Retrieval First: LLM-Assisted Type Inference for Automatic Test Case Generation in JavaScript

Lucian Negru[1]

Supervisor(s): Annibale Panichella[1]**, Mitchell Olsthoorn**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A thesis submitted to EEMCS Faculty Delft University of Technology,
In partial fulfilment of the requirements
for the degree of Master of Science in Computer Science
April 15, 2026

Name of the student: Lucian Negru
Final project course: CS5000 Master Thesis
Thesis committee: Annibale Panichella, Mitchell Olsthoorn, Christoph Lofi

Retrieval First: LLM-Assisted Type Inference for Automatic Test Case Generation in JavaScript

Lucian Negru

Delft University of Technology
Delft, The Netherlands

Abstract

Automatic test case generation for dynamically typed languages such as JavaScript is significantly hindered by the absence of explicit type information, which expands the search space for search-based testing and reduces its effectiveness. While prior probabilistic and neural type inference methods address this, they struggle with complex user-defined types, higher-order functions, and external package dependencies. This paper presents and evaluates three LLM-based approaches for type inference in JavaScript. The primary contribution is a Retrieval-Augmented Generation (RAG) approach that constructs a vector database of semantically rich code embeddings. These embeddings include ASTs, program slices, and code annotations. This enables efficient, project-wide context retrieval paired with Chain-of-Thought prompting. In a large-scale empirical evaluation against the SynTest framework, the RAG approach achieves a 29% average accuracy improvement over non-RAG LLM approaches, an 85% reduction in computation time, and a 63% accuracy improvement over probabilistic inference for deep, user-defined types. For primitive types, probabilistic methods remain competitive. These findings motivate future hybrid strategies combining probabilistic and LLM-based inference.

Keywords

Type inference, JavaScript, Large language model, Retrieval-augmented generation, Automatic test case generation, Empirical software engineering

1 Introduction

Software testing plays a critical role in ensuring the correctness, reliability, and maintainability of software systems. Well-tested code reduces the likelihood of defects, informs refactoring, and increases confidence in software [38]. However, writing and maintaining test suites is a time-consuming and error-prone process, and in practice, many software projects suffer from insufficient test coverage [40]. This issue is particularly prevalent in ecosystems that prioritise rapid development and flexibility.

Automatic Test Case Generation has been proposed as a means to mitigate these challenges. By automatically generating test cases, it increases coverage while reducing manual effort. From the automated approaches, search-based approaches stand out for their ability to optimize for this coverage criteria without requiring formal program specifications [15]. Search-based approaches have demonstrated strong results in statically typed languages, particularly with respect to code coverage [19] and bug detection [2]. Tools such as *EvoSuite* [11] and *Testful* [4] exploit static type information to constrain the search space of possible inputs, making test generation efficient and effective.

Dynamically typed languages have gained significant traction over the past decade due to their lack of coding constraints and reduced boilerplate, enabling faster prototyping and iteration. A 2025 Stack Overflow survey[1] reports that JavaScript and Python are the most popular programming languages, with JavaScript maintaining the leading position since 2014. Despite this widespread adoption, research by Fard and Mesbah [10] shows that in 2017, 22% of public JavaScript code lacked tests, raising concerns about code quality, stability, and maintainability.

In dynamically typed languages, the absence of explicit type information significantly complicates search-based test generation. Search algorithms must explore a substantially larger input space, as they cannot rely on type constraints to guide test setup and input construction. As a result, the effectiveness of automatic test case generation drops considerably when applied to languages such as JavaScript.

Prior work has explored enabling automatic test case generation for dynamically typed languages through type inference. Early approaches, such as Pynguin for Python [25], rely on randomly guessing variable types, which is effective for primitives but inadequate for user-defined objects. More advanced techniques, including JSNice [39] and SynTest for JavaScript [41], apply probabilistic inference based on observed usage. While these methods improve primitive type prediction, support for complex user-defined objects remains limited and often depends on heuristics, predefined declarations, or iterative refinement during test execution, as also explored in Type Tracing [21]. Neural-network-based approaches such as DeepTyper [16] and NL2Type [26] achieve high accuracy but require domain-specific training data, for example, JSDoc comments, which are rarely available in practice (0.1% of JavaScript projects reported on Stackshare[2] use JSDoc). Large language models (LLM) represent a promising direction for type inference due to their ability to analyse various forms of context-rich information—such as Abstract Syntax Trees (AST), flow-graphs, variable names, and code annotations (e.g. [16, 26, 30, 39]). Unlike neural networks, which require domain-specific pre-training, LLMs can leverage diverse, context-rich information, enabling them to reason about both structural and linguistic cues that probabilistic methods often overlook. This flexibility makes LLMs particularly suitable for exploring context-aware inference. Nevertheless, current approaches to LLM-based type inference often process entire ASTs [49] or source files, which increases computational cost and inference time due to token usage and risks information dilution or hallucination (e.g. [37, 44, 47]).

Conference'17, Washington, DC, USA
2026. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1] https://survey.stackoverflow.co/2025/technology#most-popular-technologies-language
[2] https://stackshare.io

To address this gap, we can investigate techniques such as code embeddings, which allow for more effective code search, enabling semantic retrieval, cross-file and cross-project inference, and improved in-context learning [24, 32]. Additionally, they offer a highly scalable approach to information retrieval suitable for large code bases due to their efficient token usage.

This empirical study investigates techniques for extracting context-rich information from source code to improve LLM-based type inference in JavaScript, particularly for complex user-defined types. The approach applies Retrieval-Augmented Generation (RAG) techniques by generating code embeddings from a range of information-rich snippets, storing them in a vector database. These include abstract syntax trees (ASTs), program slices, heuristic hints, and code annotations. The embeddings are then retrieved and used alongside Chain-of-Thought (CoT) prompting to the LLM. The study also assesses local models for possible local applicability. The performance of the approach is evaluated with both local and cloud models against the probabilistic inference of the SynTest framework.

The results demonstrate that the RAG-based approach significantly outperforms the other LLM approaches, achieving an average accuracy improvement of 29% and an average reduction in computation time of 85%. They also show a substantial accuracy improvement of 63% over the probabilistic technique of SynTest [41]—specifically for deep, user-created types and package-reliant functions.

The paper presents insights into the use of LLM-based approaches for type inference for JavaScript and argues that they offer an applicable benefit in real-world cases involving complex user-defined types, higher-order functions, and a reliance on external packages. We found that for use-cases involving primitive types, probabilistic approaches fare better as their speed allows them to iterate multiple times before any model can respond. This insight reveals potential future work in hybrid approaches relying on probabilistic methods for primitive inference and LLM-based methods for more complex types.

The main contributions of this research are:

(1) A RAG-based LLM type inference approach for JavaScript.
(2) A large-scale evaluation of multiple models on type inference.
(3) A replication package [33].

The remainder of this thesis is structured as follows: Section 2 details the background information and the related work, Section 3 introduces the proposed approaches, Section 4 describes the study design, Section 5 presents and analyses the experimental results, Section 6 tackles the threats to validity, and the final section discusses the findings and outlines directions for future work.

2 Background and Related Work

This section introduces the key concepts required to understand this study and discusses the related work.

2.1 Automatic Test Case Generation

Thorough software testing ensures functionality and maintainability, but it is tedious and time-consuming work [9]. Therefore, researchers have developed research methods of automating the generation of tests while maintaining high coverage.

Early on, these included random testing [14] and partition testing [35], which employed black-box techniques for identifying bugs and reducing the number of test cases for sufficient coverage. However, these lead to difficulties in reproducing bugs and covering corner cases. Later, symbolic [20] and constraint-based [8] methods were used to automatically catch edge cases. Search-Based Software Testing (SBST), by means of genetic algorithms for example, generate test cases that optimise a fitness function—often branch coverage [11]. Of these SBST techniques, the DynaMOSA [36] algorithm has been shown to be the most effective in automatic test case generation.

2.2 Type Inference

The lack of a static type system in languages like JavaScript can lead to late error-detection, unexpected type conversions, and risky refactors [13]. But in the context of automatic test case generation, it can reduce the effectiveness of the search-based approaches by increasing their search space. To address this, type inference was introduced.

Early approaches (e.g. [1], [46]) to type inference in JavaScript and similar dynamically typed languages were inspired by soft typing, where static analysis techniques attempt to infer types without enforcing them, allowing programs to remain flexible while still detecting potential type inconsistencies [18]. However, they lack strict guarantees and are imprecise, often resolving to general types (e.g. `any`-like abstractions), making them poorly suited for tooling.

Building on this idea, TypeScript[3] introduced an optional static type system for JavaScript, combining explicit annotations with type inference to improve developer tooling and error detection while maintaining compatibility with existing code. The drawback is that the project, and its dependencies, all require this type safety for the static inference to work accurately. Patterns involving `any`, dynamic property access, or runtime mutation can reduce precision.

Subsequent research shifted toward probabilistic approaches, which model likely types based on observed usage patterns. For example, JSNice [39] applies graphical models to infer types and variable names, while approaches such as Type Tracing [21] and SynTest [41] incorporate dynamic observations and probabilistic reasoning to refine type predictions during execution or test generation. These probabilistic approaches perform well on primitive types and common patterns but struggle with complex user-defined objects, nested structures, and higher-order functions.

More recently, neural network-based approaches have been proposed to capture deeper semantic patterns in code. Models such as DeepTyper [16], NL2Type [26], and Type4Py [30] leverage machine learning to predict types based on large code corpora, often incorporating contextual information such as naming conventions or natural language comments, and demonstrating improved performance over purely probabilistic methods, especially for complex user-defined types. However, they are highly dependent on training data and, as such, applicable to the same context, making them less generalisable as a whole.

Lastly, LLM-based approaches (e.g. [37, 44, 47]) have been developed to take advantage of the deep semantic patterns without the drawback of strong dependency to domain-specific training data. Current approaches leverage the AST [49] or the entire source files to draw context; however, this can bloat the context and miss out on relevant information elsewhere in the source code. To address

[3] https://www.typescriptlang.org/

this, we investigate a RAG-based approach for efficient and relevant context-retrieval.

2.3 Prompting

Prompt design plays a crucial role in the effectiveness of large language models, as it determines how tasks are interpreted and how context is utilised. Early approaches such as zero-shot and few-shot prompting rely on instructions and examples, but are often sensitive to phrasing and less effective for complex tasks [43, 51]. More advanced techniques, such as Chain-of-Thought (CoT) prompting, improve reasoning by encouraging intermediate steps, though at the cost of increased token usage [45].

To improve robustness and consistency, structured prompting defines explicit input and output formats, often using schema-like representations (e.g., JSON and TOON [28]). This is particularly useful for program analysis tasks, where outputs must be machine-readable and consistent. At the same time, efficient context usage is essential when working with large codebases. Providing full source files can introduce noise and exceed token limits, motivating approaches such as context pruning [17].

Finally, robustness can be further improved through techniques such as enforcing output constraints and using retry mechanisms to handle uncertain predictions. These strategies produce accurate and efficient responses in large-scale code settings.

2.4 Retrieval-Augmented Generation

Retrieval-Augmented Generation is a technique in which LLM querying is combined with a **retrieval system** so the model can look up relevant information before generating an answer. This contrasts with how LLMs normally generate responses based on the parameters that were formed during training. The approach was initially developed for NLP tasks in specific domains, where the retrieval system would query a domain-specific index of information to outperform parametric-only (non-RAG) models in generation correctness and diversity [23]. It works by indexing a domain-specific knowledge base into a database, which can then be queried for relevance to the user's prompt.

