Proactive AI in IDEs

A Design Exploration and Evaluation of the Impact on Developer Experience in JetBrains Fleet

Nadine Kuo

Delft University of Technology

TU Delft

Proactive AI in IDEs

A Design Exploration and Evaluation of the Impact on Developer Experience in JetBrains Fleet

by

Nadine Kuo

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday July 18, 2025 at 1:30 PM

Student number:	5204895
Project Duration:	December, 2024 - July, 2025
Thesis committee:	Assistant Prof. dr. U. Gadiraju, TU Delft, chair
	Assistant Prof. dr. M. Izadi, TU Delft, supervisor
	Assistant Prof. dr. M. Gürel, TU Delft

Cover:	Canadarm 2 Robotic Arm Grapples SpaceX Dragon by NASA under CC BY-NC 2.0 (Modified)
Style:	TU Delft Report Style, with modifications by Daan Zwaneveld

TU Delft

Abstract

Recent advances in LLMs have transformed AI coding assistants from simple autocompletion tools into conversational partners that support a wide range of development tasks through natural language interaction. While this shift promotes closer human-AI collaboration, this also places a burden on developers to craft precise prompts and integrate sufficient context to accomplish their task. To address this, there is growing interest in proactive AI systems that can anticipate developer intent and surface timely, contextually relevant suggestions — potentially revealing insights developers might not have considered. However, existing research on proactive coding assistance has largely been confined to controlled, experimental settings, leaving open questions about its effectiveness and integration in real-world development environments.

In this work, we design and implement a proactive AI assistant within JetBrains Fleet, a polyglot enterprise IDE. Our system signals AI activity via in-editor cues and delivers context-aware code improvement suggestions via a chat interface. We adopt heuristics based on IDE activity to determine when it is timely to intervene, during various stages across the development workflow: formulating needs, idea execution and committing changes. Lastly, we conducted a five-day in-the-wild study with 18 professional developers to evaluate its impact on user experience and engagement patterns.

Our findings demonstrate that timing is a critical aspect influencing perceived value of proactive AI assistance. Suggestions delivered at natural workflow boundaries (e.g. post-commit) achieved the highest engagement, while mid-task interventions were often dismissed. The consistent engagement trends (and voluntary continued use) indicate strong potential for integrating proactive AI into everyday software engineering workflows. Our study not only underscores the feasibility and practical value of proactive assistance in enterprise development environments, but also provides actionable design insights for future tools. While challenges remain — such as improving suggestion relevance and supporting diverse user preferences — these highlight important directions for advancing adaptive, intent-aware proactive systems that can further enhance developer workflows.

Contents

Nomenclature

Abbreviations

Abbreviation	Definition
LLM	Large Language Model
AI	Artificial Intelligence
UX	User Experience
HAX	Human AI Experience
IDE	Integrated Development Environment
SUS	System Usability Scale

1 Introduction

Software development has undergone a major shift in recent years as LLM-powered programming tools have emerged and revolutionized traditional coding workflows. These AI assistants have evolved from simple code completion tools to sophisticated collaborative partners capable of supporting a wide range of tasks, including explaining code, suggesting refactorings, or assisting with debugging workflows — all through natural language interaction. This transition marks a fundamental change in how developers interact with their development environments, with human-AI experience (HAX) playing an increasingly critical role. Yet, even as these tools reshape coding workflows, the current generation of LLM-based tools still faces important limitations in how they support developers in practice [45, 8, 36].

Most existing LLM-based programming tools operate reactively, requiring developers to explicitly invoke assistance through prompting. This reactive paradigm imposes significant cognitive burdens, as developers must craft precise prompts, provide sufficient context, interpret responses, and manually integrate suggestions into their codebase. Recent studies indicate that these interaction costs can offset productivity gains, with developers spending considerable time formulating requests rather than solving core programming challenges [35, 51, 24].

In response to these limitations, there is growing interest — both commercially and academically — in proactive AI programming assistance [46]. Initial forms of proactive assistance appeared in auto-completion tools such as GitHub Copilot [14] and Visual Studio IntelliCode [31], offering code completions without explicit user invocation. More recently, AI-first code editors, including Cursor [11] and Windsurf [52], have introduced more sophisticated forms of proactive assistance, including next cursor position prediction and smart code edits based on the surrounding code context, comments, or detected errors. However, these primarily focus on localized suggestions tied to cursor position, missing opportunities for broader programming support.

On the other hand, chat-based AI assistants can consider a wider project scope and higher-level project goals. Moreover, their conversational nature enables more seamless human-AI collaborative workflows that extend beyond mere code generation, promoting deeper code understanding through contextual explanations and reasoning. Many now incorporate agentic capabilities, autonomously handling complex tasks directly within code editors (Cursor Chat [11], Windsurf Cascade [52], JetBrains Junie [22]) or via sandbox environments on the web (OpenAI Codex [39], Devin [10]). Despite these advances, the burden of help-seeking and intent articulation still falls on users. [1]

Prior studies have explored proactive coding assistants that periodically surface suggestions via chat interfaces, based on code context and user activity. These system-initiated suggestions have been shown to enhance developer productivity by alleviating intent specification costs and surfacing valuable ideas developers might not have considered [6, 55]. Further work extended proactive communication channels beyond chat, through support for direct in-editor code edits, along with local chat threads and visual indicators such as thought bubbles [43]. These richer representations were found to enhance

[1] Thus in this work, we define *proactivity* as the system initiating assistance by anticipating user needs, in contrast to agentic systems that spin off tasks autonomously only after user-specified requests.

users' awareness of AI presence and thus reduce disruption. Nonetheless, a major limitation of prior work is their evaluation in experimental coding environments, with predefined tasks carried out in single-session Python-only contexts — or lacking user studies altogether [55]. This restricts generalizability to real-world IDEs, where integrating proactivity introduces challenges including code maintainability and workflow disruption possibly due to concurrent AI features.

This work addresses these gaps by designing, implementing and evaluating a proactive chat assistant in JetBrains' Fleet IDE [2] with focus on improving code quality. Specifically, our research aims to:

Research Objective 1

Build an understanding of *when* to show proactive suggestions to users, to minimize disruption to developer workflows

Research Objective 2

Build an understanding of *how* to present proactive suggestions in the IDE, to ensure value-alignment, transparency and maintain user control

Research Objective 3

Evaluate our prototype with end users to understand the impact on user experience and interaction patterns across different development workflow stages

Addressing these objectives, we started out by outlining key design considerations shaped by the practical constraints of software development environments. Following our design goals, our prototype was iteratively refined based on pilot studies with internal, professional developers. The final prototype delivers timely assistance across key stages of the development workflow, with assistance triggered on in-IDE user events based on heuristics grounded in human-AI interaction literature [46, 35, 2]. The assistant ensures suggestion relevance by drawing context from IDE artifacts including workspace documents, IDE-detected problems, git diffs and chat history — with tool access to enrich context dynamically as needed. To enhance transparency, we signal AI activity via in-editor cues and visualize the assistant's working context via an in-chat context panel. Finally, we facilitate user control and ownership by providing explicit confirmation mechanisms when the AI intervenes, as well as options to apply suggested patches on demand into their workspace when deemed useful.

Through an in-the-wild study with 18 professional developers over a five-day period, we found that proactive AI assistance generally led to enhanced workflows — by surfacing useful insights developers might not have considered and reducing the cognitive load of expressing their intent. Analysis of 229 AI interventions revealed that timing plays a central role in shaping user receptivity. Suggestions delivered at workflow boundary points (e.g. after commits) consistently achieved highest engagement, whereas mid-task prompts were often perceived as disruptive. Importantly, we observed continued use beyond the formal study period, indicating strong potential integration into developers' natural workflows. Nonetheless, we also highlight challenges that should be further explored: enhanced intent modeling, context integration and adaptivity to personal preferences. All in all, our design exploration and empirical findings offer practical guidance for integrating proactive AI support into real-world development workflows.

This work presents the following key contributions:

- An exploration of the design space for integrating proactive AI support into developer workflows, with a particular focus on the timing and in-IDE representation of the AI
- A proactive AI assistant prototype integrated directly within an enterprise IDE, with in-editor cues to signal its presence and code improving suggestions presented within a chat interface
- A five-day in-the-wild user study with professional developers to evaluate the impact of proactive AI assistance on developer experience in real-world environments

[2] A polyglot, collaborative IDE that comes with built-in AI features including a chat assistant where we present proactive suggestions. https://www.jetbrains.com/fleet/.

2

Background & Related Work

2.1. AI Coding Tools: From Autocompletion to Vibe Coding

In recent years, software engineering workflows have been fundamentally reshaped by the rise of LLM-powered programming tools. While early code assistance tools such as IntelliSense [30] and Tabnine [48] focused on autocompletion, today's coding assistants can act as collaborative partners supporting a wide range of tasks from assisting in debugging workflows to coordinating multi-step refactorings across the codebase — all through natural language interaction with developers. This evolution represents a fundamental shift in how developers engage with their development environments, making human-AI experience (HAX) an increasingly critical factor in the design and implementation of AI-powered coding tools [45, 51, 24].

Despite rapid advances, current tools still face several important limitations in practice. Among these, one particularly relevant to our work is that most systems operate in reactive modes, requiring explicit developer invocation through prompting. This interaction style places substantial cognitive demands on developers, who must formulate precise intents, provide sufficient context, interpret and verify AI outputs, and manually integrate suggestions into their workspace. Prior studies [7, 36, 38] have shown that these interaction costs can erode the very productivity gains these tools promise, as developers often spend considerable time crafting requests instead of focusing on core programming tasks.

2.1.1. Growing Interest in Proactivity

Recognizing these limitations, there is growing commercial and academic interest in proactive AI to assist in coding tasks — systems that can anticipate developer needs and surface relevant suggestions without explicit prompting. Initial forms of proactive assistance emerged as inline autocompletion tools, where predictive models generate completions based on caret context or coding patterns [31, 48]. Early research explored when and how to best surface completions [34, 37], balancing intrusiveness, accuracy, and developer control. More recently, the rise of AI-centric IDEs such as Cursor [11] have enabled more sophisticated proactive behaviors. These systems incorporate next-edit and cursor predictions, as well as smart refactorings, preemptively assisting developers in navigating and editing code. Nonetheless, such features are largely tied to localized suggestions, with most proactive behaviors anchored around the immediate cursor context or adjacent code elements.

2.1.2. Chat Assistants and Agents

In contrast, chat-based AI assistants expand the scope of support by operating over larger programming contexts and engaging with higher-level project goals. Moreover, their conversational nature enables iterative, exploratory collaboration, allowing developers to ask for explanations, reason through design alternatives, or coordinate multi-step workflows. Modern assistants are typically integrated in IDEs such as Cursor [11], Windsurf [52] and JetBrains IDEs [22], or operate on the cloud via sandbox environments such as OpenAI's Codex [39] and Google's Jules [15]. Many now possess agentic capabilities allowing them autonomously handle complex tasks: implementing features, running code, iteratively refining

solutions, and verifying outputs, often operating across multiple files or tools. This has given rise to what some call a "vibe coding" paradigm [44], where developers express high-level intentions and the system dynamically executes, adapts, and improves the solution across multiple interaction cycles.

However, despite their autonomy, these agents are still predominantly reactive: they wait for explicit user commands or requests to trigger action. To clarify: we define *proactivity* as the system's ability to initiate assistance by anticipating user needs, while *autonomy* refers to the agent's capacity to independently spin-off tasks, typically after an initial user-specified request. This highlights a key opportunity for chat assistants to proactively surface useful suggestions or insights that the developer might not have explicitly requested, potentially increasing developer productivity and user experience.

