	2.1	1.6
D-27	LLaMa 3.1 - 8B	✓	✓	✓	8.3	4.2	4.2	4.5
D-22	LLaMa 3.1 - 8B	✓	✓	×	9.7	1.3	4.2	1.9
D-12	LLaMa 3.1 - 8B	×	×	×	4.0	1.2	1.8	1.4

Educational Agent

Table D.4: Evaluation results for Educational Agent.

Case ID	Model	isinstance	Quality	GPT OSS	Llama 3.3	Qwen3
E-7	Claude Sonnet 4	✓	6.8	4.3	4.1	4.8
E-3	Claude Sonnet 4	✓	6.5	4.1	4.4	4.1
E-5	Claude Sonnet 4	✓	7.4	4.1	4.1	4.8
E-12	Claude Sonnet 4	✓	7.3	4.1	4.5	4.7
E-1	Claude Sonnet 4	✓	8.7	3.9	4.3	4.6
E-2	Claude Sonnet 4	✓	6.3	4.2	4.2	4.5
E-10	Claude Sonnet 4	✓	8.8	3.9	4.2	5.0
E-13	Claude Sonnet 4	✓	8.5	4.2	4.3	4.9
E-14	Claude Sonnet 4	✓	7.2	4.2	4.4	4.9
E-18	Claude Sonnet 4	✓	N/A	4.1	4.1	N/A
E-8	Claude Sonnet 4	✓	6.1	4.1	4.3	4.9
E-9	Claude Sonnet 4	✓	7.5	4.1	4.3	4.3
E-22	Claude Sonnet 4	✓	8.0	4.1	4.4	4.5
E-21	Claude Sonnet 4	✓	6.7	3.9	4.3	4.9
E-11	Claude Sonnet 4	✓	6.7	3.9	4.1	4.5
E-15	Claude Sonnet 4	✓	8.3	4.2	4.2	4.9
E-16	Claude Sonnet 4	✓	7.3	4.1	4.1	4.2
E-6	Claude Sonnet 4	✓	6.8	4.5	4.1	4.3
E-20	Claude Sonnet 4	✓	6.8	4.3	4.1	4.7
E-23	Claude Sonnet 4	✓	6.5	4.3	4.3	4.1
E-17	Claude Sonnet 4	✓	6.7	4.2	4.1	4.2
E-4	Claude Sonnet 4	✓	7.3	4.2	4.4	4.5
E-19	Claude Sonnet 4	✓	7.7	4.1	4.1	4.8

Continued on next page

Table D.4 – *continued from previous page*

Case ID	Model	isinstance	Quality	GPT OSS	Llama 3.3	Qwen3
E-7	LLaMa 3.1 - 8B	✓	N/A	3.7	4.0	4.3
E-3	LLaMa 3.1 - 8B	✓	N/A	2.8	3.8	3.9
E-5	LLaMa 3.1 - 8B	✓	N/A	1.6	2.6	2.1
E-12	LLaMa 3.1 - 8B	×	N/A	1.0	1.0	1.0
E-1	LLaMa 3.1 - 8B	×	N/A	1.0	1.0	1.0
E-2	LLaMa 3.1 - 8B	✓	N/A	2.8	3.3	4.0
E-10	LLaMa 3.1 - 8B	×	N/A	1.0	1.0	1.0
E-13	LLaMa 3.1 - 8B	×	N/A	1.0	1.0	1.0
E-14	LLaMa 3.1 - 8B	✓	N/A	1.5	2.7	2.1
E-18	LLaMa 3.1 - 8B	×	N/A	1.0	1.0	1.0
E-8	LLaMa 3.1 - 8B	×	N/A	1.0	1.0	1.0
E-9	LLaMa 3.1 - 8B	×	N/A	1.0	1.0	1.0
E-22	LLaMa 3.1 - 8B	✓	N/A	1.0	1.0	1.0
E-21	LLaMa 3.1 - 8B	✓	7.3	3.9	4.1	4.3
E-11	LLaMa 3.1 - 8B	×	N/A	1.0	1.0	1.0
E-15	LLaMa 3.1 - 8B	✓	N/A	2.6	2.7	3.1
E-16	LLaMa 3.1 - 8B	✓	N/A	3.1	3.8	3.3
E-6	LLaMa 3.1 - 8B	✓	N/A	3.2	3.5	3.8
E-20	LLaMa 3.1 - 8B	✓	N/A	2.8	3.8	2.6
E-23	LLaMa 3.1 - 8B	✓	N/A	2.4	4.0	3.6
E-17	LLaMa 3.1 - 8B	×	N/A	1.0	1.0	1.4
E-4	LLaMa 3.1 - 8B	×	N/A	1.0	1.0	1.0
E-19	LLaMa 3.1 - 8B	×	N/A	1.0	1.0	1.0