In the context of software development, code snippets can be converted into embeddings,a numeric representation of information,and stored in a vector database. Alon et al. [3] present suitable code snippet extraction techniques in Java, for the purpose of predicting method names. Their technique analyses a Java program's AST, extracts information relevant to method definition, and embeds it in a vector database used by an RAG model. This technique has also been widely adopted by code editors and LLM extensions such as VSCode[4] and Copilot[5]. Copilot in VSCode builds an index of the workspace (indexable files, folder structure and symbols/definitions) to draw context. When the client sends a query, Copilot uses a mix of strategies (remote code search via GitHub, local semantic search via embeddings, text search, symbol metadata, file-name heuristics) to find relevant snippets. These snippets are then included in the context of the initial prompt.

The benefits of this approach are that the model can 'understand' larger code bases by performing lookups, it reduces hallucination by providing more factual responses, and it is not limited to possibly outdated or incorrect knowledge from training [50].

[4]https://github.com/microsoft/vscode
[5]https://code.visualstudio.com/docs/copilot/reference/workspace-context

3 Approach

This section presents the approaches investigated in this thesis for applying large LLMs to type inference in JavaScript. By inferring types for functions and variables before test generation, it becomes possible to constrain the input domain and guide the search process more effectively.

Recent advances in LLMs have demonstrated strong capabilities in code understanding and reasoning, making them a promising candidate for performing type inference in dynamically typed languages [48]. However, the accuracy of the inferred types depends heavily on how contextual information is provided to the model. In particular, JavaScript programs often rely on implicit contracts, cross-file interactions, and project-wide conventions, which may not be captured when analysing a function or a single file in isolation.

To address this challenge, we explore multiple approaches that differ in how program context is extracted, represented, and supplied to the LLM. These approaches range from providing the complete source file as input to structured representations of the program via abstract syntax trees, and finally to a RAG strategy that leverages project-wide information through the retrieval of embeddings. The flow of each approach can be seen in Figure 1. Each approach reflects a different trade-off between contextual completeness, token use, and computational overhead. Additionally, the prompts used for each approach can be viewed in Appendix A.

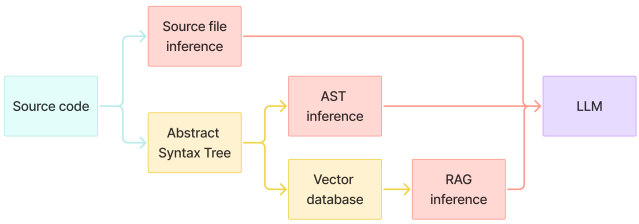

Figure 1: Overview of the three proposed approaches.

3.1 Source File Inference

The first approach investigated in this thesis relies on providing the large language model with source-level context derived directly from the JavaScript project. Unlike other approaches explored later, this method does not perform semantic preprocessing. Instead, it focuses on selectively expanding the prompt context based on import dependencies, while preserving the original source code representation. The core idea of this approach is to provide the LLM with sufficient local and transitive context to infer types accurately, without overwhelming it with the entire project.

Given a source file containing the function under test, the file itself is always included in the prompt. In addition, the approach analyses the list of imports declared in the file and selectively incorporates the corresponding declarations into the context. For each imported entity, the source declaration is added to the prompt. In the case of classes, this includes the class definition and its constructor. If the class constructor or imported function in turn depends on other imported

entities, the process is applied recursively. This recursive expansion allows the prompt to capture transitive dependencies that may be necessary to understand object structure, parameter expectations, or return values. The maximum recursion depth is configurable to control prompt size and computational cost.

Once the contextual information has been assembled, the LLM is prompted to perform type inference for the functions and variables of interest. The model is instructed to return its predictions using a predefined, structured output format. This structure is validated after generation to ensure syntactic correctness and compatibility with the evaluation pipeline.

To increase the usefulness of the inferred types for search-based test generation, the LLM is further instructed to decompose complex types into their primitive components. Rather than returning high-level or nominal types (e.g., a class name), the model must express such types as object structures that explicitly list their attributes and the inferred types of those attributes. This representation enables a more fine-grained restriction of the search space. All instructions are then parsed into Token-Oriented Object Notation[6] (TOON) to improve the efficiency of the request, calculated as $\textit{Efficiency} = \textit{Accuracy} \div \textit{Tokens}$.

Overall, this source file–based approach represents a baseline strategy that prioritises simplicity to the original code. It serves as a reference point for evaluating the impact of more structured and retrieval-based context construction techniques introduced later in this section.

3.2 AST Inference

The second approach builds upon the source file–based strategy described previously, but introduces an intermediate abstraction step by parsing the aggregated source code into an AST. The motivation for this approach is to provide the large language model with a more structured and distilled representation of the program, while retaining the same contextual scope defined by the Source File Inference approach.

As in the source file approach, the context construction begins by aggregating the source file containing the function under test together with selected imported declarations and their transitive dependencies, up to a configurable depth. Instead of forwarding the raw source code directly to the LLM, the combined code is first parsed into an AST using the Babel parser[7]. This AST serves as the basis for extracting semantically relevant snippets that are expected to be informative for type inference.

The traversal of the AST focuses on a subset of syntactic constructs that Yang et al. [49] identified as important for understanding data flow and type usage in JavaScript programs. These include function declarations, arrow functions assigned to variables, variable declarations, class declarations, and class methods. For each of these constructs, a summarised representation is derived and later included in the prompt.

3.2.1 Functions. For function declarations and class methods, the analysis focuses on two aspects: parameter usage and function body behaviour. Parameter usage is examined to identify how parameters are referenced within the function body, such as whether they are accessed as objects, invoked as functions, or used through property or method access. This information provides indirect evidence about the expected structure and capabilities of the parameter types. In parallel, the function body is summarised by analysing statements related to return behaviour, variable usage, function and method calls, control flow constructs, and lightweight type hints derived from expressions. Together, these summaries aim to capture how values are produced and consumed without exposing the full syntactic complexity of the original code.

3.2.2 Arrow Function Variables. Arrow functions assigned to variables are treated similarly. When these functions contain block bodies, the same body analysis is applied as for regular function declarations. When the body consists of a single expression, the expression is analysed directly to infer high-level characteristics, such as whether it represents a literal value, a binary operation, a function call, or a composite data structure. From this, type hints can be derived.

3.2.3 Variable Declarations. Variable declarations are analysed primarily through their initialisers. Literal initialisers provide direct evidence of primitive types, while array and object initialisers allow limited inference of element and property types using heuristics. Constructor calls are interpreted as indications of nominal types. When no reliable inference can be made, the variable is conservatively assigned a generic type. This initialiser-focused analysis provides early type constraints that can be propagated downwards.

3.2.4 Class Declarations. Class declarations are summarised at two levels. At the class level, information about available methods and properties is extracted to convey the overall shape of the object. At the method level, individual class methods are treated similarly to standalone functions, with parameter usage and body summaries generated separately. This separation is designed to allow the LLM to reason both about the structure of class instances and the behaviour of their methods.

3.2.5 Design Rationale. The AST-based approach is designed to reduce noise in the information provided to the LLM by filtering out syntactic details that are unlikely to contribute directly to type inference, while explicitly highlighting semantically relevant usage patterns. By emphasising how values are accessed, combined, and passed through the program rather than presenting raw source code, this abstraction is intended to make implicit type information more accessible to the model. Additionally, the structured nature of the extracted AST nodes is assumed to enable more consistent prompt formatting, which may lead to more stable and interpretable model outputs. Given that the AST structure produced from the analysis is verbose and may lead to large token usage, TOON for the AST was considered as a means of improving the efficiency of the approach by up to 22% [29]; however, the heavily-nested structure of the AST led to diminishing returns in comparison to *JSON-compact* notation.

At the same time, this approach may introduce certain limitations. The reliance on heuristics for interpreting binary operations, array elements, and object properties may result in incomplete or overly coarse type hints. Furthermore, the abstraction process omits some contextual information that could potentially aid type inference. The additional preprocessing step may also increase the inference computation and token usage substantially.

[6] https://github.com/toon-format/toon?tab=readme-ov-file
[7] https://babeljs.io/docs/babel-parser

3.3 RAG Inference

The third approach explored in this thesis, illustrated in figure 2, adopts a RAG strategy to support type inference in JavaScript. The primary goal of this approach is to provide the large language model with access to relevant, project-wide information while avoiding excessive prompt sizes that would result from supplying entire source files or dependency graphs. By allowing the model to retrieve targeted contextual evidence on demand, this approach is intended to promote correctness and reduce the likelihood of hallucinations.

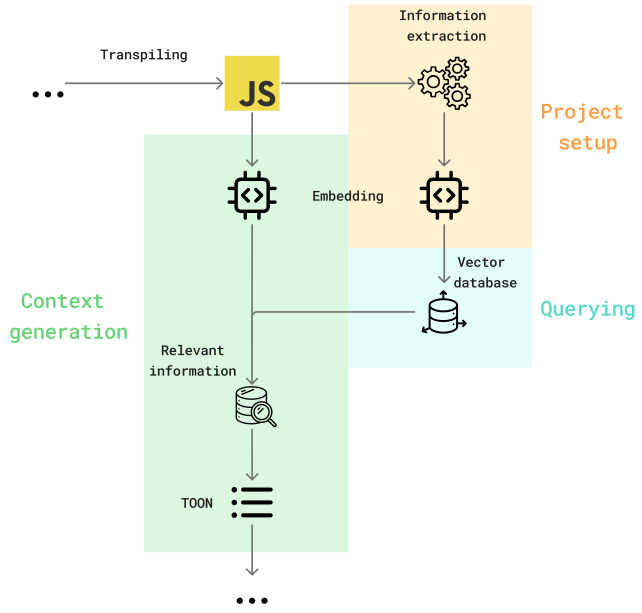

Figure 2: RAG-based type inference pipeline.

In contrast to the previous approaches, which explicitly construct a fixed prompt context, the RAG-based method is designed to enable scalable reasoning over large codebases. It aims to approximate a form of "project awareness" by allowing the model to search through the entire codebase for semantically related information without directly embedding all of it into the prompt. This makes the approach particularly suitable for local and resource-constrained scenarios.

The approach consists of two main stages: (1) information extraction and storage, and (2) retrieval and querying at inference time.

3.3.1 Information Extraction and Storage. In the extraction stage, the complete project directory is traversed, and all relevant source files are parsed into abstract syntax trees. From these ASTs, a set of code snippets is extracted into information categories. Alon et al. [3] propose that the categories defined for code snippets in *code2vec* may serve as a suitable basis for predicting method names. Each extracted unit is enriched with metadata such as its source location, a brief semantic description, and the corresponding raw source code.

The extracted information includes categories that are expected to provide useful information for type inference:

- **Function definitions and return expressions**, which describe parameter structure, control flow scope, and produced values.
- **Call sites**, which provide insight into how functions are invoked, the shapes of arguments passed, and the contexts in which return values are used.
- **Object literal shapes**, which expose structural type information common in JavaScript's object-centric programming style.
- **Variable declarations and assignments**, which offer strong initial type signals and information about mutability.
- **Class declarations, methods, and inheritance relationships**, which convey type structure and usage patterns.
- **Imports and exports**, which reveal module boundaries and inter-file dependencies relevant to type propagation.
- **JSDoc comments**, which may contain explicit developer-provided type annotations and contracts.
- **Literal values**, which serve as the most direct evidence for primitive types.
- **Control-flow hints**, such as `typeof` checks and switch statements, which often refine types within specific branches.