In this work, we specifically explore proactivity in JetBrains Fleet [23] — a lightweight, polyglot IDE equipped with a suite of AI-powered features. Its built-in AI Chat provides context-aware support through integration with multiple IDE components — including the editor, terminal and git. It leverages LLMs via JetBrains AI Service [21], incorporating models from cloud providers such as Anthropic, Google, and OpenAI. At the time of building our prototype, this assistant had agentic capabilities to the extent of executing commands or editing files on behalf of the uses. [1] Nonetheless, similar to other existing chat assistants, it operates reactively: developers invoke it through explicit prompts or editor actions. Hence our research aims to introduce proactivity, enabling it to anticipate developer needs and offer timely suggestions without explicit user invocation.

2.2. Designing Proactive Assistants: Lessons and Challenges

Proactive AI systems, which take the initiative to offer help or suggestions without explicit user commands, have long been studied in human-computer interaction (HCI). Early systems such as Microsoft's Clippy are often cited as cautionary tales: their poor contextual awareness, poor timing, and lack of user control led to annoyance and rejection [3, 6]. Subsequent research refined the principles of effective proactivity by exploring user expectations, emphasizing the importance of relevance, explainability, and user-configurable control [29]. Moreover, Horvitz's mixed-initiative framework [17] argues that successful proactive systems blend human and machine contributions, adapting dynamically to context and shifting initiative as appropriate.

Recent work explores proactive systems in diverse domains: spanning healthcare [25, 41, 28], climate control [12], driving assistance [32] or the more well-known personal assistants such as Google Assistant or Siri. Across these studies, it becomes clear that explainability is essential: proactive suggestions are more likely to be accepted if users understand the reasoning behind them [33]. Moreover, timing is crucial: proactive help must respect moments of low cognitive load, avoiding interruptions during periods of focused attention [1].

2.2.1. Proactive Programming Assistance

Recent research has begun unpacking the design space for LLM-based proactive assistants in the programming domain, focusing specifically on when and how to surface suggestions — and what trade-offs arise between developer benefit and workflow disruption. Chen et al. [6] and Zhao et al. [55] explored periodic system-initiated suggestions within an in-editor chat interface — finding that while such proactive inputs can boost efficiency, excessive or poorly timed interventions quickly overwhelm users and diminish productivity. Hence, they argue that proactive assistants should evaluate and time their suggestions carefully, aligning them with the developer's current programming context.

Another body of work explored proactive AI representations beyond chat interfaces, leveraging carets, cursors and thought bubbles to signify the AI's thinking and edit traces [43]. Pu et al. confirmed that proactive AI assistance improved developer efficiency, but also raised concerns about workflow disruptions, code comprehension, and long-term maintainability of code. To address these risks, they emphasize the importance of explainability, giving developers clear justifications for AI interventions, and ensuring user control over when and how to apply suggestions. Furthermore, they recommend the use of visible presence indicators within the editor to signal the AI's activity — enhancing transparency and thus mitigating user confusion and workflow disruption.

[1] At the time, Fleet's AI Chat did not possess long-running agentic capabilities, sophisticated planning abilities nor long-term memory yet.

A major limitation of prior prototypes is the fact that these were integrated in experimental code editors, with evaluation limited to predefined tasks and Python-only contexts — or lacking user studies altogether [55]. This restricts generalizability of their findings to real-world IDEs, where integrating proactivity introduces challenges related to workflow disruption and code maintainability. Addressing this gap, our work integrates proactive AI assistance directly into an enterprise IDE. Moreover, we evaluate our prototype by conducting an in-the-wild user study with professional developers, spanning five days formally, to assess the impact on practical development workflows. Based on the trade-offs and lessons highlighted in prior works, we present a set of design considerations thereby taking into account constrains imposed by real-world software development settings (in Section 3).

2.3. Leveraging LLMs for Code Quality Enhancement

Earlier research on AI-powered programming assistance has shown that developers tend to favor actionable assistance that boosts their productivity and improves their overall code quality [6, 49, 5]. Building on these findings, our work focuses on AI assistance specifically aimed at code quality improvements. In fact, recent advances show that LLMs are increasingly capable of improving code quality across multiple fronts, including refactoring, bug repair, code review, performance optimization, and design smell detection. In refactoring, tools such as EM-Assist [42] integrate LLMs with static analysis to recommend transformations such as “extract method” with higher recall than traditional rule-based tools, though they often require human verification to guard against semantic errors or hallucinations. MANTRA pushes this further with multi-agent, RAG-enhanced frameworks that deliver automated, compilable refactorings, however also showing that current models still struggle with project-wide, cross-file changes [54].

For bug repair, earlier systems including CORE [50] and RepairAgent [4] combine static analysis and dynamic repair tools to fix complex bugs, achieving high success rates across multi-line patches. In the realm of automated code review [9], recent frameworks [47, 20] show that AI-generated comments can improve review depth but also introduce workflow trade-offs, sometimes increasing pull request closure times or injecting irrelevant feedback. Similarly, LLM-based optimization systems [13] and smell detectors [53] push beyond syntactic improvements to tackle performance and architectural issues, though they typically require iterative, multi-pass refinement [40].

Aligning with the hybrid strategies explored in prior work, our approach augments LLM capabilities with contextual signals directly from the IDE. We combine detected code inspection [2] issues (surfaced via static analysis, or any of Fleet’s code-processing engines [3] and language servers) and “Quick Fixes“ with real-time code context, in-editor AI interactions and AI chat history. Differently from above works, our focus is not on building specialized tools for dedicated code quality tasks, but rather on integrating proactive assistance into real-world development workflows — spanning insights on best practices, robustness, modularity, performance optimization etc. This addresses a critical limitation of earlier tools: namely, that most of them operate offline, limiting the generalizability of their findings to practical software development settings.

[2] https://www.jetbrains.com/help/idea/code-inspection.html
[3] In fact, Fleet relies on both IntelliJ IDEA and Rider code-processing engines.

3

Designing The Proactive AI Assistant

3.1. Design Goals

Our design of a proactive assistant for code improvement carefully balances AI utility with real-world development practicalities. These principles, summarized in Table 3.1, directly address key challenges identified in existing literature on proactive assistance and human-AI collaboration.

(DG1) Timely proactive AI assistance focuses on anticipating developer needs in order to preserve workflows and minimize disruption — especially alongside existing IDE features. Periodic suggestions as seen in earlier work lead to disruption [6, 55] hampering developer productivity. Although smart trigger models [34] represent a promising future direction, current limitations — including privacy regulations and overall lack of training data — make them impractical within our scope. To address the above, we establish local heuristic triggers based on IDE events that serve as proxies for assistance-seeking moments.

(DG2) Contextually relevant and applicable suggestions are crucial to ensure value-alignment with developers. Rather than general-purpose suggestions [6, 55], we focus specifically on code quality improvements such as optimizations, best practices and modularity that are directly applicable. For suggestion generation, we adopt a hybrid approach in the sense that we leverage LLMs and code inspection issues detected by the IDE. As real-time analysis of industrial-sized codebases is costly and infeasible scalability-wise, we scope LLM context to live IDE signals given the intervention type (e.g. code selections, commit diffs, inline prompts to AI). The AI assistant's underlying LLM also has access to tools to augment its context dynamically (e.g. getting the 10 most recently edited files) — which were native to Fleet's AI Chat so not specific to our study.

(DG 3) Explainability and transparency around AI assistance is crucial to build developer trust, particularly in industrial settings. Earlier work [43] highlighted how lack of understanding regarding AI suggestions can lead to long-term maintenance and extensibility issues. Moreover, we aim to avoid visual clutter from too many in-editor AI signals and potential distractions from varying interaction scopes [43]. To counter this, our design introduces minimal visual cues in-editor to signal AI activity with the global chat serving as central communication channel. Moreover, clear explanations are provided upon intervening or presenting in-chat suggestions. The AI chat assistant we built on top of also featured a context panel, enhancing transparency with regards to the assistant's working context.

(DG 4) Preserving user control is critical to ensure AI suggestions can be utilized efficiently while ensuring ownership [43, 6]. Our design enables developers to quickly dismiss or engage with the proactive AI whenever it intervenes in the editor. Moreover, we allow users to "Apply" generated patches on demand in proactively invoked chat sessions, as opposed to having them applied automatically [1]. Lastly, to avoid loss of user control and code understanding, suggested code snippets are scaffolded with expansion on demand.

[1] As can be seen in Cursor's Agent Mode, and was also the default setting at the time. Note that the application itself relied on Fleet's applier feature which was still under active development at the time we were building the prototype.

Table 3.1: Core design considerations adopted to explore the design and implementation space of proactive AI assistance, thereby listing constraints imposed by real-world IDE settings and interventions we take to address these

Real-world Constraints	Our Approach
Design Goal 1: Timely proactive AI interventions to preserve developer workflow	
Intrusiveness & Technical Feasibility • Periodic suggestions are costly and interrupt developer workflow • Insufficient data for trigger models, also adding inference overhead and costs, especially if cloud-based	• Adopt heuristic-based triggers given IDE activity, as proxies for assistance-seeking • Directly leverage LLMs to infer intent from IDE activity and developer context
Design Goal 2: Contextually relevant & applicable suggestions to provide value	
Intrusiveness • Raising too many non-actionable, merely informative, proactive suggestions can disrupt workflow	• Focus AI assistance on code quality improvements (optimizations, best practices etc.)
Costs & Scalability • Real-time analysis of industrial-sized codebases infeasible	• Adopt a hybrid approach, leveraging context-augmented LLMs with IDE-detected issues • Provide LLM with tool access for dynamic context augmentation
Design Goal 3: Explainability and transparency in AI presence to build developer trust	
Intrusiveness • Too many in-editor AI signals create visual clutter • Varying interaction scopes distract users due to context switching	• Minimal visual cues in-editor to signal AI activity, within close proximity of user's current working scope • Leverage global chat as central medium of communication with AI
Code Maintainability & Extensibility • Lack of code understanding may lead to developer distrust, but also long-term maintainability and extensibility issues	• Provide clear explanations for AI interventions as well as suggested patches • In-chat context panel for transparency on the assistant's working context
Design Goal 4: Preserving user control and ownership to support productivity	
Intrusiveness, Code Maintainability • Excessive AI autonomy leads to loss of user control, workflow disruption and long-term code maintainability issues due to overreliance on AI	• Explicit confirmation mechanisms upon AI interventions to allow users to efficiently engage or dismiss • Scaffolding of AI suggestions in the chat interface e.g. allowing users to expand code snippets on demand • Integrating generated suggestions via "Apply" on-demand when deemed relevant

3.2. Formative Pilot Study

In order to ensure feasibility of our prototype, we iteratively refined it based on pilot studies (see Figure 5.2) — two rounds were performed. In total, we recruited seven professional developers from within JetBrains to participate in a contextual inquiry [16] study. We ensured these participants were already active users of Fleet, and were regular users of in-IDE AI tools — avoiding confounding factors.

Each session was conducted remotely and recorded, with each session taking around one hour. During the session, participants used a development build version of the IDE with our early-stage prototype of the proactive assistant while thinking out loud. We gathered feedback by semi-structured interviews (see Appendix E) — for which all recordings were transcribed and analyzed to summarize recurring themes and iterate on initial user feedback. This phase served to identify and address key usability issues of the prototype and suggestions for improvement.