D.3. Generated Geometries

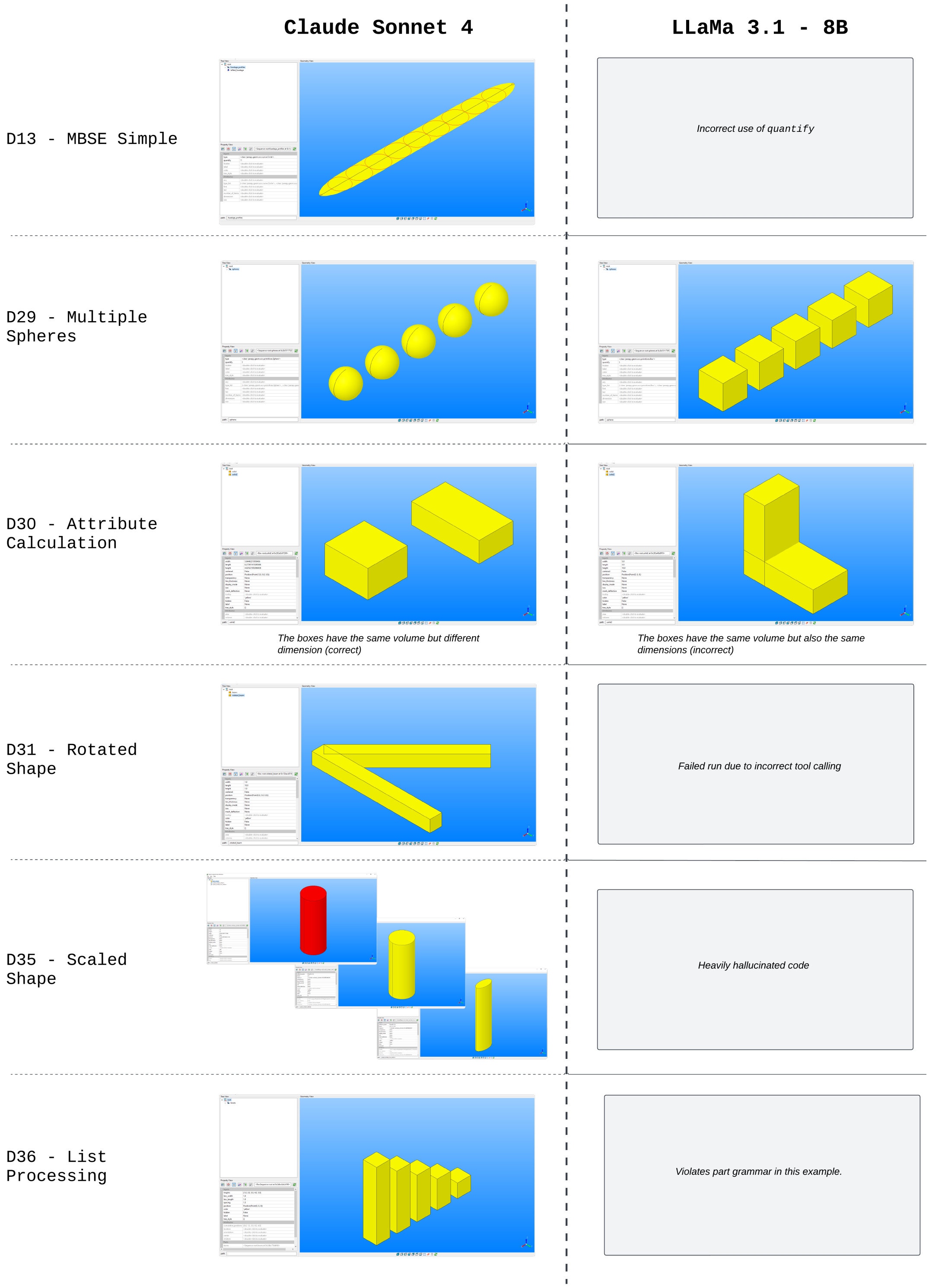


Figure D.1: Comparison of Developer Agent performance using Claude Sonnet 4 and LLaMa 3.1 – 8B. The results correspond to the skeleton code completion evaluation cases and the associated geometries generated via the ParaPy SDK.