The reason we extract this diverse set of snippets is to capture type-relevant evidence from multiple perspectives: declaration-based, usage-based, structural, and control-flow–based. While any single category may provide incomplete information, their combination is expected to give the LLM a richer basis for inference similar to that of *code2vec* [3].

Once extracted, all snippets are embedded and stored in a vector database. This allows similarity-based retrieval across the entire project, enabling later queries to surface information that is semantically related even if it is not syntactically or lexically adjacent.

3.3.2 Retrieval and Querying. At inference time, only the function under test is provided as the query input to the retrieval system. This query is embedded and used to retrieve the **top-N** most relevant snippets from the Qdrant[8] vector database. If one of the retrieved snippets is the exact function-under-test, it gets omitted from the set. These retrieved snippets are intended to represent the most informative project-level context with respect to the code under test.

The retrieved snippets are then supplied to the LLM alongside the function under test, without including the full source file or its import dependencies. This design choice is intended to minimise prompt size while maximising informational relevance. The LLM is prompted for type inference using the same structured output constraints as in the previous approaches.

3.3.3 Design Rationale. Vector-based retrieval enables semantic similarity search, which is well-suited for code, where relevant context may not share exact identifiers or syntactic structure [42]. Additionally, vector databases are optimised for efficient nearest-neighbour queries, making them practical for repeated inference requests [42]. The use of a locally deployed database instance aligns with the goal of local applicability.

More broadly, the RAG approach is based on the hypothesis that type-relevant information is often distributed across a codebase rather than localised within a single file. By decoupling context selection from prompt construction and delegating it to a retrieval

[8] https://qdrant.tech/

mechanism, the approach is expected to scale better to larger projects than the source file–based or AST-based strategies.

However, this approach is also assumed to introduce new challenges. The quality of retrieved context depends on the effectiveness of the embeddings and the suitability of the extracted snippet representations. Irrelevant or weakly related snippets may dilute the prompt, while missing critical context could still lead to incomplete type inference. Furthermore, the approach introduces additional system complexity through the need for preprocessing, storage, and retrieval infrastructure.

4 Study Design

This section describes the design of the empirical study conducted to evaluate the approaches proposed in this thesis. The goal of the study is to systematically assess the effectiveness of LLM–based type inference techniques for JavaScript. In addition to this goal, the local applicability of the techniques is prioritised.

For this assessment, we present the following research questions:

- **RQ1 — Type Inference Effectiveness.**
  How does the accuracy of the proposed LLM-based type inference approaches vary when using different models?
- **RQ2 — Token and Time Consumption.**
  What are the costs associated with the proposed approaches, measured in token usage and inference time?
- **RQ3 — Comparison with Probabilistic Technique.**
  How does the accuracy of LLM-based type inference approaches compare to SynTest's probabilistic inference?
- **RQ4 — Retrieved Context in RAG.**
  What types of information are most frequently retrieved for the RAG approach?

To ensure that the evaluation is rigorous, reproducible, and interpretable, this section details the benchmarks, experimental configurations and parameter settings, experimental protocol and hardware setup, and the statistical analysis methods used to assess the significance and magnitude of observed differences.

4.1 Benchmark

The benchmark is constructed based on widely used TypeScript projects. TypeScript projects are selected instead of native JavaScript projects because they provide explicit type annotations that can be treated as ground truth. This allows the study to compare inferred types against known reference types while still evaluating the approaches in a JavaScript setting by transpiling the code to JavaScript. This is explored further in section 4.3

The benchmark consists of three projects: TypeScript[9], VSCode[10], and Vue[11]. Together, these projects cover a range of programming styles and abstraction levels commonly encountered in real-world development.

The benchmark focuses on 50 functions that are suitable for automatic test generation. Files primarily intended for distribution, such as pre-built artefacts, as well as index and aggregation files, are excluded from selection to avoid non-executable or structurally trivial code. the files were selected randomly based on these criteria. These functions amass 84 `string` types, 185 `number` types, 43 `boolean` types, 122 `undefined` types, 35 `null` types, and 204 custom types.

[9] https://github.com/microsoft/TypeScript
[10] https://github.com/microsoft/vscode
[11] https://github.com/vuejs/vue

4.2 Configurations and Parameters

To address the research questions, the experimental study evaluates multiple configurations for the large language model used for type inference, the deployment setting (cloud-based versus local), and the embedding model employed by the RAG approach. The configurations can be seen in Table 1.

Table 1: The model configurations of the study alongside their respective parameter counts. Closed-source cloud models don't publicise their parameter count.

Category	Model	Parameter Count
Local	Qwen 3[12]	30B
	Qwen 3 Coder[13]	30B
	Qwen 2.5 Coder[14]	32B
	Deepseek-R1[15]	32B
	StarCoder 2[16]	15B
Cloud	GPT-3.5-turbo	-
	GPT-4	-
	GPT-5 mini	-
Embedding (local)	lamma-embed-nemotron[17]	8B
	Qwen3-Embedding[18]	4B
	bert-base-uncased[19]	110M

[12] https://ollama.com/library/qwen3
[13] https://ollama.com/library/qwen3-coder:30b
[14] https://ollama.com/library/qwen2.5-coder:32b
[15] https://ollama.com/library/deepseek-r1:32b
[16] https://ollama.com/library/starcoder2
[17] https://huggingface.co/nvidia/llama-embed-nemotron-8b
[18] https://huggingface.co/Qwen/Qwen3-Embedding-4B
[19] https://huggingface.co/Xenova/bert-base-uncased

4.2.1 Large Language Models for Type Inference. The study considers both cloud-hosted and locally deployed large language models in order to evaluate differences in type inference performance, token and time consumption, and practical applicability.

For cloud-based inference, three models provided by OpenAI are evaluated. These models were selected to represent a range of capabilities, speed, and operational cost within a single provider ecosystem. Additionally, it allows for a controlled comparison between models that differ primarily in scale and reasoning capacity while sharing a common API and deployment environment.

For local inference, the study evaluates several open-weight models that can be deployed on local hardware. These models were chosen based on prior work and publicly reported performance on code-related tasks, as discussed by Coignion et al. [6] in a comprehensive study of LLM performance on Leetcode. The inclusion of both general-purpose and code-specialised models allows the study to examine how structured data, such as ASTs, or non-structured data, such as documentation, affect the performance of the inference across each model type.

4.2.2 Embedding Models for Retrieval-Augmented Generation. For configurations that employ the RAG approach, the choice of embedding model constitutes an additional experimental variable. The embedding models were selected from the Hugging Face Massive Text Embedding Benchmark leaderboard[12], to cover a range from lightweight embeddings to larger, code- and language-focused models. This allows the study to investigate how embedding quality and model scale influence the types of retrieved information queried from the database and, indirectly, type inference accuracy.

4.2.3 Parameter Settings. All experiments are conducted using a fixed set of prompting parameters to ensure comparability across configurations. A temperature value of 0.1 is used for both cloud-based and locally deployed models. This value is chosen instead of the default temperature of 1.0 to allow limited flexibility in model responses and to discourage responses with types `any` or `unknown`.

For RAG-based retrieval, all embedding models are used with their respective default configurations. In the RAG setup, extracted code snippets are stored in the vector database with a dimensionality configured to match the output dimensionality of the selected embedding model. Similarity between vectors is computed using **cosine** distance, which is commonly used for embedding-based semantic retrieval as it ignores the magnitude of the vectors and focuses on their direction. A recent study by Levy [22] argues that this makes it optimal for asserting similarity in text and code embeddings.

4.2.4 Rationale. The experimental configurations are designed to support a comprehensive evaluation of the proposed approaches. In particular, these configurations enable comparisons between cloud-based and local inference (RQ1, RQ2), assessments against prior test generation techniques (RQ3), and analyses of the role of retrieved context and embedding quality in the RAG approach (RQ4).

4.3 Experimental Protocol

We have designed an evaluation pipeline for extracting ground truth of the types and comparing against the inferred responses, seen in figure 3. The pipeline extracts the types from the TypeScript functions using the language's standard type-assertion methods. These are stored for evaluation, and the functions are converted to plain JavaScript using the `transpile` method. The JavaScript functions are passed to the approaches, and their responses are compared against the stored ground truth.

To answer **RQ1**, we run the evaluation pipeline on 50 functions from the 3 projects stated in the Benchmark 4.1. For each of the 3 approaches and 8 model configurations, 5 local and 3 in the cloud, we perform 10 runs; this produces 12,000 total type-inferred functions. All runs will capture token usage and inference time to help answer **RQ2**. To answer **RQ3**, we compare SynTest's *proportional sampling* approach to the best performing approach from the previous experiment using the same evaluation.

RQ4 requires that we investigate the categories of units queried from the database for the RAG approach. For this, we query the same 50 functions for the 10 most relevant units and analyse which categories identified in subsection 3.3.1 they fall under. As mentioned previously, we omit retrievals which fully match the function-under-test. These 50 functions are run 10 times for each of the 3 configurations for embedding models, producing 1,500 total runs.

[12]https://huggingface.co/spaces/mteb/leaderboard

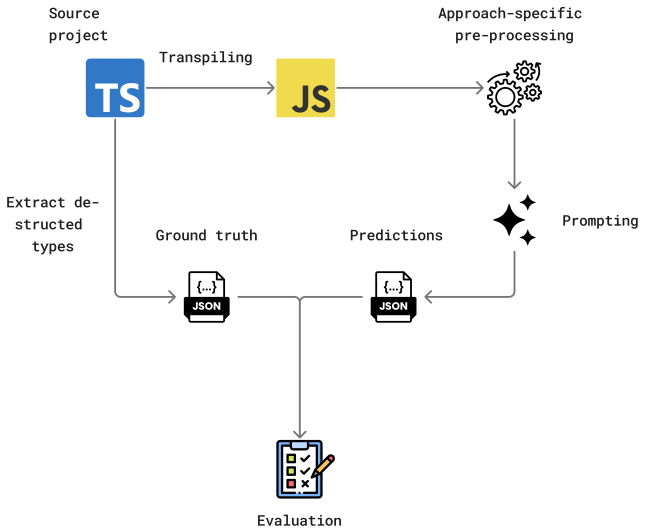

Figure 3: Evaluation pipeline used for evaluating LLM-based type inference approaches in JavaScript.

The experiments will evaluate multiple metrics for answering the research questions. *Accuracy*, or *top-1 accuracy*, describes the correctness of an inferred type as the model's first guess. Likewise, we note the accuracy until *top-5*, correctness in the first 5 guesses, to construct a Mean Reciprocal Rank (MRR) metric [7, 31]. This metric conveys how many attempts are needed to reach a certain level of correctness. Lastly, we measure resource consumption in the form of *token usage* and *inference time*, the latter of which can be used in conjunction with the accuracy to determine *efficiency*.

To ensure statistical significance, we use statistical analyses on the results for RQ1 and RQ3. We employ the Friedman test [12], a non-parametric statistical ranking test used to detect differences in treatments across multiple related groups—in our case, being the approaches—with a significance level of $\alpha = 0.05$. This gives us 144 blocks for the results of RQ1 and 18 blocks for the results of RQ3. We use Kendall's concordance coefficient W [5] to assess the agreement among the files in the benchmark against the results of the approaches. We also use the Nemenyi post hoc test [34] to determine which approaches show significant differences after the Friedman test rejects the null hypothesis. Lastly, the significance of the results is plotted on a Critical Distance (CD) diagram.

The experiments are performed on a system with an M1 Pro SoC (8 performance cores and 2 efficiency cores at 3.2 GHz) and 32 GB of unified memory. Each experiment is given a maximum of 24 GB of VRAM.