Below the main findings of the pilot studies:

Intrusiveness

Participants appreciated that the assistant inferred intent without requiring explicit prompts or manual context passing. However, unsolicited invocations of the AI chat — as was the case in our early-stage prototype — was generally seen as intrusive. Developers preferred lightweight, in-editor pop-ups to confirm chat invocation, noting that suggestion rejection often stemmed from task switching rather than disapproval.

Patch interaction

Receiving multiple small patches was preferred over a single large one. Participants valued the ability to preview changes inside the editor upon applying, but requested finer-grained diffs within patches in the chat itself. Suggestions felt more trustworthy when users could inspect changes suggested before applying them.

Alignment with context

Suggestions were occasionally perceived as generic or redundant. Users recommended incorporating prior edit history and avoiding repeated or already-attempted actions. Suggestions on detection of vague user prompts and post-commit were seen as helpful, but only when clearly grounded in context.

Discoverability & Explainability

As the proactive interventions were timed in a way for which users first had to invoke a certain context action, discoverability of the proactive suggestions was a recurring element. Participants proposed having a explicit options, either via context menu or floating toolbars, to request code improvement suggestions on demand. Also, terminology in the assistant's messages and UI components also required refinement to better match user expectations.

Based on this feedback, we introduced in-editor confirmation options upon intervening to enhance user control. Furthermore, we revised the type of interventions to align better with developers' workflows, and focused on presenting suggestions in separate, smaller patches that could be applied in tandem. Moreover, we refined system prompts for more effective inference of edit history and user intent. Lastly, we introduced an explicit context-menu AI action for manually requesting code improvements (on selection or entire file), alongside proactive interventions — the latter still being the core of this study.

4

Integrating Proactive AI Into The IDE

4.1. System Design Overview

Our proactive, LLM-powered assistant integrates directly into the IDE through Fleet's AI plugin, establishing bidirectional communication between IDE components, user interface, and the underlying agentic system. To determine when it is timely to intervene (**DG 1**, Section 4.2), the system has access to the user's work context, including documents, in-editor activities, terminal outputs, git operations, and code inspection issues. Rather than passively awaiting user requests, we use heuristics based on IDE activity to determine when the programmer may need assistance, thereby aiming to minimize workflow disruption. Upon deciding to intervene, the system collects relevant metadata from the current context to generate targeted LLM suggestions for code quality improvements (**DG 2**, Section 4.3). The user interface is where developers can verify, refine, and apply suggestions **(DG 3-4**, Section 4.4). The underlying assistant has agentic capabilities including tool access for fetching relevant IDE data and action execution, working memory in the form of conversation history, and internal planning through the underlying model's reasoning capabilities.

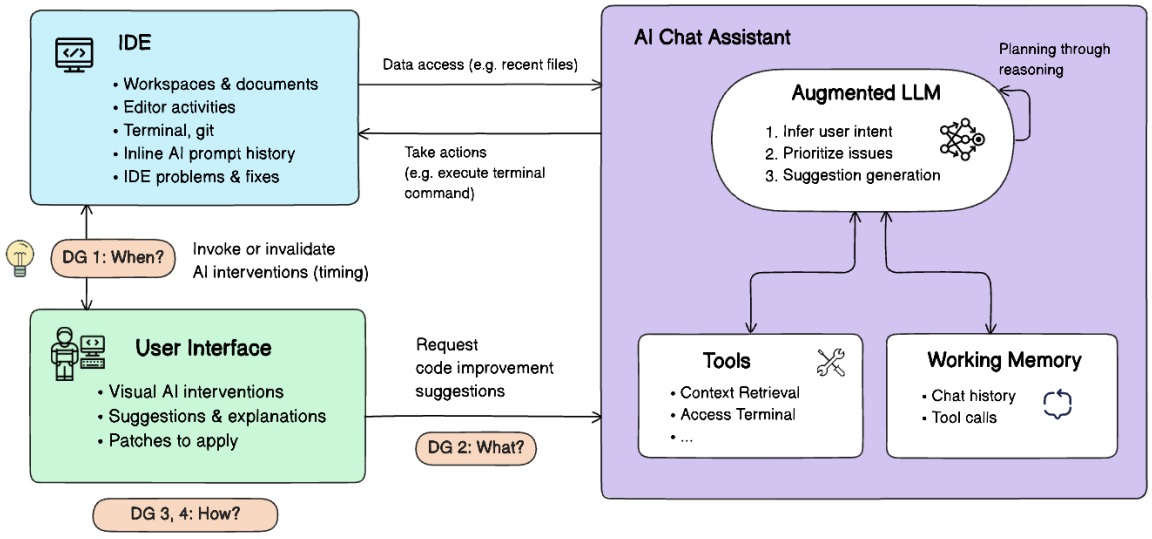


Figure 4.1: High-level system design depicting communication between IDE components, UI and Fleet's AI plugin serving the AI Chat Assistant. Note how the three main components were established already — our contribution is highlighted by the orange labels, spanning our design goals.

4.2. Timing of AI Interventions

In contrast to earlier works on proactive coding assistance raising periodic proactive suggestions [6, 43], we adopt an event-driven approach to minimize unnecessary disruption and resource consumption in real-world contexts. We leverage heuristics, partially grounded in existing literature, mapped to certain IDE events serving as proxies for developers seeking assistance. To minimize wasting of resources and minimize latency, we implement event filtering logic and debouncing mechanisms to discard rapid input noise from the user. Figure 4.2 depicts when the system proactively intervenes during various stages across the development workflow. We refer to these interventions as *Ambiguous Prompt*, *Declined Edit* and *Commit Changes* — with the context behind the first two being the developer iterating on code with inline AI, via Fleet's built in context action "Edit Code" (depicted in Figure 4.8).

Ambiguous Prompt Detected (Stage: Formulating Needs)

Heuristic: intervene when user is potentially looking for assistance through implicit signals [38, 43]

As developers may struggle to articulate precise instructions for an AI assistant, the system evaluates user prompts by adopting an LLM-as-judge approach with assessments against predefined criteria, including ambiguity (e.g. "make this better"), lack of context (e.g. requesting a fix without providing error details), or inconsistent references (e.g. using undefined variables). For this purpose, we specifically use GPT-4o given its support for structured outputs, allowing for a systematic assessment of user prompts. We avoid wasteful LLM requests by setting a minimum threshold of 15 characters for user prompts before we initiate the analysis, based on statistics given prompts fed to JetBrains' AI Assistant [1]. When a prompt is flagged as unclear, the system intervenes by suggesting more targeted approaches to improve their code quality in the chat. Note that we adapt the AI's intervention style according to the underlying intervention type, which is more exploratory in this case.

User Declines AI Edits (Stage: Executing Idea)

Heuristic: intervene when user is signalling unmet needs.

If a developer rejects an AI-generated edit (in this case when editing code via the inline AI action "Edit Code"), it signals that the provided solution may not fully meet their needs and further assistance can be helpful. Engaging with proactive interventions in this scenario leads to invocation of the AI chat assistant, where alternative suggestions are presented — possibly by explaining the reasoning behind the original suggestion or pointing out other issues detected that may be worth addressing. Similarly here, the AI's intervention style is more exploratory.

Committing Changes (Stage: Finish Writing Code)

Heuristic: intervene at task boundary i.e. natural checkpoints [19, 43]

As the user signifies that they have finished writing code, the assistant can provide immediate feedback on best practices and code quality in terms of criteria including modularity, efficiency, robustness etc. Upon successful commits, developers get an option to "Review Changes with AI" which then invokes the AI chat assistant. Here, its intervention style is more prescriptive compared to the above two scenarios.

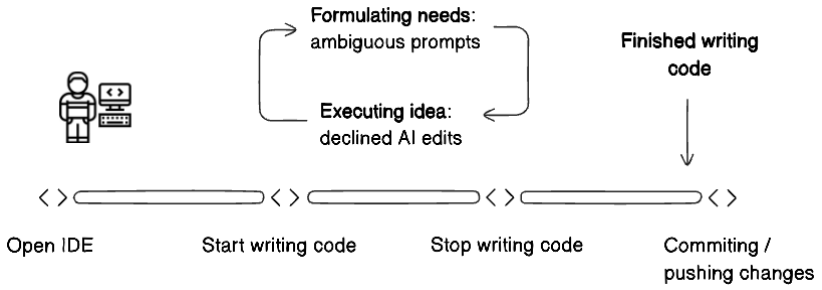


Figure 4.2: When the system proactively intervenes during various stages across the development workflow, given heuristics for assistance seeking.

[1] https://www.jetbrains.com/ai/

4.3. Suggestion Generation

To ensure the proactive assistant provides actionable suggestions to ensure value-alignment, we specifically target code quality improvements. Suggestions may cover refactoring opportunities, potential bug fixes, adherence to coding standards, performance optimizations, and ways to modularize the code further. By focusing on these high-impact areas, the assistant helps fostering developer productivity, while minimizing workflow disruption.

We leverage the power of LLMs augmented with IDE context — enhanced with tools and working memory (i.e. conversational history) — to provide contextually relevant assistance that adapts to the developer's needs and environment. Moreover, we augment the LLM's context with code inspection issues derived from either:

- Static analysis
- Fleet's code processing engines (IntelliJ or Rider) or language servers (LSP-based)

The user prompt contents vary depending on the user event that invoked the AI intervention. For instance, in the case of *Ambiguous Prompt* or *Declined Edit*, we attach the current user file with the selected code marked using `<begin of selection>` and `<end of selection>` tags — along with the prompt that the user fed to inline AI when using the "Edit Code" feature. On the other hand, in the case of reviewing *Committed Changes*, the LLM is provided with the associated git diffs.

By automatically incorporating the user's programming context into the assistant, we eliminate burden from the user having to gather necessary context himself. Instead, the user can shift his attention to actual problem-solving. Furthermore, by explicitly emphasizing human-AI collaboration in the system prompt, we steer the LLM to act as pair programmer. This is crucial to avoid loss of user control, potentially leading to code understandability (and possibly long-term maintainability) issues.

Figure 4.3 provides an example of how suggestions are generated via LLM prompts. The underlying action model is determined based on the user's model picker choice, which included (at the time of conducting our user study):

- Claude 3.5 / 3.7 / 4 Sonnet
- GPT-4o, GPT-4.1
- O3-mini, o4-mini

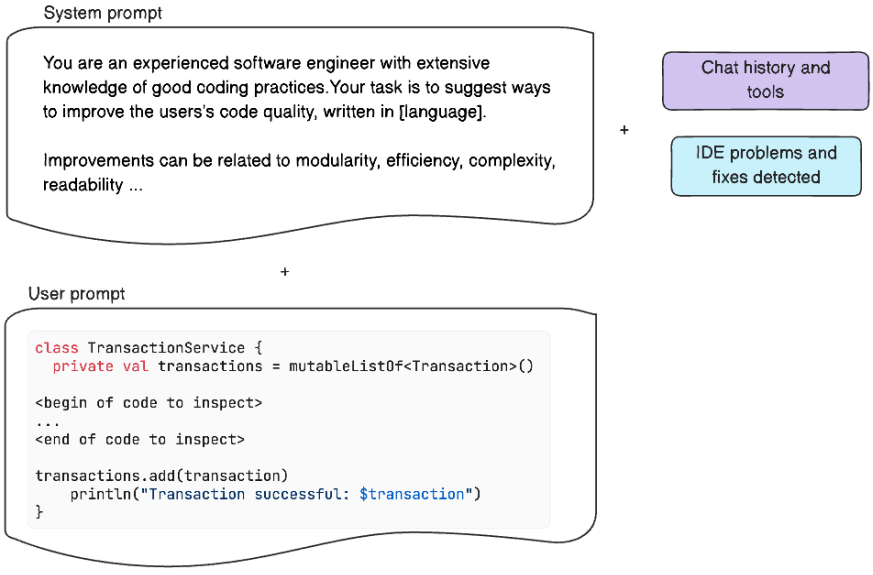


Figure 4.3: Illustrative overview of how suggestion prompts are curated given various IDE artifacts.

4.3.1. IDE Context Integration

The proactive assistant integrates multiple sources of information from the user's IDE environment as can be seen in Figure 4.1:

- User's workspace context: Recent changes, currently selected snippet, file opened etc.
- Editor activities: Navigations, code selections, interactions with AI etc.
- Terminal: Enabling the assistant to execute commands or intervene upon error messages
- Git: Any operation including metadata such as commits being pulled/pushed including descriptions, authors and timestamps
- In-editor user prompts: User requests made when iterating on code via the inline "Edit Code" feature — serving as indicators of user intent
- Code inspection issues: Issues detected in the user's codebase, identifying potential areas for improvement or fixing

Furthermore, the LLM has a short-term working memory (i.e. chat message history) containing previous interactions that help build incremental knowledge construction — aligning generated outputs with the user.

4.3.2. LLM Tools

To enhance the assistant's contextual awareness on-the-fly, and allow it to directly perform actions in the IDE (on request), we provide the underlying LLM with access to the following tools [2]:

- `Context-retrieval tool`: Retrieves additional context within the project workspace when necessary, such as recently changed files or specific classes
- `Web-search tool`: Gathers relevant web content given search query, providing both summarized insights and specific excerpts with associated URLs for traceability
- `Question-user tool`: Enables the assistant to ask clarifying questions, or confirmation questions with multiple answer variants
- `Search-in-embeddings tool`: Allows the assistant to find snippets of code that are semantically most relevant to the user's search query in chat

To avoid wasting system resources and ensure efficient operation, we impose limits on the frequency of tool invocations given a single request. These safeguards help maintain IDE responsiveness, prevent excessive background operations, and reduce unnecessary computational overhead.

[2] All of these tools were native to Fleet's AI Chat, thus not specifically introduced for this study.

4.4. In-IDE Representation of Proactive AI

The user interface plays a central role when it comes to explainability and transparency in the AI assistance to enhance developer trust, particularly crucial in the case of industrial settings. From earlier pilot studies, we found that signalling AI activity in-editor close to the user's current interaction scope is necessary to minimize distraction. This is in contrast to earlier works [6, 55] that directly displayed proactive suggestions in the chat interface without user confirmation. To this end, we introduce minimal visual cues inside the editor to signal AI activity.

To illustrate, Figure 4.4 displays an inline banner that appears when the system detects ambiguous user prompts and decides it is timely for the AI to intervene. Moreover, this provides the developer with an explicit option to either dismiss or engage with the intervention, which would invoke the AI chat panel to display suggestions for improving the user's code.

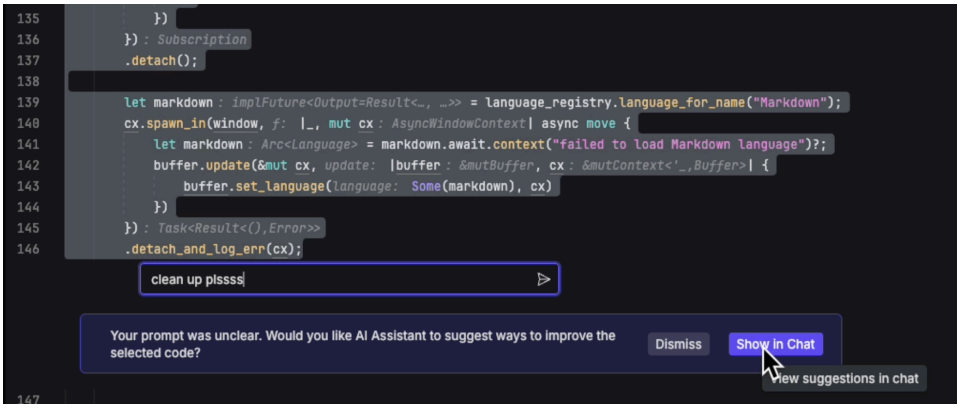


Figure 4.4: Display of visual cue in the editor to signify AI intervening when ambiguous user prompts are detected

Similarly, Figure 4.5 depicts how developer's receive a notification upon successful commits with the option to "Review Changes with AI". In this case, they can engage by clicking on that button or dismiss by simply closing the notification.

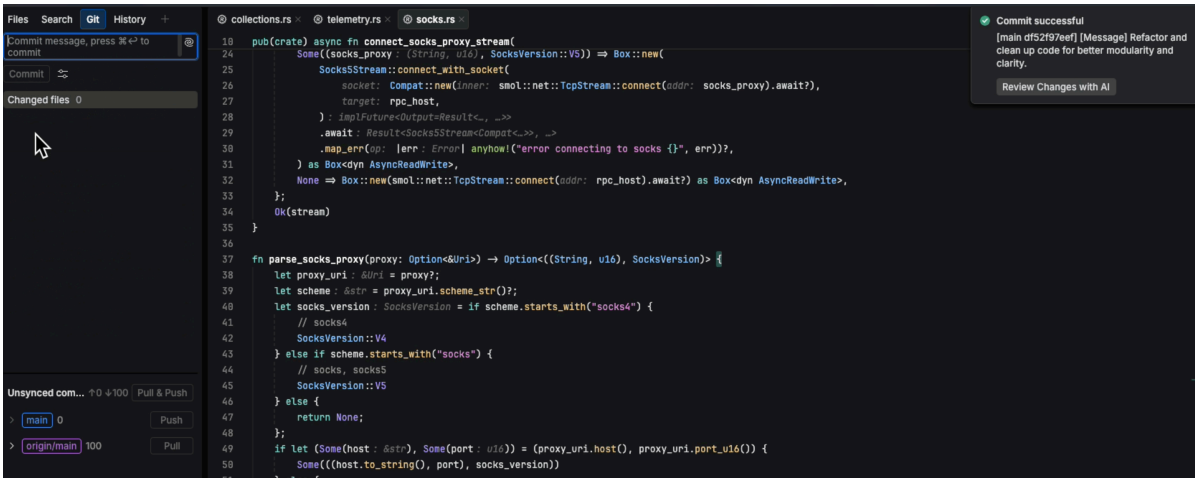


Figure 4.5: Post-commit notification in the editor to provide developers with the option to review their changes, emphasizing code quality and good practices.

By leveraging the IDE's AI Chat as central communication channel between developer and AI, we minimize workflow distraction due to varying scopes of communication leading to context switching.

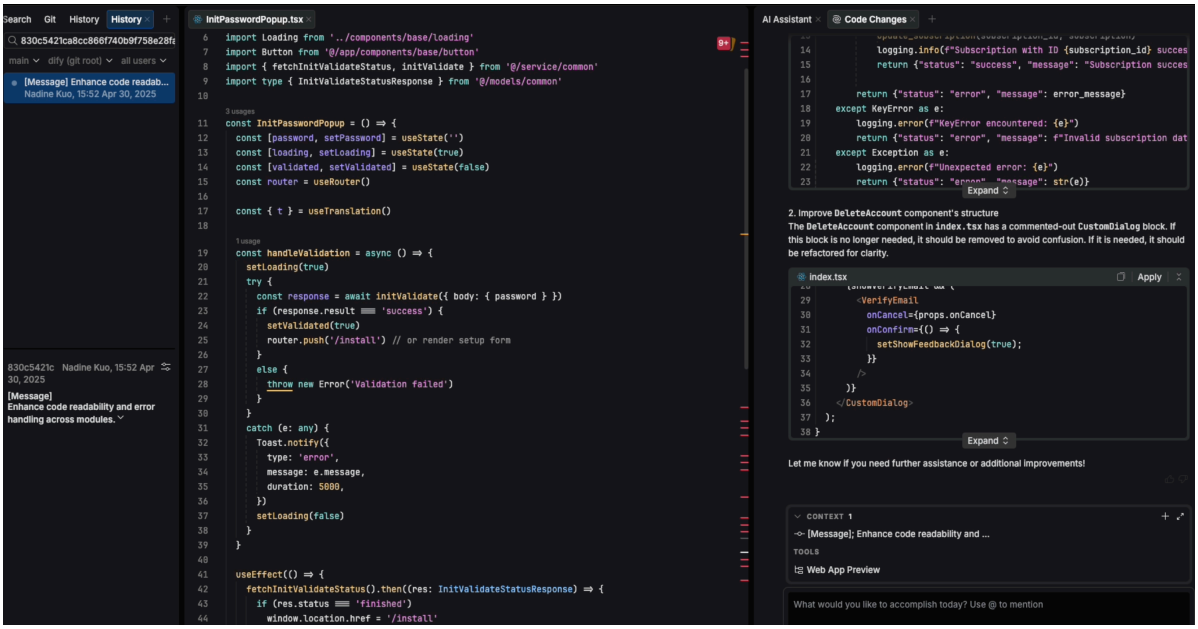


Figure 4.6: Chat interface showcasing display of code improvement suggestions that can be applied on user demand, which will subsequently allow for accepting or rejecting (Figure 4.7). The in-chat context panel allows for transparency on the assistant's working context — in this example listing the commit being reviewed, which can be clicked on to navigate to the commit directly in the git panel.

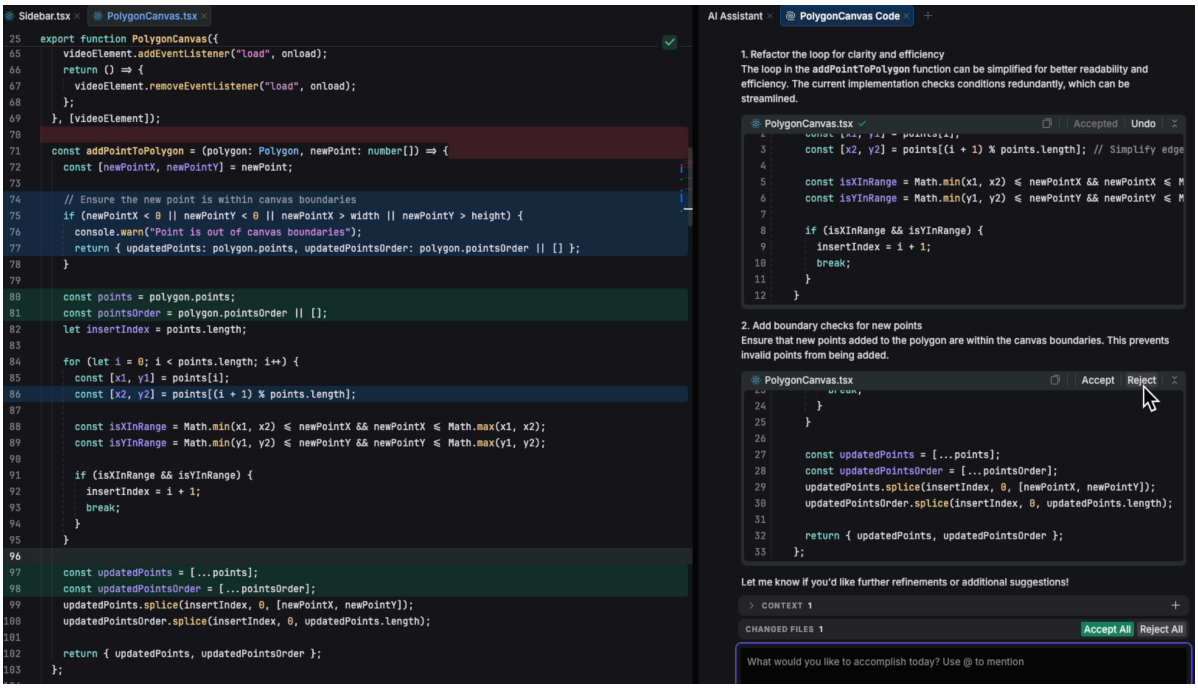


Figure 4.7: Chat interface showcasing display of code improvement suggestions, along with explanations and patches that can be automatically applied into the editor via git diff format — after which developers can opt to accept, reject and undo changes.