5 Results

This section presents and analyses the empirical findings of the study to answer the research questions defined in Section 4. For each question, the corresponding experiments, quantitative outcomes, and relevant observations are reported and interpreted in relation to the study objectives.

5.1 Results for RQ1: Type Inference Effectiveness

Table 2 shows the Top-1 through Top-5 accuracies, along with the MRR@5 constructed from these, for each model configuration and approach, on the average of the benchmarks.

On average for each configuration, *Source File* inference achieves a Top-1 accuracy of 30.3%, *AST* inference achieves 20.1%, and *RAG* inference achieves 39.6%. Additionally, the approaches achieve an average MRR@5 score of 0.47 for *Source file* inference, 0.34 for *AST* inference, and 0.57 for *RAG* inference.

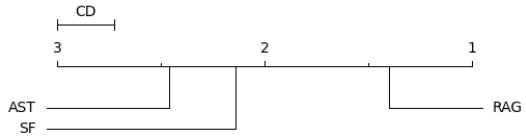

Figure 4: Critical Distance diagram for the 3 approaches across all configurations and benchmarks. A difference larger than the critical distance indicates a significant difference in the results of the approaches.

Figure 4 plots the result of the Friedman test and Nemenyi post hoc test for significant difference on a Critical Distance diagram, and shows that all approaches yield significantly different results; with RAG achieving the best mean rank (1.3993), followed by Source File (2.1424) and AST (2.4583). Effect sizes reinforce this interpretation. The comparison between RAG and AST shows a large effect, while RAG versus Source File shows a medium effect. In contrast, the difference between the Source File approach and the AST approach is small. The full statistical analysis results are provided in Appendix B.1 and in the Notebook provided in the project source code.

One of the surprising results is the large decrease in accuracy from the Source File approach to AST. The AST approach is designed to extract more high-quality data from source code, such that the models offer more precise predictions and are less likely to hallucinate. However, increasing the context provided to the model doesn't always yield better results, especially when the data is in a different form than what it is likely trained on [27], i.e. Abstract Syntax Tree form. Furthermore, ASTs of source code tend to be significantly larger in terms of tokens than the source code itself; this increases the chance of attention bias, which is an LLM's natural bias to give more importance to the first N and the last M tokens of the request than they give the middle [52]. The results for the AST inference indicate that these drawbacks have had a larger impact on the approach's performance than the additional high-quality data.

Insight 1. AST-based context, despite its structural richness, underperforms against Source File inference by 10.2% on average. Unfamiliar representation and inflated token counts introduce attention bias that outweighs the benefit of structured data.

From the model configurations, the **best performing local model is Qwen 3 Coder** with an MRR@5 of 0.57, 0.45 and 0.66 across Source File, AST and RAG approaches, respectively. The **best performing cloud model is GPT-5 mini**, with an MRR@5 of 0.55, 0.44 and 0.61. Both the size of the models and the reasoning capabilities have shown to be important factors in their performance. Additionally, the coding models: Qwen 3 Coder, Qwen 2.5 Coder, StarCoder 2, and to some extent GPT-5 mini perform significantly better than their non-coding counterparts. Additionally, Qwen 3 Coder running locally marginally outperformed GPT-5 mini, the highest-scoring cloud model, despite OpenAI's model being theorised to be much larger. This may be largely influenced by the fact that Qwen 3 Coder is trained on more source code proportionally than GPT-5 mini is. This also sheds insights on GPT-5 mini's slight win over Qwen 2.5 Coder in the AST approach, which benefits from better reasoning over contextual data likely not similar to what the model is trained on.

Table 3 shows the results of each local and cloud model configuration against each type of inference approach for every benchmark file in the evaluation. The approaches are labelled SF (Source File inference), AST (AST inference), and RAG (RAG inference). The units column represents the number of functions evaluated within a specific file. The metrics represent the average Top-1 correctness across 10 runs of the file. The best-performing approach per model is highlighted.

The file that is type-inferred most accurately across the board is TypeScript's `emitter.ts`. The units in this file contain detailed documentation and various abstracted functionality through call sites. This usage of natural language, paired with expressively-named helper functions, may help the non-coding models maintain competitive results. Conversely, the worst performing units are under VSCode's `editorService.ts` and Vue's `patch.ts`. Both of these files' custom types are aggregates of other custom types from other classes throughout the project. This affects the performance of the Source File and AST approaches as they only add, at most, 2 layers of imports to their context. It is worth noting that for the `editorService.ts` units, the RAG approach performs significantly better; this may be due to the nested custom types often containing the word "Editor", which the RAG approach can query for project-wide without encountering the import limitation of the other two approaches.

Insight 2. Retrieval matters more than representation quality. The primary performance driver across all approaches was not how well context was structured, but whether relevant context could be retrieved selectively.

We can observe that the greatest factors influencing performance across the approaches are the size of the provided context and the ability to query for relevant context. The former provides diminishing returns as context size grows in proportion to the model's size, and the latter allows even the smallest model (StarCoder 2) to perform competitively.

5.2 Results for RQ2: Token and Time Consumption

Table 2 also shows the number of tokens each approach passes as context, along with the average per-unit inference time.

The preprocessing technique of the AST approach provides significantly more tokens to the model's context than the other approaches. This has the drawback of increasing costs for cloud models, increasing inference time and increasing the likelihood of hallucination [50].

Table 2: Results across all approaches, benchmarks and configurations. RAG is configured with the *llama-embed-nemotron* embedding model which queries for the top-10 relevant snippets. Metrics are per-function averages from 10 runs. The inference time for the RAG approach takes into account the time for querying the function-under-test.

Approach	Model	Metrics							
		Top-1	Top-2	Top-3	Top-4	Top-5	MRR@5	# Tokens	Computation time (s)
Source file								**∼5.6K**	
	Qwen 3 (30B)	0.34	0.56	0.68	0.72	0.75	0.51		34.74
	Qwen 3 Coder (30B)	**0.38**	**0.62**	0.74	**0.81**	**0.87**	**0.57**		33.61
	Qwen 2.5 Coder (32B)	0.31	0.50	0.63	0.73	0.77	0.48		36.13
	DeepSeek-R1 (32B)	0.22	0.39	0.53	0.60	0.62	0.37		42.94
	StarCoder 2 (15B)	0.24	0.43	0.58	0.69	0.73	0.42		26.64
	GPT-3.5-turbo	0.27	0.45	0.59	0.65	0.71	0.43		13.49
	GPT-4	0.30	0.47	0.62	0.71	0.74	0.46		42.38
	GPT-5 mini	0.36	0.60	**0.75**	**0.81**	0.86	0.55		**12.18**
AST								**∼13.1K**	
	Qwen 3 (30B)	0.22	0.41	0.56	0.63	0.66	0.39		73.20
	Qwen 3 Coder (30B)	0.26	**0.49**	**0.65**	**0.70**	**0.73**	**0.45**		72.36
	Qwen 2.5 Coder (32B)	0.23	0.45	0.57	0.66	0.70	0.41		83.95
	DeepSeek-R1 (32B)	0.15	0.26	0.32	0.35	0.39	0.24		91.07
	StarCoder 2 (15B)	0.16	0.29	0.34	0.39	0.42	0.26		49.54
	GPT-3.5-turbo	0.11	0.16	0.21	0.24	0.25	0.16		32.25
	GPT-4	0.20	0.41	0.52	0.58	0.63	0.37		54.62
	GPT-5 mini	**0.28**	0.47	0.62	0.67	0.69	0.44		**20.73**
RAG								**∼3.9K**	
	Qwen 3 (30B)	0.42	0.67	0.84	**0.89**	0.94	0.62		34.84
	Qwen 3 Coder (30B)	**0.49**	**0.68**	**0.86**	**0.89**	**0.93**	**0.66**		34.71
	Qwen 2.5 Coder (32B)	0.43	0.65	0.81	0.84	0.85	0.60		32.26
	DeepSeek-R1 (32B)	0.35	0.51	0.63	0.73	0.76	0.50		37.35
	StarCoder 2 (15B)	0.35	0.53	0.62	0.71	0.75	0.51		26.79
	GPT-3.5-turbo	0.32	0.54	0.60	0.67	0.72	0.48		**12.06**
	GPT-4	0.38	0.61	0.76	0.88	**0.93**	0.58		33.24
	GPT-5 mini	0.43	0.62	0.79	0.87	0.90	0.61		12.23

Another downside is that the provided context might be too large for a model's context window, as is the case with the GPT-3.5-turbo results, in which case the context gets cut short and relevant information can be left out.

Inference time is also measured and reported as a per-unit average, which includes both processing and response generation. The responses are generated without streaming, which means the whole response is received at once. For the local models, their size and their reasoning capacity influence the inference time. Results show that the reasoning models take longer to generate a response than the non-reasoning ones. Likewise, the non-coding reasoning models take longer than the coding ones, suggesting they are predisposed to reason about the context for longer.

To further assess the trade-off in performance and inference time, we calculate an efficiency metric defined as MRR@5 divided by inference time. The best-performing configuration overall, RAG with Qwen 3 Coder, achieves an MRR@5 of 0.66 with a inference time of 34.71 seconds, resulting in an efficiency score of approximately 0.019. In contrast, the fastest configuration, RAG with GPT-3.5-turbo, achieves an MRR@5 of 0.48 in 12.06 seconds, corresponding to an efficiency of approximately 0.040. The most efficient configuration is GPT-5 mini using the RAG approach; with an MRR@5 of 0.61 and inference time of 12.23 seconds, it achieves an efficiency of 0.050. Although its absolute predictive performance is lower than Qwen 3 Coder, it delivers much more performance relative to its inference time.

From the perspective of search-based automatic test case generation, type inference must be evaluated not only by predictive accuracy but also by its speed. More efficient configurations deliver lower absolute accuracy but substantially higher performance per second. Because automatic test case generation iteratively evaluates and improves candidate inputs, even imperfect type predictions can significantly accelerate convergence if they eliminate invalid regions of the search space. Therefore, the most suitable configuration for automatic test case generation is not necessarily the most accurate one, but the one that offers the best balance between search space reduction and computational efficiency.

Table 3: Top-1 correctness of the approaches for each model. SF represents *Source File inference*, AST represents *AST inference*, and RAG represents *RAG inference*. RAG is configured with the *llama-embed-nemotron* embedding model. Results are shown for 30 TypeScript, 15 VSCode and 15 Vue functions for 10 runs each.

Panel A: Local models.