Each suggestion includes a concise title, explanation that articulates both the problem with the current code and rationale behind the proposed improvement, and an associated patch. This ensures that the assistant can help developers gain not just declarative knowledge (facts), but also procedural (how) and conditional knowledge (when, why) i.e. ensuring explainability. Any generated code snippet is scaffolded by default, and can be expanded on user demand. Not only does this minimize cognitive

overload, but also loss of user control and potential code understandability issues due to overreliance on AI.

By having an explicit option for users to integrate suggestions into their code, as depicted in Figure 4.6, we avoid unnecessary costs, latency and performance overhead — as each “Apply” incurs an additional LLM request in the background. After applying patches, the associated diff will be displayed in the editor, after which the developer can ultimately decide to “Accept” or “Reject” — either an individual snippet, all snippets or chunk-by-chunk inside the editor.

When it comes to minimizing latency and performance overhead, we also emphasize the critical importance of handling complex or long-running background tasks asynchronously, in our case leveraging Kotlin coroutines to ensure non-blocking execution. Specifcially, all operations that involve network requests (e.g. LLMs on the cloud), file system access, or large-scale computations are dispatched off the main thread to prevent UI freezes or unresponsiveness. On the other hand, any updates that affect the UI — including visual feedback, status indicators, or editor changes — are explicitly scheduled on the main thread to preserve optimal developer experience. This separation between background processing and UI updates is essential: even small delays in the IDE interface can significantly disrupt developer focus and workflow, leading to frustration and reduced productivity.

Lastly, we introduce an explicit AI Action in the context menu (on right-click in the editor) to manually request code improvements from the AI —- depicted in Figure 4.8 — based on earlier feedback from pilots. Moreover, this can allow future work better understand when proactive AI assistants should intervene.

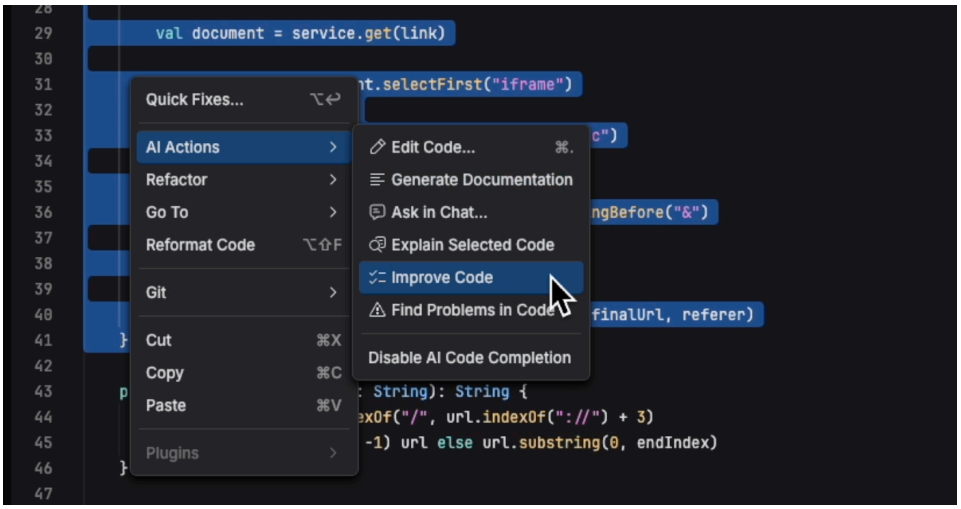


Figure 4.8: Additional manual AI Action via context menu to “Improve Code“ with AI, either for selected code or entire file (if user does not have anything selected).

5

Evaluation Through an In-The-Wild User Study

We conducted a five-day in-the-wild study to investigate the effects of proactive AI assistance on developer experience and productivity. The study employed a mixed-methods approach, combining quantitative telemetry data with qualitative survey responses to provide comprehensive insights into user interactions across their workflow. The study was conducted in line with our company's ethical standards, adhering to the values and guidelines outlined in the ICC/ESOMAR International Code [18]. The research questions we aim to address are as follows:

Research Question 1

How do user interactions with proactive AI vary across different workflow stages?

Research Question 2

What is the effect of in-IDE proactive AI assistance on user experience?

5.1. Participants

We recruited participants through JetBrains' research panel using a comprehensive screening (see Appendix B). The screening process yielded 139 total responses with a 20.9% completion rate, resulting in 29 qualified participants who met all selection criteria. Of these, 18 participants actively engaged with the proactive AI features during the study period between 7 May 2025 and 3 June 2025 (two rounds). Ultimately, 15 out of these 18 met the five-day participation protocol.

Participants were required to have regular experience with AI tools in day-to-day development work, with 60.9% using AI tools multiple times daily. All participants indicated familiarity with JetBrains products, with 49.2% using IntelliJ IDEA Ultimate, 34.8% using PyCharm Professional, and 29.5% already using Fleet IDE. Participants showed diverse AI tool usage, with 69.4% regularly using ChatGPT, 63.9% using GitHub Copilot, 38.9% using Anthropic Claude, and 36.1% using JetBrains AI Assistant. Another requirement to take part in the study was familiarity with in-IDE AI tools specifically, minimizing confounding variables as much as possible.

The participant pool consisted of various (possibly overlapping) job roles, with 89.7% identifying as Developer/Programmer/Software Engineer roles, 27.6% in Team Lead positions, and 24.1% as Architects. Professional experience was substantial, with 55.1% having over 11 years of coding experience and 24.1% having 6-10 years of experience. In terms of primary programming languages of the participants, Python (44.8%), Kotlin (37.9%), Java (20.7%), Go (20.7%), and TypeScript (20.7%) being most common.

Table 5.1: Participant Demographics

ID	Job Role	Languages	Experience	Frequency of AI Tool Usage
P1	Dev, DevOps	C#, Groovy, Py	6–10y	Multi/day
P2	Dev	Kt, TS	11–16y	1/week
P3	Dev, Researcher	Dart, Py, Shell	3–5y	Multi/day
P4	Dev	C#	16+y	Multi/day
P5	Dev, DevOps	Dart, Kt, Py	3–5y	Few/week
P6	Dev	Java, Kt, SQL	11–16y	Multi/day
P7	Dev, Analyst	Py, SQL, TS	6–10y	Multi/day
P8	Dev	Go, JS	11–16y	Multi/day
P9	Architect	Go, Kt, Py	16+y	Multi/day
P10	Dev	HTML, JS, Kt	1–2y	1/day
P11	Dev	C#, TS	3–5y	Multi/day
P12	Dev	Dart, Rust, Swift	6–10y	Multi/day
P13	Dev, ML Eng	Py	6–10y	Multi/day
P14	Dev, Architect	Go, PHP, TS	16+y	Multi/day
P15	Dev, Architect	Kt, Py	11–16y	Multi/day

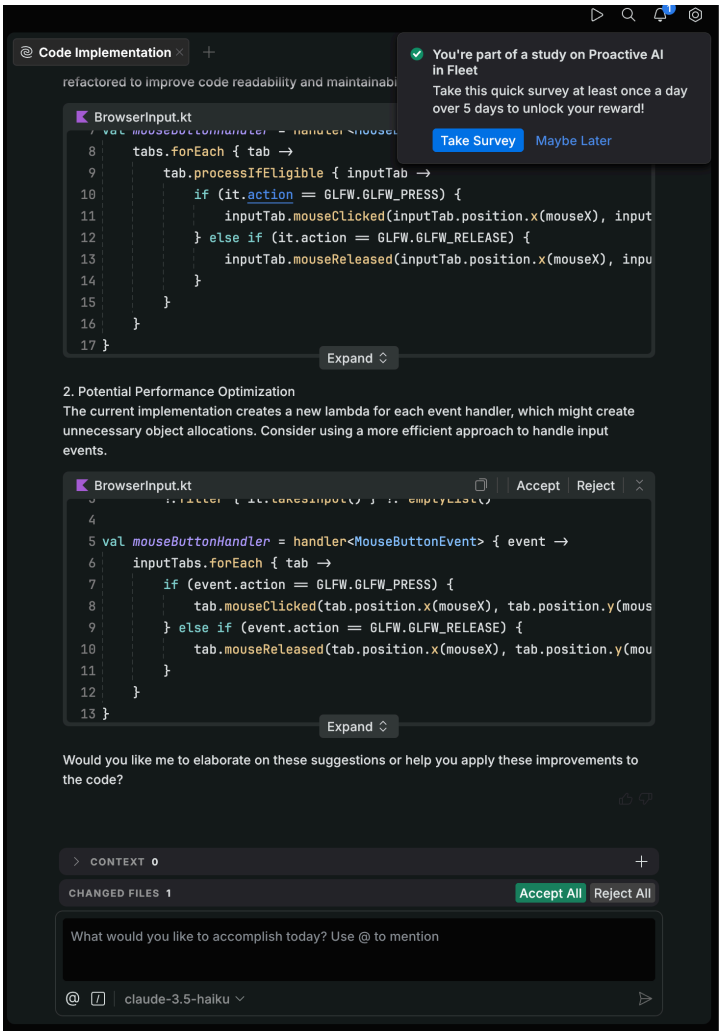


Figure 5.1: In-IDE notifications to nudge users to fill in daily surveys, five minutes after interaction with any proactive AI intervention.

5.2. Procedure & Data Collection

An overview of our data collection approach is provided in Figure 5.2. The study followed a structured timeline spanning four weeks in total, during which participants were required to actively use the system for at least five days (not necessarily consecutive) during their day-to-day work. Participants were compensated with their choice of either a USD 150 Amazon gift card or a one-year JetBrains All Products Pack subscription.

In the initial screener survey, participants were asked to accept research terms and conditions [1]. Participants eligible for the study received comprehensive setup instructions, which included troubleshooting guides and animated demonstrations of the proactive features. A key setup step involved explicitly enabling data sharing, allowing for anonymous usage statistics to be collected as telemetry data for analysis. Importantly, no personal data or sensitive information — such as file paths, the participant's source code, full-text inputs sent from the IDE to LLMs, or their corresponding outputs — were collected.

Throughout the study period, participants engaged in self-directed programming tasks reflective of their routine professional work. There were no restrictions on the programming languages, frameworks, or project types employed. See Appendix F for the user study manual that participants received.

To mitigate potential fraudulent participation, in-IDE notifications prompted users to complete short "daily surveys" five minutes after interacting with any of the proactive features, limited to once per IDE session [2] (see Figure 5.1). The survey (see Appendix C) asked participants about which specific aspect of the proactive AI assistant they used, the task context they applied it to, their perceptions of its behavior — and included an open-ended question for additional comments. We also used the survey to track participation, as anonymous telemetry data from the prototype could not be linked to names. Note how it was designed to take minimal effort i.e. users took approximately five minutes on average.

After using the feature for at least five days and completing five daily surveys, participants were asked to complete a post-study survey (see Appendix D). This survey included five-point Likert-scale items from the System Usability Scale (SUS) [26], as well as additional questions assessing perceived reliability, intent alignment, intrusiveness, and the impact of the feature on efficiency. Two open-ended questions invited participants to elaborate on how the suggestions affected their code quality and to provide feedback on how the prototype could be improved to better support their workflow. Filling out this post-study survey took 15 minutes on average.

All participants operated under consistent experimental conditions, using the same nightly build versions with the requisite feature toggles enabled. Furthermore, LLM temperature was explicitly set to zero to ensure deterministic and consistent outputs across interactions. While participants could select any LLM available via the AI Chat's model picker, they were encouraged to use GPT-4o, as it had the most robust support at the time.

[1] https://www.jetbrains.com/legal/docs/terms/general-research-terms/
[2] That is, the moment the user opens their workspace, to the moment of closing the IDE.

5.3. Data Analysis

Data collection employed both quantitative and qualitative sources:

- Telemetry data (N=18): Anonymous user interaction data was collected via Fleet's Feature Usage Statistics (FUS) system. This included fine-grained IDE events such as navigation patterns, UI element interactions (e.g. proactive interventions), git operations etc.
- Daily survey responses (N=15): Short daily surveys were part of the study to monitor active participation, and capture user experiences across the study period regarding helpfulness as well as open-ended feedback.
- Post-study survey responses (N=15): A post-study survey collected self-reported ratings on system usability, productivity and open-ended feedback regarding the overall user experience with the proactive AI.

To investigate how user interactions with the proactive assistant varied across different workflow stages (**RQ 1**), we analyzed telemetry data capturing interaction types and their rates of engagement, dismissal or ignoring — with a total of 229 interventions analyzed. We report descriptive statistics (e.g. proportions and usage patterns) to characterize engagement, serving as indicator for effectiveness of intervention strategies adopted. Appendix A provides some figures on general participation.

To assess the impact of the proactive assistant on user experience (**RQ 2**), we mainly analyzed responses from the post-study survey. This included both quantitative data from Likert-scale items and qualitative feedback from open-ended questions. Moreover, we analyze interpretation times of AI suggestions to evaluate impact on workflow efficiency.

By combining telemetry and survey data, we derive contextualized insights into the usability and perceived usefulness of the deployed prototype.

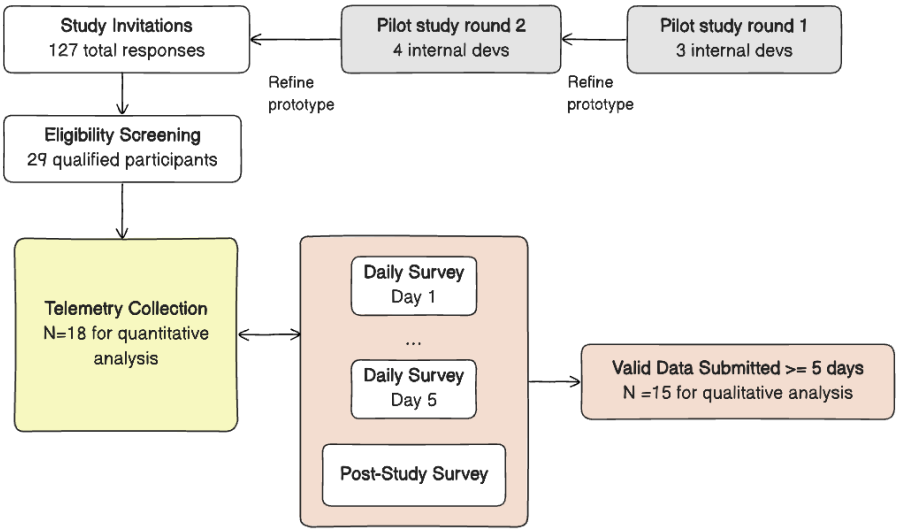


Figure 5.2: Overview of our data collection approach, spanning both qualitative and quantitative sources.

6

Results

We present findings from our mixed-methods, in-the-wild study evaluating a proactive AI assistant integrated into a real-world IDE. Our analysis combines in-IDE telemetry (N=18 machines with active usage declining to 8 machines by day five) and a post-study questionnaire with open-ended feedback (N=15).

6.1. RQ1: How Do Developers Interact with In-IDE Proactive AI Suggestions Across Workflow Stages?

We analyzed 229 interventions (Figure 6.1) across three workflow-triggered intervention types: (1) *Ambiguous Prompt*, (2) *Declined Edit*, and (3) *Commit Changes* over the study period of five days. Each intervention provided developers with options to either engage with the AI for code improvement suggestions, dismiss the notification, or ignore it entirely.

***Ambiguous Prompt* interventions** (N=35) yielded moderate engagement at 46%, with 23% dismissed and 31% ignored. The balanced distribution suggests varying underlying scenarios. In case of genuinely struggling to articulate their needs, the developer is likely in an exploratory state where they are more receptive to proactive suggestions that help refine their approach. However, others may know their desired outcome but provide insufficient detail in their prompt due to brevity preferences rather than uncertainty. Qualitative feedback revealed mixed reactions, with some developers expressing surprise at proactive interventions, observing that *"it seemed that 'simplify' was insufficient context for the AI to act upon"* [P1] or noting *"this behavior does not align with my experience with existing AI assistants"* [P3]. These suggest that developers feel disrupted by the AI and instead prefer having full control when they are carrying out set tasks at hand.

***Declined Edit* interventions** (N=39) achieved the lowest engagement rates at 31% and highest dismissal rates at 62%. Despite signaling unmet needs, these interventions appear to feel *"like advertisement"* [P3], indicating unnecessary intrusion when developers have already formulated specific task intentions. Also, context-switching between different AI interaction modalities appears to be causing cognitive overload, with participants expressing confusion about "when to use inline AI and chat" [P15]. In fact, telemetry data reveals frequent dismissals coinciding with active ongoing chat sessions.

***Commit Changes* interventions** (N=155) demonstrated the strongest reception, achieving 52% engagement rates. This timing is based on natural workflow boundary points when developers complete logical units of work and are *"still in the mindset of wanting to submit good code"* [P5]. This suggests that developers entering an evaluative mindset, having finished writing code, experience reduced cognitive switching costs and thus are more receptive to engage with proactive AI.

Temporal analysis over the study period revealed stable interaction patterns across all intervention types. For *Commit Changes* interventions, the consistent engagement rates throughout the study period indicate that the effectiveness of boundary-point interventions is not diminished by repeated expo-

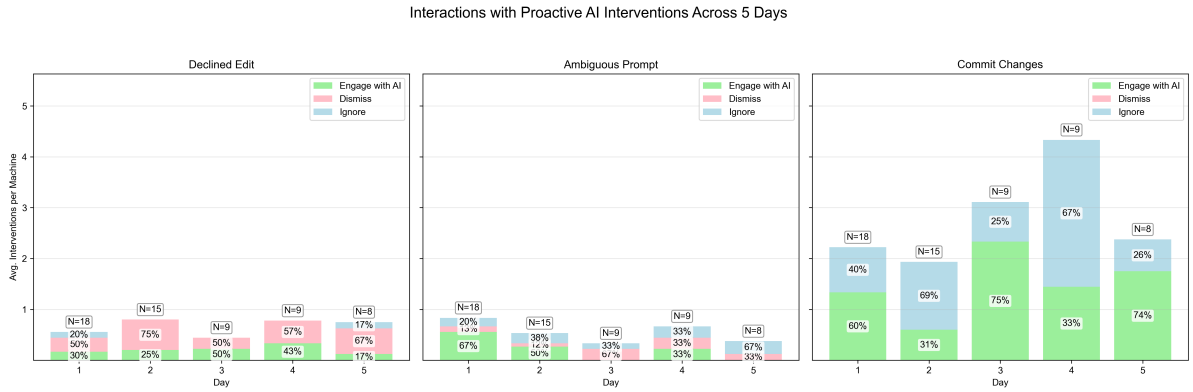


Figure 6.1: Rates of interaction with proactive AI interventions across various development stages: formulating needs (Ambiguous Prompt), executing idea (Declined Edit) and finish writing code (Commit Changes). Users can choose to "Invoke chat" or "Dismiss" the pop-up — or ignore it by navigating away or reformulating their prompt in the case of "Ambiguous Prompt". In total, we recorded 229 proactive AI interventions over the 5-day study period.

sure. Daily survey feedback confirmed that having AI review changes after commits remains effective and helpful, with users expressing desire to *"continue using it once the feature becomes available"* [P5]. In contrast, *Declined Edit* interventions maintained consistently low engagement throughout the study period, suggesting that poor reception was not due to an initial learning curve that users might overcome, but rather reflects a more fundamental disruption to their workflow. Moreover, 8 of 15 participants continued using the assistant beyond the required five days, indicating integration into regular workflows rather than novelty-driven usage.

Research Question 1

The observed timing patterns underscore that proactive AI's value depends critically on developers' cognitive context and thus readiness to receive assistance. The most effective interventions occur at natural workflow boundaries when developers are in the mindset of reviewing submitted code, while interventions during focused implementation tasks often feel disruptive and intrusive (despite signalling unmet needs).

6.2. RQ2: How Do Developers Experience the Interaction with In-IDE Proactive AI Suggestions?

Having established when proactive AI interventions are most effective, we now examine what underlying aspects influence how developers experience proactive AI assistance in their workflows. Here, we primarily walk over the qualitative data obtained from the post-study survey (see Figure 6.2).

Overall, participants had a positive perception of the system's usability, with 77.3% of participants expressing interest in frequent system use. Moreover, this was reflected in an above-average SUS score [1] (72.8 > 68). This score corresponds to a grade of B (indicating strong usability with room for improvement) and suggests that users are likely to be "passive" in NPS classification — holding generally favorable opinions but not necessarily promoting the system to others.

The SUS score is calculated using the following formula:

$$\text{SUS Score} = 2.5 \times \left(\sum_{k=1}^{5} (X_{pos,k} - 1) + \sum_{k=1}^{5} (5 - X_{neg,k}) \right) \tag{6.1}$$

where $X_{pos} = \{X_1, X_3, X_5, X_7, X_9\}$ represents the raw scores for positive statements (odd-numbered items) and $X_{neg} = \{X_2, X_4, X_6, X_8, X_{10}\}$ represents the raw scores for negative statements (even-numbered items). Each raw score X_i ranges from 1 to 5 based on the Likert scale responses. Likert

[1] https://measuringu.com/interpret-sus-score/

scale responses are mapped to numeric values: “Strongly disagree” = 1, “Disagree” = 2, “Neither agree nor disagree” = 3, “Agree” = 4, and “Strongly agree” = 5.

6.2.1. Developers Engaged with Proactive AI for Varying Purposes

From our qualitative data analysis, we identify several aspects that shape developers’ perceived value of proactive AI in terms of user experience and productivity — including alignment with developer intent and mental model. Participants appreciated when the system successfully understood their intent, as noted by P6: *“In most cases the suggestions were exactly what I wanted. Most of them I was able to accept as they fitted my task I wanted to complete.”* Moreover, from the post-study survey responses we observed that 63.6% of participants indicated it was easy to understand how proactive AI suggestions related to what they were trying to achieve.