Benchmark & File name	# Units	Qwen 3 (30B)			Qwen 3 Coder (30B)			Qwen 2.5 Coder (32B)			DeepSeek-R1 (32B)			StarCoder 2 (15B)		
		SF	AST	RAG	SF	AST	RAG	SF	AST	RAG	SF	AST	RAG	SF	AST	RAG
TypeScript	**30**															
parser.ts	5	0.05	0.02	**0.10**	0.15	0.07	**0.37**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	0.02	0.02	**0.05**
checker.ts	6	0.12	0.00	**0.26**	0.16	0.04	**0.22**	0.10	**0.31**	0.29	0.02	0.00	**0.17**	0.00	0.00	**0.14**
emitter.ts	4	**0.45**	**0.45**	0.43	0.45	0.32	**0.56**	0.28	0.42	**0.50**	**0.22**	0.07	**0.22**	**0.34**	0.21	0.25
binder.ts	3	0.27	**0.34**	**0.34**	0.36	**0.43**	0.34	**0.27**	**0.27**	0.21	0.13	0.11	**0.16**	0.11	**0.18**	0.14
services.ts	5	0.00	0.00	**0.12**	0.04	**0.10**	0.08	0.00	0.04	**0.10**	0.00	0.00	**0.05**	**0.00**	**0.00**	**0.00**
completions.ts	4	**0.10**	0.00	0.05	**0.18**	0.05	0.05	**0.08**	0.02	0.03	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**
session.ts	3	0.06	0.15	**0.28**	0.02	0.26	**0.54**	0.00	0.18	**0.43**	0.00	0.08	**0.21**	0.02	0.00	**0.25**
VSCode	**15**															
uri.ts	4	0.16	0.02	**0.25**	0.25	0.22	**0.36**	0.23	**0.26**	**0.26**	0.02	0.00	**0.03**	0.14	0.10	**0.21**
instantiationService.ts	3	**0.18**	0.07	0.14	0.11	**0.17**	0.13	**0.12**	0.00	0.10	0.04	0.05	**0.13**	0.06	0.00	**0.11**
editorService.ts	3	0.00	0.02	**0.18**	0.10	0.00	**0.22**	0.00	0.08	**0.16**	**0.10**	0.00	0.00	0.00	**0.01**	0.00
editorBrowser.ts	2	**0.50**	0.35	0.22	**0.30**	0.11	**0.30**	0.26	0.21	**0.28**	0.16	**0.23**	**0.23**	0.15	0.18	**0.20**
workbench.ts	3	0.12	0.23	**0.58**	0.12	0.38	**0.74**	0.09	0.13	**0.21**	0.04	0.02	**0.13**	0.14	0.18	**0.27**
Vue	**15**															
parse.ts	2	0.25	0.16	**0.37**	0.20	0.12	**0.53**	0.28	**0.31**	0.26	0.24	0.15	**0.27**	**0.31**	0.15	0.27
compileScript.ts	3	0.10	0.00	**0.18**	0.14	**0.18**	**0.18**	0.09	0.00	**0.13**	0.00	0.00	**0.08**	**0.00**	**0.00**	**0.00**
src/render.ts	3	0.06	**0.08**	**0.08**	**0.06**	0.00	0.03	**0.07**	0.00	**0.07**	**0.03**	0.00	0.00	**0.01**	**0.01**	0.00
init.ts	2	**0.66**	0.14	0.53	**0.54**	0.05	0.52	0.25	0.08	**0.34**	0.26	0.21	**0.37**	0.15	0.06	**0.32**
instance/render.ts	2	0.15	0.15	**0.25**	0.23	0.20	**0.38**	0.16	**0.23**	**0.23**	**0.18**	0.03	0.14	0.23	0.10	**0.28**
patch.ts	3	0.00	**0.05**	0.00	0.00	0.00	**0.13**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**	**0.00**

Panel B: Cloud models

Benchmark	File name	# Units	GPT-3.5-turbo			GPT-4			GPT-5 mini		
			SF	AST	RAG	SF	AST	RAG	SF	AST	RAG
TypeScript		**30**									
	parser.ts	5	**0.02**	0.00	0.00	0.04	**0.06**	**0.06**	0.12	0.03	**0.21**
	checker.ts	6	0.18	**0.23**	0.21	0.11	0.03	**0.22**	0.19	0.01	**0.23**
	emitter.ts	4	0.47	0.36	**0.56**	0.54	0.52	**0.67**	0.47	0.44	**0.58**
	binder.ts	3	0.30	0.28	**0.31**	0.22	0.26	**0.27**	0.32	0.29	**0.38**
	services.ts	5	0.00	0.00	**0.23**	0.00	0.01	**0.08**	0.03	**0.05**	**0.05**
	completions.ts	4	**0.01**	0.00	**0.01**	**0.08**	0.00	**0.08**	0.14	0.11	**0.26**
	session.ts	3	0.00	0.04	**0.20**	0.04	0.00	**0.25**	0.07	0.05	**0.29**
VSCode		**15**									
	uri.ts	4	0.22	0.14	**0.25**	0.14	0.04	**0.23**	**0.21**	0.13	**0.21**
	instantiationService.ts	3	**0.14**	0.00	0.12	**0.15**	0.05	0.13	0.15	**0.16**	0.15
	editorService.ts	3	0.00	0.00	**0.08**	0.00	0.00	**0.26**	0.16	0.04	**0.27**
	editorBrowser.ts	2	0.23	0.17	**0.27**	0.47	0.23	**0.54**	0.27	0.16	**0.38**
	workbench.ts	3	0.05	0.03	**0.11**	0.00	0.01	**0.12**	0.38	0.25	**0.58**
Vue		**15**									
	parse.ts	2	**0.26**	**0.26**	0.23	**0.17**	0.04	**0.17**	0.15	0.07	**0.23**
	compileScript.ts	3	0.06	0.00	**0.18**	0.03	0.00	**0.24**	0.15	0.17	**0.21**
	src/render.ts	3	0.01	0.00	**0.03**	0.02	**0.07**	**0.07**	0.03	0.00	**0.07**
	init.ts	2	**0.24**	0.00	0.19	0.35	0.16	**0.38**	0.35	0.00	**0.49**
	instance/render.ts	2	0.19	0.19	**0.28**	0.14	0.15	**0.19**	**0.16**	0.13	0.15
	patch.ts	3	0.00	0.00	**0.01**	**0.00**	**0.00**	**0.00**	0.02	0.00	**0.05**

Insight 3. The trade off between accuracy and computation time of LLM-based type inference is caused by the size of the model and the number of tokens passed to the context.

5.3 Results for RQ3: Comparison with Probabilistic Technique

Table 4 shows the performance of the 3 LLM-based approaches configured with Qwen 3 Coder against SynTest's *proportional ranking* approach across each file in the benchmarks. The performance is measured in Top-1 correctness, as the *proportional ranking* approach returns one inferred type. The results indicate that SynTest's *proportional sampling* performs slightly better than the LLM-based Source File inference, with average accuracies of 0.19 and 0.20, respectively; while the RAG inference approach achieves a higher 0.32 average accuracy.

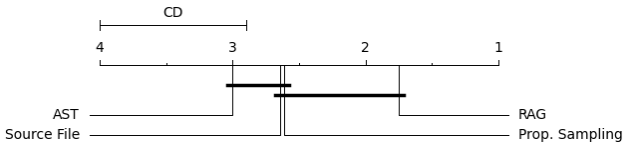

Figure 5: CD diagram visualizing the results of the Friedman and Nemenyi statistical tests on the results of the 4 approaches. A difference larger than the critical distance indicates significant difference in the results of the approaches.

Figure 5 visualises these results after performing the Friedman and Nemenyi non-parametric tests. There, we can see that the results of all the approaches, apart from those of AST and RAG, are not significantly different from one another, but they may still be practically meaningful.

The Friedman test indicates a statistically significant difference among the four approaches ($p = 0.024$), confirming that the observed variation in performance across configurations is unlikely to be due to chance. Based on mean ranks (lower is better), RAG achieves the best overall ranking (1.75), followed by Proportional Sampling (2.61), Source File (2.64), and AST (3.00). This ranking pattern mirrors the mean-based results, with RAG consistently outperforming the alternative approaches across benchmarks.

The Nemenyi post-hoc analysis further supports this interpretation. The largest and statistically meaningful separation is observed between RAG and AST, corresponding to a large effect size. The difference between RAG and Source File is of medium magnitude, while the gap between RAG and Proportional Sampling is smaller but still favours RAG. In contrast, the differences among AST, Source File, and Proportional Sampling are small and not statistically pronounced, as reflected by their close ranks and overlapping groupings in the critical difference diagram. This again suggests that the primary performance gain stems from the introduction of retrieval augmentation rather than from differences in how static context is represented.

Certain files, such as `parser.ts` from TypeScript, perform poorly for the Proportional Sampling approach (Top-1 accuracy of 0.03) while performing substantially better for RAG (Top-1 accuracy of 0.37). Looking at the file, we can notice that it contains a large number of classes extending common interfaces. An excerpt of these interfaces can be seen in Listing 1. These interfaces extend `TypeNode`, which itself extends `Node`, resulting in many commonly named methods between the types. The probabilistic approach leverages the names and calls to these methods to infer the types of the objects upon which they was called, often guessing wrongly. The RAG approach, however, is able to retrieve all these definitions into it's context through semantic search similarity, and infer the correct one by means of natural language processing and behaviour analysis.

Listing 1: Excerpt of the the class interfaces declared in Typescript's `parser.ts` file.

```
1   interface NamedTupleMember extends TypeNode,
        Declaration, JSDocContainer {
2       readonly kind: SyntaxKind.NamedTupleMember;
3       readonly dotDotDotToken?: Token<SyntaxKind.
            DotDotDotToken>;
4       readonly name: Identifier;
5       readonly questionToken?: Token<SyntaxKind.
            QuestionToken>;
6       readonly type: TypeNode;
7   }
8   interface OptionalTypeNode extends TypeNode {
9       readonly kind: SyntaxKind.OptionalType;
10      readonly type: TypeNode;
11  }
12  interface RestTypeNode extends TypeNode {
13      readonly kind: SyntaxKind.RestType;
14      readonly type: TypeNode;
15  }
```

Regarding dispersion, RAG exhibits the highest variability (MAD = 0.19) compared to the other approaches (0.08–0.10). Two factors may explain this. First, RAG introduces an additional stochastic retrieval component, which can increase variability across runs and benchmarks. Second, RAG constructs a selective, condensed context rather than providing complete structural information. When the retrieved snippets align closely with the semantic intent of the target function, performance can be substantially improved; however, when retrieval yields partially relevant or less informative snippets, performance may decline more noticeably. The full statistical analysis results are provided in Appendix B.2 and in the Notebook provided in the project source code.

Despite this increased variability, RAG's advantage remains robust. Its median performance (0.320) is substantially higher than that of AST (0.115), Source File (0.155), and Proportional Sampling (0.160), and its superior mean rank confirms consistent outperformance across tasks. Overall, the analysis reinforces the conclusion that retrieval-augmented prompting yields an improvement over both probabilistic and non-retrieval LLM-based approaches.

Insight 4. Probabilistic inference remains competitive for primitive types, where speed allows multiple iterations before any LLM responds. LLM-based approaches — particularly RAG — offer a clear advantage only when types are complex, user-defined, or package-reliant.

Table 4: Top-1 correctness of LLM approaches compared to SynTest's probabilistic *proportional sampling* approach. All LLM approaches run on *Qwen 3 Coder*, the RAG approach is configured with the *llama-embed-nemotron* embedding model. Results are shown for 30 TypeScript, 15 VSCode and 15 Vue functions for 10 runs each.

Benchmark	File name	# Units	Source File	AST	RAG	Proportional sampling
TypeScript		**30**				
	parser.ts	5	0.15	0.07	**0.37**	0.03
	checker.ts	6	0.16	0.04	0.22	**0.28**
	emitter.ts	4	0.45	0.32	**0.56**	0.49
	binder.ts	3	0.36	**0.43**	0.34	0.15
	services.ts	5	0.04	**0.10**	0.08	0.00
	completions.ts	4	**0.18**	0.05	0.05	0.10
	session.ts	3	0.02	0.26	**0.54**	0.17
VSCode		**15**				
	uri.ts	4	0.25	0.22	0.36	**0.43**
	instantiationService.ts	3	0.11	0.17	0.13	**0.36**
	editorService.ts	3	0.10	0.00	**0.22**	0.00
	editorBrowser.ts	2	**0.30**	0.11	**0.30**	0.21
	workbench.ts	3	0.12	0.38	**0.74**	0.24
Vue		**15**				
	parse.ts	2	0.20	0.12	**0.53**	0.06
	compileScript.ts	3	0.14	**0.18**	**0.18**	0.15
	src/render.ts	3	**0.06**	0.00	0.03	**0.06**
	init.ts	2	**0.54**	0.05	0.52	0.48
	instance/render.ts	2	0.23	0.20	**0.38**	0.26
	patch.ts	3	0.00	0.00	**0.13**	0.07

5.4 Results for RQ4: Retrieved Context in RAG

Table 5 summarises the frequency of selected snippet types across TypeScript, VSCode, and Vue for the evaluated embedding models.