Furthermore, we identified various underlying purposes for engaging with proactive AI. For exploratory purposes, proactive AI served as an effective code quality gate, with participants appreciating its role as a second pair of eyes for identifying overlooked issues or general enhancements. P3 noted: *“There were a few suggestions around safety ... an aspect that actually is really important in order to catch senseless errors when speed coding and did improve my code significantly.”* Similarly, P5 mentioned how *“[They] didn’t catch that loading the asset might through an error as [they were] following an implementation guide from the library docs.”* Participants also seemed to get an intuition for the AI’s strengths: *“the AI is helpful for identifying repetitive code and extracting it to improve modularity. It also handles typing well.”* [P13]

For accelerative purposes, while developers generally rejected proactive suggestions after declining inline edits, some participants appreciated that the system proactively escalated to the AI chat with relevant context to help them accomplish their task — particularly when the inline AI had failed to produce the desired results. This indicates that proactive AI can alleviate the burden of intent specification or switch between different AI modalities, effectively *“offering proactive code editing while [they were] writing code so that [they wouldn’t] have to open a chat window or go somewhere else to fill in what’s missing”* as P12 explained.

The study reveals that proactive AI assistance generates overall positive effects on perceived productivity, with the majority indicating that using the proactive suggestions helped them complete tasks more quickly (68.1%) and easily (59.1%). Comparing interpretation time of code snippets accepted or copied over, across reactive and proactively invoked chat sessions, shows that users take significantly less time to interpret proactive suggestions and incorporate them into their code, compared to manually requested ones (Wilcoxon signed-rank test: $W=109.00$, $p=0.0016 < 0.01$), as illustrated in Figure 6.3.

6.2.2. User Experience was Largely Influenced by the AI's Contextual Understanding

The perceived utility of proactive AI suggestions is also greatly affected by the AI’s contextual understanding of the developer’s workspace, impacting the quality of suggestions presented. However, only 31.8% believed that the suggestions generated in the AI chat were reliable, and 44.5% felt comfortable accepting the LLM suggestions. Interestingly, P3 noted: *“I did not feel that any suggestion was misleading ... so perhaps the coder’s perception and friction with the code they are writing is an important factor here.”*

Trust diminished when AI *“lacked understanding of current code patterns”* [P3] leading to irrelevant suggestions, or when higher-level intent behind design choices was not understood i.e. in some cases *“repetitiveness was necessary and unavoidable for the specific functionality intended to achieve”*. Developers working with specialized frameworks experienced frustration when AI *“failed to account for domain-specific syntax and patterns”* or suggested *“code refactoring that makes it unnecessarily complicated for a neural network”* [P13]. We note that LLM training data here plays a substantial role, as models may lack sufficient exposure to specialized domain patterns or recent framework versions, leading to suggestions that appear syntactically correct but are contextually inappropriate. As P3 observed: *“However, at least half of the times that this happened, other LLM chats (asked after this tool’s response had failed) would get it wrong in their first try as well.”* To enhance suggestion relevance, participants desired integration with broader codebase context *“possibly using nearby open tabs or recently edited files”* [P3]. Note that the underlying LLM serving the AI chat did have tools available to invoke the latter

dynamically, as passing this by default on every request is costly and not always necessary.

Furthermore, the application of patches [2] into the developer's codebase impacts user experience, regardless of suggestion quality. Overall, the majority (68.2%) thought that the various functions in the system were well integrated. However, unreliable patch application that breaks builds or requires additional manual editing creates workflow friction and disruption that outweighs potential benefits. Even valuable suggestions lose utility when delivered through unreliable system integration, as P4 experienced: *"The suggestions were sort of helpful but messed up the code file. I manually copied the suggestions - undid the changes, then pasted the suggestions into the correct place."*

6.2.3. Developers Have Varying Needs and Preferences

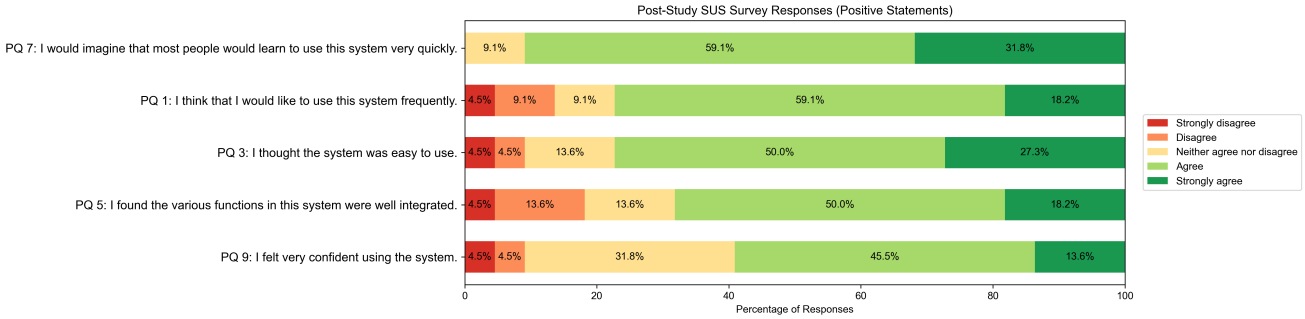
Design aspects that impact developer engagement with proactive AI also come with personal developer preferences, indicating a need for adaptivity. In terms of user control around interaction with AI, the majority indicated that most people would learn to use the system quickly (91%) and it was overall easy to use (77.3%), as shown in Figure 6.2. Most valued explicit confirmation mechanisms, appreciating that the system was *"even better than some other IDEs with integrations with the AI agent and the editor (e.g., offering custom UI in form of buttons when asking you for confirmation)"* [P12]. The transparency in AI presence also contributed to helping developers feel like they *"always knew what the suggestions were talking about"* [P3]. In terms of when proactive interventions are triggered, developers generally prefer having control, as P5 noted: *"I prefer to use hotkeys where possible ... or a command or something that can be used."* However, other participants preferred more AI autonomy, with P15 expressing: *"I would like to immediately move the context to the 'Chat' without even asking me, since I didn't accomplish what I tried to do - just move to the 'Chat' and give me the answer."*

Regarding the degree of proactiveness, some participants indicated that *"it would be good if the Review Commit option could either be triggered automatically on any commit"* [P5] or even *"extend that ability to any changes made in git"* [P10]. Whereas others found the frequency of interventions excessive, noting that suggestions were *"not always necessary... dismissing the dialog is annoying"* [P15]. For granularity of suggestions, some participants desired more granular assistance, requesting *"finegrained comments in code and documentation generation"* [P3] for detailed explanations. Similarly, when issues such as code repetition were detected, some participants wanted comprehensive fixes rather than partial demonstrations, noting frustration when AI chat *"only shows a small part of the file"* [P11]. Others preferred brevity, indicating that they *"would prefer for a suggested code to be returned ... no need for explanation text"* [P15]. Regarding types of suggestions, users indicated they wanted to *"choose the types of suggestions I want (e.g., performance, readability, security)"* [P4].

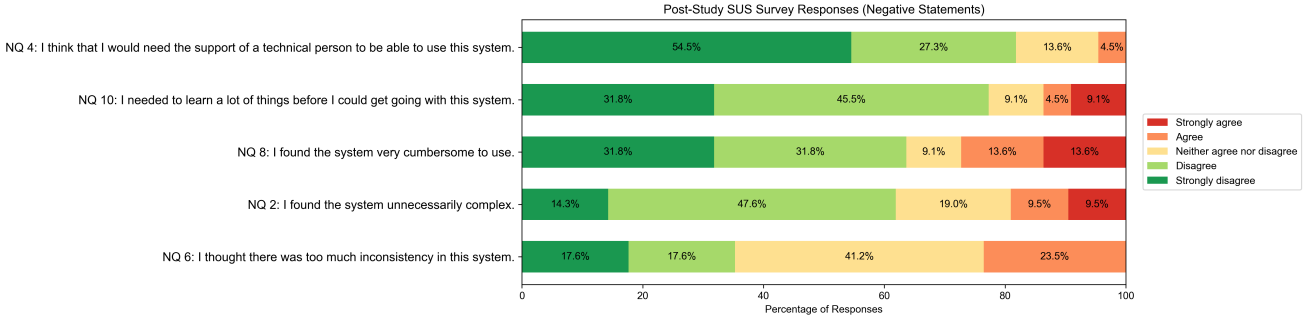
Research Question 2

Our findings reveal that developers engage with proactive AI for both exploratory and accelerative purposes, achieving higher perceived productivity and significantly faster code interpretation times compared to reactive AI use. However, developer experience is largely constrained by the AI's contextual understanding of their workspace, with trust diminishing when suggestions lack domain-specific knowledge or fail to integrate reliably into the codebase. The significant variation in developer preferences regarding proactiveness degree, suggestion granularity, and interaction control highlights the critical need for adaptive and customizable proactive AI systems that can accommodate diverse individual workflows and requirements.

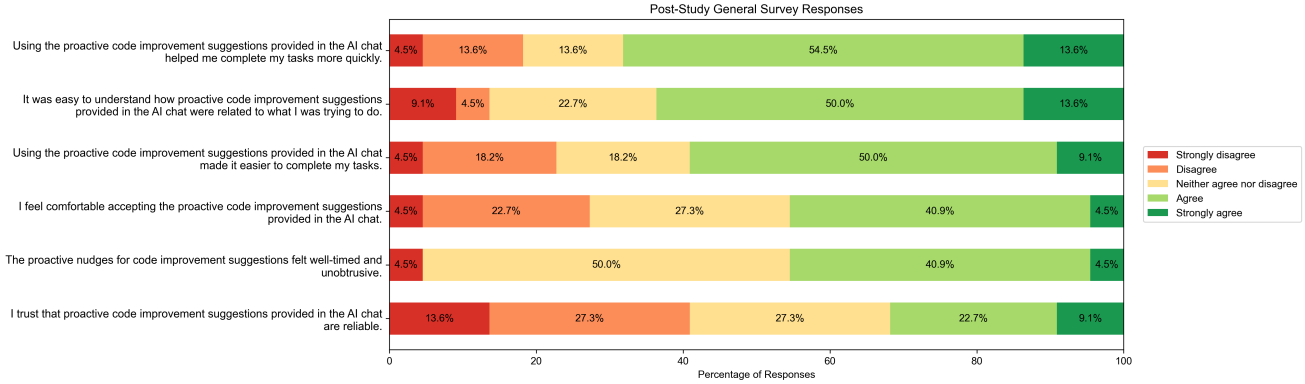
[2] Note that we relied on the built-in patch application system in Fleet, which was still under active development at the time of conducting our user study i.e. not fully stable.



(a) System Usability Scale (SUS) positive statement results



(b) System Usability Scale (SUS) negative statement results



(c) Additional post-study survey results

Figure 6.2: Responses from post-study surveys based on the 15 participants that participated consistently over the entire study period.

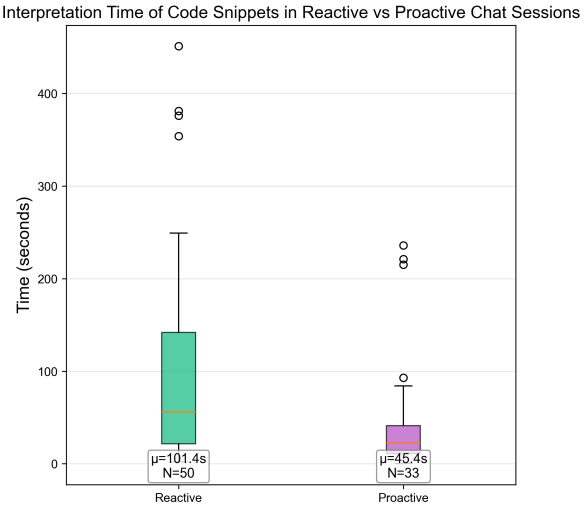


Figure 6.3: Comparison of interpretation time of in-chat code snippets across proactively invoked chat sessions and reactively invoked chat sessions (manual). Specifically, this is the time recorded between the LLM response was received, and the user either accepting or copying over the snippet into their code. To account for inactivity, we discard those response times exceeding 10 minutes.