Across all projects, **function definitions** are the most frequently retrieved snippet type. This is particularly pronounced in the TypeScript project, where function snippets dominate the results. This behaviour can be explained by the architecture of the compiler, in which the functions under test represent primary units of logic and are highly coupled with other functions. Because embedding similarity is influenced by lexical overlap and structural resemblance, functions with similar names and definitions are strongly matched.

In TypeScript, **call sites** are also frequently retrieved. The compiler makes extensive use of recursion and delegation, meaning that functions under test often contain numerous calls to other functions with related semantics. These call expressions closely resemble call-site snippets stored in the database, increasing their retrieval likelihood. Additionally, the compiler architecture is dominated by large `switch` statements over `SyntaxKind`. These blocks often represent substantial portions of the function logic and may appear in retrieval results as control-flow hints. While such hints are less frequent than function definitions, they are noticeably more common in TypeScript than in the other projects. This aligns with the compiler's heavy reliance on type-based branching. The relatively high number of retrieved JSDoc snippets suggests that core TypeScript compiler APIs are generally well documented; matching function names often align with `@param` and `@returns` descriptions. In contrast, variables, imports, and exports appear infrequently, likely because they provide less distinctive semantic information about the core function logic.

The VSCode results exhibit a different distribution. While function definitions remain common, **class snippets** are retrieved far more frequently than in the other projects. This reflects VSCode's architecture, which is structured around services, controllers, and views. Logic is grouped within classes rather than separated into stand-alone functional units, increasing the structural similarity between queried functions and class definitions. Consequently, both method definitions and their enclosing classes are retrieved. Compared to TypeScript, control-flow hints are substantially less frequent. VSCode avoids large `switch(kind)` constructs and instead relies on polymorphism, where behaviour is distributed across class hierarchies. As a result, fewer explicit `typeof` checks or switch-based control structures are present in proximity to the queried functions, reducing their retrieval rate.

Vue presents yet another pattern. A high number of **function** and **call-site** snippets are retrieved, which can be attributed to the extensive use of nested function definitions, delegation, and compile-time macros that are parsed as call expressions. Additionally, Vue shows a comparatively high number of retrieved variable snippets. Unlike the class-centric structure of VSCode, Vue frequently assigns functional logic to variables through destructuring or helper assignments, especially in the compiler and render-related modules. This may increase the semantic similarity between queried functions and variable declarations holding executable logic. Despite the presence of complex conditional logic in Vue's diffing and rendering mechanisms, control-flow hints are relatively rare. This is because

Table 5: The number of units extracted from each benchmark's vector database, grouped by information category. Shows counts for each categories after querying 30 TypeScript, 15 VSCode and 15 Vue functions for 10 runs each. For the embedding models, Llama represents *llama-embed-nemotron*, Qwen represents *Qwen3-embedding*, and Bert represents *bert-base-uncased*.

Snippet type	TypeScript			VSCode			Vue		
	LLama	Qwen	Bert	LLama	Qwen	Bert	LLama	Qwen	Bert
Function	**104**	**107**	**129**	49	41	**68**	**77**	**84**	**79**
Call Site	98	88	102	16	18	23	39	32	31
Object Shape	0	2	0	0	0	1	1	3	4
Variable	2	6	2	10	8	9	24	29	20
Class	13	17	11	**56**	**63**	35	5	1	3
JSDoc	33	17	19	13	11	9	2	0	2
Imports	0	1	1	1	2	0	0	1	0
Exports	0	2	0	2	0	0	0	0	0
Literal value	3	5	4	3	4	3	2	0	1
Control-flow hint	47	45	32	0	3	2	1	0	8

Vue typically performs property-based comparisons such as equality checks between nodes rather than primitive type checks such as `typeof`, which are parsed as control-flow hints. Furthermore, due to Vue's mixin-based architecture, class definitions are uncommon, which is reflected in the low retrieval frequency of class snippets. Finally, unlike TypeScript, Vue does not consistently document every function, explaining the limited number of JSDoc matches.

The observed differences between the embedding models can be due to their scale, training objectives, and architectural capacity. Llama Embed Nemotron (8B parameters) and Qwen 3 Embed (4B parameters) are large, instruction-tuned transformer models trained on diverse corpora, enabling them to capture higher-level semantic relationships beyond lexical similarity. Their larger parameter counts allow them to encode more nuanced structural and contextual information, which may explain their stronger alignment with semantically meaningful snippet types such as function definitions or classes. In contrast, BERT-uncased (110M parameters), originally trained on general natural language data with a masked language modelling objective, has substantially lower representational capacity and limited exposure to source code structure. As a result, its embeddings are more likely to rely on surface-level token overlap rather than deeper program semantics, potentially leading to different retrieval distributions. The parameter gap not only reflects differences in model size but also in semantic abstraction capability, domain adaptation, and robustness to variation, all of which directly influence retrieval behaviour.

Overall, the results indicate that the RAG approach predominantly retrieves structurally central elements—primarily function definitions and call sites, while the distribution of other snippet types strongly reflects project architecture. Compiler-style systems such as TypeScript favour control-flow and documentation matches, class-oriented systems like VSCode emphasise class context, and mixin or function-heavy architectures such as Vue increase variable and nested function retrieval. At the same time, differences between embedding models suggest that retrieval behaviour is also shaped by model capacity and domain alignment: larger, code-oriented models better capture semantic and structural relationships, whereas smaller, general-purpose models like BERT-uncased rely more on surface-level token similarity. Thus, both software architecture and embedding model characteristics jointly influence which information types are selected by the RAG pipeline.

Insight 5. Retrieved context distribution reflects project architecture. This makes RAG "architecture-aware" in practice, even without explicit design for it.

6 Threats to Validity

The design and execution of this work introduce potential limitations that may affect the interpretation and generalisation of the results. To ensure transparency and credibility, this section discusses the main threats to validity associated with the study. It focuses on external validity, concerning the extent to which the results generalise beyond the evaluated benchmarks and configurations, and conclusion validity, relating to the reliability of the statistical analysis and the inferences drawn from the experimental results. For each identified threat, we outline the measures taken during the study design and evaluation process to mitigate its potential impact.

6.1 External Validity

A potential threat to external validity stems from the benchmark selection, as the study evaluates only three large projects: the TypeScript compiler, VSCode, and Vue. However, these systems represent substantially different architectural and programming paradigms commonly observed in modern JavaScript/TypeScript ecosystems. The TypeScript compiler follows a largely functional, compiler-oriented design with extensive control-flow logic and recursive function structures. VSCode, in contrast, adopts a strongly object-oriented architecture composed of services, controllers, and interacting classes. Vue represents a framework-style architecture with extensive use of mixins, factory patterns, and nested functions. These structural differences expose the evaluated approaches to diverse coding patterns, dependency structures, and type usage scenarios. As a result, the benchmark captures a broader range of real-world development practices, helping mitigate the risk that the findings are specific to a single project architecture or coding style.

Another potential threat to external validity is data leakage, where models may perform well not due to genuine inference capabilities

but because they have previously seen the evaluated code during training. To mitigate this risk, the study constructs benchmarks from TypeScript projects which are transpiled to JavaScript before being provided to the evaluated approaches. This process serves two purposes. First, it enables the extraction of ground truth type information directly from the original TypeScript source, ensuring reliable evaluation. Second, by providing the models with the transpiled JavaScript rather than the original TypeScript code, the likelihood that the models can simply recall previously seen type annotations is reduced, encouraging genuine inference based on the available context. Additionally, the training datasets of the evaluated models were investigated to determine whether the selected benchmarks were explicitly included. Among the evaluated models, only StarCoder 2 provides full transparency regarding its training corpus, which includes the Stack v2[13] dataset. An inspection of this dataset revealed that none of the three benchmark projects—TypeScript, VSCode, or Vue—or the functions under test are present, further reducing the likelihood that the results are influenced by recall rather than inference.

6.2 Conclusion Validity

A potential threat to conclusion validity arises from the non-deterministic nature of large language models, which can produce different outputs for the same prompt due to stochastic decoding and internal sampling processes. This variability may introduce noise into the experimental results and affect the reliability of performance comparisons. To mitigate this risk, each experiment in this study was executed ten times, allowing the results to capture the variability inherent in the models' responses. The aggregated outcomes were then analysed using non-parametric statistical methods, specifically the Friedman test to detect overall differences among the evaluated approaches and the Nemenyi post-hoc test to perform pairwise comparisons based on rank differences. By relying on repeated measurements, the study reduces the impact of stochastic variation and increases confidence that the reported differences reflect systematic performance trends rather than random fluctuations.

Another potential threat to conclusion validity is prompt design bias. Since large language models are sensitive to the structure and wording of prompts, differences in prompt formulation may influence model performance independently of the evaluated approach. In this study, slightly different prompts were required for each approach to accommodate the different forms of context provided (e.g., AST representations, source files, or retrieved snippets). However, the core instructions, task description, and output requirements were kept consistent across prompts. Furthermore, all configurations of a given approach used the exact same prompt, ensuring that comparisons within each approach reflect differences in model behaviour rather than prompt variations. This design helps ensure that performance differences primarily stem from the contextual information provided to the models rather than from prompt formulation.

7 Conclusion and Future Work

This study investigated the use of large language models for type inference in JavaScript, with the goal of improving automatic test case generation. By designing and evaluating three approaches across multiple models and real-world projects, the study assessed both inference effectiveness and efficiency. The experimental pipeline enabled a systematic comparison against probabilistic techniques, using ground-truth types derived from TypeScript to evaluate accuracy and performance.

The findings show that retrieval-augmented approaches consistently outperform both traditional LLM prompting strategies and probabilistic inference, particularly for complex, user-defined, and package-dependent types. The results highlight that selectively retrieving relevant context is more impactful than increasing the amount or structural richness of provided information. Additionally, while LLM-based approaches excel in handling complex structures, probabilistic methods remain competitive for primitive types due to their speed and iterative capabilities.

Overall, the study demonstrates that LLM-based type inference is a viable direction for enhancing test generation in JavaScript, especially when combined with retrieval mechanisms. However, the trade-off between accuracy and efficiency suggests that hybrid approaches, leveraging probabilistic methods for simple cases and LLMs for complex ones, offer the most practical path forward. Combined with the increasing performance of LLMs and machine learning hardware, these insights provide a foundation for future research into type inference systems.

[13] https://huggingface.co/datasets/bigcode/the-stack-v2

References

[1] Alexander Aiken, Edward L Wimmers, and TK Lakshman. 1994. Soft typing with conditional types. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 163–173.

[2] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 263–272.

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[4] Luciano Baresi, Pier Luca Lanzi, and Matteo Miraz. 2010. Testful: An evolutionary test approach for java. In *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 185–194.

[5] Sung Nok Chiu, Dietrich Stoyan, Wilfrid S Kendall, and Joseph Mecke. 2013. *Stochastic geometry and its applications*. John Wiley & Sons.

[6] Tristan Coignion, Clément Quinton, and Romain Rouvoy. 2024. A performance study of llm-generated code on leetcode. In *Proceedings of the 28th international conference on evaluation and assessment in software engineering*. 79–89.

[7] Nick Craswell. 2016. Mean reciprocal rank. In *Encyclopedia of database systems*. Springer, 1–1.

[8] Richard A DeMilli and A. Jefferson Offutt. 1991. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering* 17, 9 (1991), 900–910.

[9] Elfriede Dustin, Thom Garrett, and Bernie Gauf. 2009. *Implementing automated software testing: How to save time and lower costs while raising quality*. Pearson Education.

[10] Amin Milani Fard and Ali Mesbah. 2017. JavaScript: The (un) covered parts. In *2017 IEEE international conference on software testing, verification and validation (ICST)*. IEEE, 230–240.

[11] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[12] Milton Friedman. 1937. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the american statistical association* 32, 200 (1937), 675–701.

[13] Zheng Gao, Christian Bird, and Earl T Barr. 2017. To type or not to type: quantifying detectable bugs in JavaScript. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 758–769.

[14] Richard Hamlet. 1994. Random testing. *Encyclopedia of software Engineering* 2 (1994), 971–978.

[15] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)* 45, 1 (2012), 1–61.

[16] Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 152–162.

[17] Xijie Huang, Li Lyna Zhang, Kwang-Ting Cheng, Fan Yang, and Mao Yang. 2023. Fewer is more: Boosting llm reasoning with reinforced context pruning. *arXiv preprint arXiv:2312.08901* (2023).

[18] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2009. Type analysis for JavaScript. In *International Static Analysis Symposium*. Springer, 238–255.

[19] Fitsum Kifetew, Xavier Devroey, and Urko Rueda. 2019. Java unit testing tool competition-seventh round. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 15–20.

[20] James C King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (1976), 385–394.

[21] Lukas Krodinger, Stephan Lukasczyk, and Gordon Fraser. 2025. Combining Type Inference and Automated Unit Test Generation for Python. *arXiv preprint arXiv:2507.01477* (2025).

[22] Mike Levy. 2025. Measuring Similarity and Distance between Embeddings. https://www.dataquest.io/blog/measuring-similarity-and-distance-between-embeddings/#:~:text=Euclidean%20distance%20measures%20the%20straight,Euclidean%20distance%20between%20two%20vectors.

[23] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[24] Ye Liu, Rui Meng, Shafiq Joty, Silvio Savarese, Caiming Xiong, Yingbo Zhou, and Semih Yavuz. 2024. Codexembed: A generalist embedding model family for multiligual and multi-task code retrieval. *arXiv preprint arXiv:2411.12644* (2024).

[25] Stephan Lukasczyk and Gordon Fraser. 2022. Pynguin: Automated unit test generation for python. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 168–172.

[26] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. 2019. NL2Type: Inferring JavaScript function types from natural language information. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 304–315.

[27] Ariana Martino, Michael Iannelli, and Coleen Truong. 2023. Knowledge injection to counter large language model (LLM) hallucination. In *European Semantic Web Conference*. Springer, 182–185.

[28] Elio Masciari, Vincenzo Moscato, Enea Vincenzo Napolitano, Gian Marco Orlando, Marco Perillo, and Diego Russo. 2026. Are LLMs Ready for TOON? Benchmarking Structural Correctness-Sustainability Trade-offs in Novel Structured Output Formats. *arXiv preprint arXiv:2601.12014* (2026).

[29] Ivan Matveev. 2026. Token-Oriented Object Notation vs JSON: A Benchmark of Plain and Constrained Decoding Generation. *arXiv preprint arXiv:2603.03306* (2026).

[30] Amir M. Mir, Evaldas Latoškinas, Sebastian Proksch, and Georgios Gousios. 2022. Type4Py: practical deep similarity learning-based type inference for python. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 2241–2252. doi:10.1145/3510003.3510124

[31] Gireen Naidu, Tranos Zuva, and Elias Mmbongeni Sibanda. 2023. A review of evaluation metrics in machine learning algorithms. In *Computer science on-line conference*. Springer, 15–25.

[32] Arvind Neelakantan, Tao Xu, Raul Puri, Alec Radford, Jesse Michael Han, Jerry Tworek, Qiming Yuan, Nikolas Tezak, Jong Wook Kim, Chris Hallacy, et al. 2022. Text and code embeddings by contrastive pre-training. *arXiv preprint arXiv:2201.10005* (2022).

[33] Lucian Negru. 2026. Replication package of "Retrieval First: LLM- Assisted Type Inference for Automatic Test Case Generation in JavaScript". doi:10.5281/zenodo.19496755

[34] Peter Bjorn Nemenyi. 1963. *Distribution-free multiple comparisons.* Princeton University.

[35] Simeon Ntafos. 1998. On random and partition testing. In *Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*. 42–48.

[36] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2017), 122–158.

[37] Yun Peng, Chaozheng Wang, Wenxuan Wang, Cuiyun Gao, and Michael R. Lyu. 2023. Generative Type Inference for Python. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 988–999. doi:10.1109/ASE56229.2023.00031

[38] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V Mäntylä. 2012. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th international workshop on automation of software test (AST)*. IEEE, 36–42.

[39] Veselin Raychev, Martin Vechev, and Andreas Krause. 2015. Predicting program properties from" big code". *ACM SIGPLAN Notices* 50, 1 (2015), 111–124.

[40] Muhammad Shahid, Suhaimi Ibrahim, and Mohd Naz'ri Mahrin. 2011. A study on test coverage in software testing. *Advanced Informatics School (AIS), Universiti Teknologi Malaysia, International Campus, Jalan Semarak, Kuala Lumpur, Malaysia* 1 (2011).

[41] Dimitri Stallenberg, Mitchell Olsthoorn, and Annibale Panichella. 2022. Guess what: Test case generation for Javascript with unsupervised probabilistic type inference. In *International Symposium on Search Based Software Engineering*. Springer, 67–82.

[42] Toni Taipalus. 2024. Vector database management systems: Fundamental concepts, use-cases, and current challenges. *Cognitive Systems Research* 85 (2024), 101216.

[43] Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st annual meeting of the association for computational linguistics (volume 1: long papers)*. 2609–2634.

[44] Jiayi Wei, Greg Durrett, and Isil Dillig. 2023. TypeT5: Seq2seq Type Inference using Static Analysis. arXiv:2303.09564 [cs.SE] https://arxiv.org/abs/2303.09564

[45] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[46] Andrew K Wright and Robert Cartwright. 1997. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 1 (1997), 87–152.

[47] Yanyan Yan, Yang Feng, Hongcheng Fan, and Baowen Xu. 2023. DLInfer: Deep Learning with Static Slicing for Python Type Inference. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2009–2021. doi:10.1109/ICSE48619.2023.00170

[48] Chen Yang, Junjie Chen, Bin Lin, Jianyi Zhou, and Ziqi Wang. 2024. Enhancing llm-based test generation for hard-to-cover branches via program analysis. *arXiv preprint arXiv:2404.04966* (2024).

[49] Ruofan Yang, Xianghua Xu, and Ran Wang. 2025. LLM-enhanced evolutionary test generation for untyped languages. *Automated Software Engineering* 32, 1 (2025), 20.

[50] Zezhou Yang, Sirong Chen, Cuiyun Gao, Zhenhao Li, Xing Hu, Kui Liu, and Xin Xia. 2025. An empirical study of retrieval-augmented code generation: Challenges and opportunities. *ACM Transactions on Software Engineering and Methodology* (2025).

[51] Xi Ye and Greg Durrett. 2022. The unreliability of explanations in few-shot prompting for textual reasoning. *Advances in neural information processing systems* 35 (2022), 30378–30392.

[52] Hanzhang Zhou, Zijian Feng, Zixiao Zhu, Junlang Qian, and Kezhi Mao. 2024. Unibias: Unveiling and mitigating llm bias through internal attention and ffn manipulation. *Advances in Neural Information Processing Systems* 37 (2024), 102173–102196.

A Prompt Templates

This appendix contains the prompt templates for each of the approaches.

A.1 Source File Inference

You are a static type inference assistant. Given JavaScript code, infer precise TypeScript-style types.
Analyze the following JavaScript code:
${sourceCode}
For each identifier, provide 5 possible type predictions ordered by confidence (most confident first).
Respond only with a JSON array using this exact schema for each identifier found:

```
{
  "entity": "function|
             variable|
             class|
             class-method",
  "name": "identifier\_name",
  "location": {
    "line": 1,
    "column": 0
  },
  "types": {
    "params": { "paramName": "type" },
    "return": ["type1",
               "type2",
               "type3",
               "type4",
               "type5"]
  }
}
```

IMPORTANT EXTRACTION RULES:
1. Extract ALL identifiers including:
- Top-level functions (entity: "function")
- Variables and constants (entity: "variable")
- Class declarations (entity: "class")
- Class methods (entity: "class-method", name format: "ClassName.methodName")
- Arrow functions assigned to variables (entity: "function")

2. For location field:
- Estimate line numbers by counting lines in the source code
- Use column 0 if exact position is unknown
- ALWAYS include location object with line and column numbers

3. For types object:
- ALWAYS include "return" field as an array of 5 type predictions
- Order the return types by confidence (most confident first)
- For functions and class-methods: include "params" object (can be empty)
- For variables and classes: omit "params" field entirely
- Example for function: "types": "params": "a": "number", "b": "number", "return": ["number", "any", "unknown", "void", "never"]
- Example for variable: "types": "return": ["string", "any", "unknown", "string | null", "string | undefined"]
- Example for class: "types": "return": ["ClassName", "any", "object", "unknown", "Object"]

4. TYPE INFERENCE RULES:
- Use specific TypeScript types: string, number, boolean, array, function, null, undefined, void
- For object types, prefer interface/class names if defined in the code
- For arrays, use "type[]" notation
- For class instances, use the class name as the type
- For class methods, use entity "class-method" and format name as "ClassName.methodName"
- Provide diverse alternatives (e.g., specific type, union types, general types like 'any')

5. REQUIRED JSON STRUCTURE:
- Every item MUST have: entity, name, location, types
- location MUST have: line (number), column (number)
- types MUST have: return (array of 5 strings)
- types MAY have: params (object) - only for functions and class-methods

Return only the JSON array, no markdown formatting or explanations.

A.2 AST Inference

You are a static type inference assistant. Given a detailed AST (Abstract Syntax Tree) representation of JavaScript code, infer precise TypeScript-style types.
Analyze the following AST nodes extracted from JavaScript code:
${snippetSummary}
The AST contains rich information including:
- Function bodies with return statements, variable usage patterns, and function calls
- Parameter usage patterns showing how parameters are used within functions
- Variable initializers with inferred types from literal values
- Type hints derived from operations and expressions

Use this detailed information to infer the most appropriate TypeScript types.
For each identifier, provide 5 possible type predictions ordered by confidence (most confident first).
Respond only with a JSON array using this exact schema for each identifier found:

```
{
  "entity": "function|
             variable|
             class|
             class-method",
  "name": "identifier_name",
  "location": {
    "line": line_number,
    "column": column_number
  },
  "types": {
    "params": { "paramName": "type" },
    "return": ["type1",
               "type2",
               "type3",
               "type4",
```

```
                "type5"]
    }
}
```

CRITICAL EXTRACTION REQUIREMENTS:
1. MUST include ALL nodes from the AST analysis above
2. Use correct entity types:
- FunctionDeclaration nodes → entity: "function"
- VariableDeclaration nodes → entity: "variable"
- ClassDeclaration nodes → entity: "class"
- ClassMethod nodes → entity: "class-method"
- ArrowFunction nodes → entity: "function"

3. For ClassMethod nodes:
- MUST use entity: "class-method"
- Keep the full "ClassName.methodName" format as name

4. TYPES OBJECT RULES:
- ALWAYS include "return" field as an array of 5 type predictions
- Order the return types by confidence (most confident first)
- For classes: first type should be "ClassName", then alternatives like "any", "object", etc.
- For variables: provide 5 alternative types based on usage (e.g., ["string", "any", "unknown", "string | null", "string | undefined"])
- For functions and class-methods: use both "params" and "return"
- For variables/classes: omit "params" field entirely

5. TYPE INFERENCE RULES:
- Use specific TypeScript types: string, number, boolean, array, function, void, null, undefined
- For object types, prefer interface/class names if they exist in the code
- For arrays, use "type[]" notation
- For class instances, use the class name as the type
- Analyze return statements for accurate return types
- Use parameter usage patterns to infer parameter types
- Provide diverse alternatives (e.g., specific type, union types, general types like 'any')

6. Analyze initialization values and function bodies for accurate type inference:
- String literals → ["string", "any", "unknown", "string | null", "string | undefined"]
- Number literals → ["number", "any", "unknown", "number | null", "number | undefined"]
- Boolean literals → ["boolean", "any", "unknown", "boolean | null", "boolean | undefined"]
- Array expressions → appropriate array type alternatives
- Object expressions → object type alternatives
- Class constructors → class name alternatives

NEVER return "undefined" as a type unless the value is explicitly undefined. Always include a "return" field as an array of 5 types in the types object.
Return only the JSON array, no markdown formatting or explanations.

A.3 RAG Inference

You are a static type inference assistant. Given a piece of JavaScript/TypeScript code and some context from a vector database, infer precise TypeScript-style types for the main code.
Analyze the following code snippet:
Here are the top ${topN} relevant code snippets from the database:
${contextSnippets}
And here is the source code:
${sourceCode}
For each identifier in the original code snippet, provide 5 possible type predictions ordered by confidence (most confident first).
Respond only with a JSON array using this exact schema for each identifier found:

```
{
  "entity": "function|
             variable|
             class|
             class-method",
  "name": "identifier_name",
  "location": {
    "line": 1,
    "column": 0
  },
  "types": {
    "params": { "paramName": "type" },
    "return": ["type1",
               "type2",
               "type3",
               "type4",
               "type5"]
  }
}
```

IMPORTANT EXTRACTION RULES:
1. Extract ALL identifiers from the **original code snippet** including:
- Top-level functions (entity: "function")
- Variables and constants (entity: "variable")
- Class declarations (entity: "class")
- Class methods (entity: "class-method", name format: "ClassName.methodName")
- Arrow functions assigned to variables (entity: "function")

2. For location field:
- Estimate line numbers by counting lines in the source code
- Use column 0 if exact position is unknown
- ALWAYS include location object with line and column numbers

3. For types object:
- ALWAYS include "return" field as an array of 5 type predictions
- Order the return types by confidence (most confident first)
- For functions and class-methods: include "params" object (can be empty)
- For variables and classes: omit "params" field entirely
- Example for function: "types": "params": "a": "number", "b": "number", "return": ["number", "any", "unknown", "void", "never"]
- Example for variable: "types": "return": ["string", "any", "unknown", "string | null", "string | undefined"]
- Example for class: "types": "return": ["ClassName", "any",

"object", "unknown", "Object"]

4. TYPE INFERENCE RULES:
- Use specific TypeScript types: string, number, boolean, array, function, null, undefined, void
- For object types, prefer interface/class names if defined in the code
- For arrays, use "type[]" notation
- For class instances, use the class name as the type
- For class methods, use entity "class-method" and format name as "ClassName.methodName"
- Provide diverse alternatives (e.g., specific type, union types, general types like 'any')

5. REQUIRED JSON STRUCTURE:
- Every item MUST have: entity, name, location, types
- location MUST have: line (number), column (number)
- types MUST have: return (array of 5 strings)
- types MAY have: params (object) - only for functions and class-methods

Return only the JSON array, no markdown formatting or explanations.

B Statical Analysis

This appendix contains the results of the statistical tests used to analyse the results of the experiments and answer RQ1 and RQ1.

B.1 RQ1 Statistical Results

Comparing **3 approaches** (SF, AST, RAG) across **8 models × 18 files = 144 blocks**. Each (model, file) pair is treated as a single dataset/block. The setup is summarized in Table 6.

Table 6: RQ1 statistical test setup

Item	Value
Datasets (blocks)	144
Algorithms	Source File, AST, RAG
Global test	Friedman
Concordance	Kendall's W
Post-hoc	Nemenyi
Visualisation	Critical Distance (CD) diagram

Friedman Test. Non-parametric repeated-measures test: are there significant differences among SF, AST, RAG across the 144 blocks?

The test yields **chi2 = 97.4831** and **p = 0.000000** → Significant at $\alpha = 0.05$: **at least one approach differs**.

Kendall's W Test. Degree of agreement among the 18 files in how they rank the 3 approaches. W = 0 means no agreement; W = 1 means perfect agreement.

The test yields **W = 0.3385** (n=144, k=3) → **moderate concordance** among blocks

Nemenyi Post-hoc Test. Pairwise comparison of all approaches after a significant Friedman result.

The test yields the p-value matrix in Table 7.

Table 7: p-value matrix resulting from the Nemenyi Post-hoc test on the results for RQ1.

	Source File	AST	RAG
Source File	1.000000e+00	0.02007	8.642088e-10
AST	2.007013e-02	1.00000	0.000000e+00
RAG	8.642088e-10	0.00000	1.000000e+00

Plots. The plots of the results, including the CD diagram, can be seen in Figures 6, 7 and 8.

B.2 RQ3 Statistical Results

Comparing **4 approaches** (Source File, AST, RAG, Proportional Sampling) across **18 files** for a single model (Qwen 3 Coder 30B). The setup is summarized in Table 8.

Friedman Test. Non-parametric repeated-measures test: are there significant differences among SF, AST, RAG and Proportional Sampling across the 18 blocks?

The test yields **chi2 = 9.4310** and **p = 0.024076** → Significant at $\alpha = 0.05$: **at least one approach differs**.

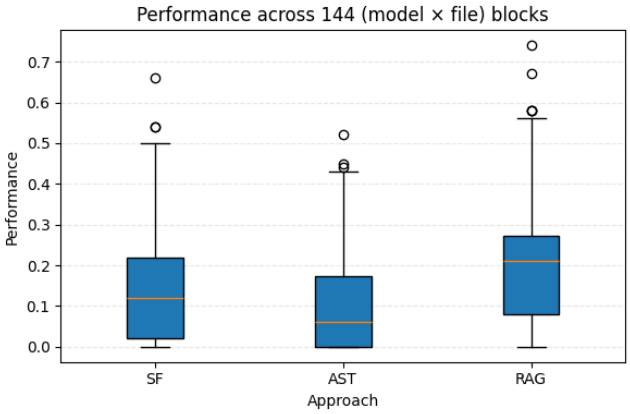

Figure 6: Boxplot of the three approaches across all 144 blocks.

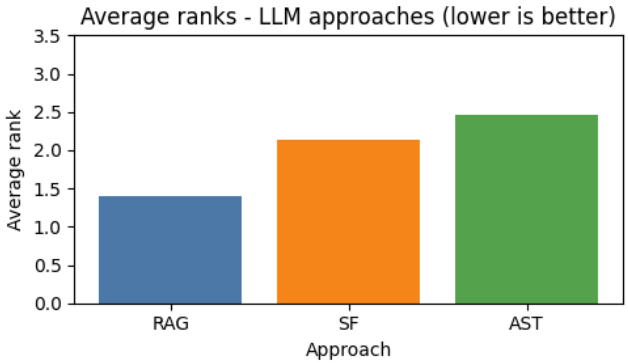

Figure 7: Average ranks for RQ1 results (Demšar-style; rank 1 = best, higher value → lower rank)

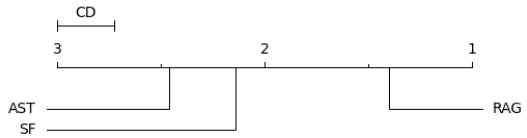

Figure 8: Critical Distance diagram for RQ1 results.

Table 8: RQ3 statistical test setup

Item	Value
Datasets (blocks)	18
Algorithms	Source File, AST, RAG, Proportional Sampling
Global tests	Friedman
Concordance	Kendall's W
Post-hoc	Nemenyi
Visualisation	Critical Distance (CD) diagram

Kendall's W Test. Degree of agreement among the 18 files in how they rank the 4 approaches. W = 0 means no agreement; W = 1 means perfect agreement.

The test yields **W = 0.1746** (n=18, k=4) → **weak concordance** among files.

Nemenyi Post-hoc Test. Pairwise comparison of all approaches after a significant Friedman result.

The test yields the p-value matrix in Table 9.

Table 9: p-value matrix resulting from the Nemenyi Post-hoc test on the results for RQ3.

	SF	AST	RAG	Prop. Sampling
SF	1.000000	0.835794	0.164467	0.999904
AST	0.835794	1.000000	0.019255	0.802874
RAG	0.164467	0.019255	1.000000	0.187522
Prop. Sampling	0.999904	0.802874	0.187522	1.000000

Plots. The plots of the results, including the CD diagram, can be seen in Figures 9, 10 and 11.

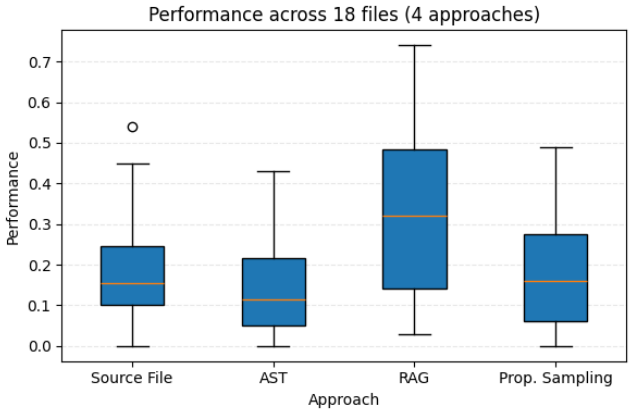

Figure 9: Boxplot of the four approaches across all 18 blocks.

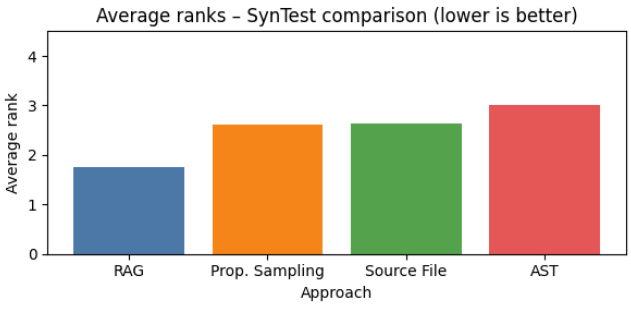

Figure 10: Average ranks for RQ3 results (Demšar-style; rank 1 = best, higher value → lower rank)

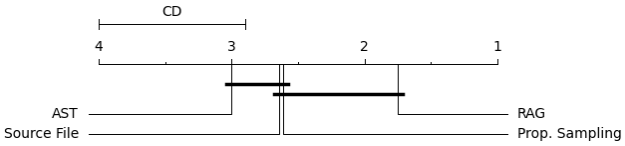

Figure 11: Critical Distance diagram for RQ3 results.