7

Discussion

7.1. Technical Implementation Challenges & Implications

The integration of LLMs in real-world IDEs presents several technical challenges that must be addressed to ensure developers receive a seamless and uninterrupted experience.

Dealing with the nondeterministic nature of LLMs, especially in multi-model scenarios, emerged as a primary challenge. Specifically, sensitivity to prompt engineering made it difficult to ensure predictable model behavior including patch generation (e.g. in specific git diff formats). To navigate this, it could be beneficial to enforce structured outputs (e.g. JSON-based), integrate validation checks and have fallback mechanisms to maintain reliability. Moreover, it could be worth exploring frameworks for modularizing prompts to enhance reusability.

Furthermore, managing chat interaction history posed challenges. As one participant noted, *"I have noticed that if a particular chat session gets long, then the model would start glitching out"* [P7]. This issue fundamentally stems from LLM token limits, for which it is non-trivial to precisely anticipate when it will be reached — as the total number of tokens is the sum of prompt tokens and response tokens. These constraints also impose constraints on context integration, suggesting a trade-off between response accuracy and responsiveness. Note that in this work, we start a new chat session for each proactive intervention by default. In fact, earlier work [27] has shown that increased context lengths have also been linked to a greater chance of incorrect answers. To this end, it can be worth exploring incremental chat summarization techniques to reduce token usage while preserving essential context. Additionally, leveraging long-term memory mechanisms (e.g. vector databases) rather than relying on short-term memory limited to the current chat session can be beneficial. The latter can also enable persistent storage of user preferences and coding styles, allowing for more personalized interactions across sessions.

There are also challenges that come with building LLM-powered applications in general. For instance, testing and debugging is more complicated compared to traditional software engineering, due to the inherent nondeterministic nature of LLMs. We highlight the need for systematic evaluation frameworks and establishing proper traceability techniques for debugging, which could include logging intermediate LLM reasoning steps. Moreover, as seen in this work, alignment with user intent is crucial — beyond correctness (syntactic or semantic). However, formally specifying alignment with user goals is hard, and online evaluation is often costly. Future research should explore scalable, efficient methods for capturing and evaluating alignment to ensure user-centered model behavior. Lastly, to ensure smooth performance for end users at any time, we stress the importance of asynchronous handling of complex background tasks (e.g. requests to LLMs on the cloud) — with any UI changes visible to the users scheduled on main thread.

7.2. Design Implications From User Perspective

Here, we revisit our design goals based on our findings, and derive implications to navigate the future design of proactive AI coding assistance in IDEs.

(DG1) Timely proactive AI assistance to preserve workflow.

Our findings reveal that timing is a crucial aspect that impacts developer engagement with proactive AI in the IDE. In fact, workflow boundary interventions demonstrate the highest effectiveness, as evidenced by sustained engagement patterns, due to minimized cognitive switching costs and natural blending into developer workflows. On the other hand, mid-task interventions (e.g. after declined edits) were frequently dismissed or ignored.

Implication: Developers' workflow context and cognitive readiness must guide when interventions are triggered.

Recommendation: Future designs should ideally achor proactive assistance to clear task boundaries (e.g. commits, builds, test runs) where developers are cognitively ready to receive suggestions. For mid-task interventions, we highlight the need for more accurate intent modeling and possibly tuning down the AI's prominence in the IDE.

(DG2) Contextually relevant & applicable suggestions to provide value.

Developers valued suggestions that reflected their current task at hand, responding positively when interventions felt well-aligned (*"exactly what I wanted"* [P6]). Nonetheless, perceived value of suggestions dimished when they were perceived as irrelevant, due to unawareness of domain-specific patterns for instance — which can be attributed to both context integration and LLM training data. While our proactive assistant's underlying LLM was equipped with tools for dynamic context augmentation, these may not always have been operationalized effectively.

Implication: Suggestion quality is largely influenced by the AI's understanding of developer intent and their workspace, highlighting the importance of enhanced integration with workspace context.

Recommendation: To further improve value-alignment with developers, we recommend more effective heuristics for integrating external context sources, *"possibly using nearby open tabs or recently edited files"* as noted by [P3]. We also underscore the need for enhanced prompt engineering strategies, as well as more systematic tool instruction frameworks to optimize the way LLMs leverage provided context. Notably, contemporary advancements in emerging context integration protocols such as MCP [1] offer promising directions in personalized, dynamic context augmentation. Lastly, when ambiguity remains, it is essential that developers can be prompted proactively for clarification.

(DG3) Explainability and transparency to build developer trust.

From qualitative responses, we observed that overall transparency in AI presence within the IDE helped establish reliability. Developers appreciated concise rationales, previewable changes, and minimal in-editor cues that respected their workflow. To further enhance trustworthiness in AI assistance, earlier pilots indicated that it *"would be helpful if each suggestion came with a confidence level"* or *"clearer UI on what changes are suggested by the AI"*.

Implication: Proactive AI suggestions must be understandable at a glance, predictable in behavior, and verifiable before application.

Recommendation: The above underscores the need to support more informed developer decision-making by reinforcing transparency through clear diffs when presenting patches, and possibly indicators of suggestion confidence or source (e.g., "based on recent edits").

(DG4) User control to preserve ownership.

Control over when and how to engage with AI suggestions was a recurring theme throughout the entire development process of our prototype. Early pilot studies revealed frustration with unsolicited chat window openings, which interrupted developer flow and created unwanted context switches. In response, our final prototype featured clear, contextually-placed visual cues within the editor, which participants

[1] https://www.anthropic.com/news/model-context-protocol

valued as it allowed them to dismiss or engage with the proactive AI. However, preferences varied across individuals; some preferred more autonomy, while others favored greater manual control via hotkeys or context menu actions. Beyond interaction modalities, users expressed their desire to have control over the type and granularity of suggestions.

Implication: To accommodate diverse user workflows and preferences, adaptive human-AI interaction strategies are needed.

Recommendation: These findings point toward promising research directions in adaptive AI assistance, that can learn from both explicit user feedback and implicit behavioral signals. It is worth exploring how user configurability in intervention timing, frequency, and suggestion types could be combined with gradual personalization mechanisms. Such systems help maintain user control while evolving to better match individual developer needs and workflow styles over time.

7.3. Comparison To Prior Work

Here we compare our findings with earlier work on proactive coding assistance, highlighting both aligning findings and novel insights that emerge from real-world deployment settings.

Firstly, our work supports previous findings by Chen et al. [6] and Pu et al. [43] that proactive AI can enhance perceived workflow efficiency through just-in-time suggestions and reduced manual prompting. Both studies also emphasized the importance of timing, with Pu et al. highlighting task boundaries as especially effective moments for intervention — a pattern we similarly observed in our in-the-wild study through sustained engagement patterns. Additionally, Pu et al. identified challenges with false positives from heuristics based on implicit user signals. This became evident from our findings on *Declined Edit* interventions as well, underscoring the need for more accurate inference of developer intent and cognitive state.

Our findings further support Chen et al.'s characterization of proactive AI as a "second pair of eyes" that helps developers catch potential issues early. As P11 described: *"The review feature really helps to spot some repetitive items in the code... gives some really nice pointers that you can check, and rubber duck some potential issues with in your code base."* This reinforces the value of proactive AI as code quality gate, enhancing developer confidence.

Nonetheless, our work extends earlier findings in several important ways. Firstly, earlier experiments were confined to predefined tasks carried out during a single session in controlled environments and Python-only contexts. In contrast, we were able to observe in-the-wild developer interactions across a five-day study period, indicating stable engagement patterns and thus potential for proactive AI to be integrated in real-world development workflows.

As our participants worked on self-directed professional programming tasks, spanning a variety of programming languages and frameworks, we observed hiccups when working with specialized domains or frameworks. This alludes to reduced suggestion quality due to underrepresentation in LLM training data, and thus also points to the need for exploration into long-term effects of proactive AI assistance across diverse technical contexts.

Furthermore, while Pu et al. found that flexible interaction scopes were effective, we find that in practice this leads to distraction and confusion due to context switching. This nuance suggests that in real-world IDEs, where various AI features may be integrated, future designs should even more carefully consider concurrent AI interaction contexts to minimize cognitive overhead and maximize developer productivity.

8

Limitations & Future Work

8.1. Threats to Validity

While our findings provide strong real-world insights into usage of and interaction with proactive AI in IDEs, several limitations must be acknowledged.

Construct Validity

Participants interacted with the proactive AI, integrated with a full-featured AI assistant. As such, post-study perceptions may have been shaped by the assistant as a whole. We see this as a strength of ecological integration, yet it introduces ambiguity in interpreting system usability scores. To this end, qualitative data was analyzed rigorously, confirming that the proactive AI contributed to the observed user experience — and thus making our qualitative analysis relevant nonetheless.

Internal Validity

Participants were exposed to all intervention types, but engagement varied by individual workflow (e.g. in terms of IDE usage) and day-to-day conditions. As telemetry data was anonymous, mapping individual machines to survey responses could not be done precisely. This means that in our work, both datasets have been analyzed separately. We ensured qualitative data only spanned participants who provided meaningful daily survey responses over at least five days (N=15). Due to the objective nature of telemetry, we included data from all participants who engaged with the proactive AI, even if they did not complete the full five-day protocol (N=18 machines in total with active usage declining to 8 machines by day five). This approach may have introduced noise, but nonetheless preserved participant privacy whilst maximizing available data.

Moreover, there were several factors that may have influenced study outcomes. Due to the undeterministic behavior of LLMs, participants experienced varied response quality; despite attempts to reduce randomness (e.g. ensuring LLM temperature set to 0 at all times), this inconsistency may have led to inaccuracy in findings. Also, we conducted our user studies on a nightly build of Fleet, hence not the most stable and thus potentially led to confounding factors in terms of user experience.

External Validity

Our work closely mirrors adoption of proactive AI in practice: as the prototype was integrated into an enterprise IDE, and our in-the-wild study participants engaged in self-directed tasks representative of their professional work — with no limits on languages and project types. Compared to prior work conducted in controlled laboratory environments [6, 43, 55], this approach offers higher ecological validity by capturing natural developer behaviors. However, this realism also introduces variability across users and sessions, which can complicate interpretation.

Participants, recruited from JetBrains' research panel, underwent a screening survey to ensure they were already familiar with the company's IDEs, and in-IDE AI tools in general. While this enabled

us to observe realistic use cases and workflows, findings may not fully generalize to developers less experienced with these tools or to those newer to AI-assisted development.

Moreover, our sample size was limited (N=18 for quantitative analysis and N=15 for qualitative analysis) — and engagement varied considerably across days and intervention types. Consequently, statistical testing would have had limited power, increasing the risk of Type II errors.

Engagement patterns were analyzed over a relatively short five-day period, leaving longer-term adoption dynamics unexplored. Additionally, we merely examined three heuristic-based intervention strategies throughout the development workflow, to steer future research into smarter trigger models.

8.2. Future Work

This work demonstrates the feasibility of integrating proactive AI into a production-grade IDE and evaluating it in an in-the-wild setting, advancing research on proactive coding assistance in real-world environments. Building on our design principles and empirical findings, several promising directions emerge for future exploration.

First, future systems should expand both the variety and sophistication of proactive triggers. While task boundaries (e.g. post-commit) proved effective, more advanced trigger models are needed to better capture developer intent and dynamically adjust intervention strategies. Such models could leverage richer signals from live code context, edit history and further in-IDE interactions. However, developing smarter trigger models will require more research into developer intent modelling, more comprehensive training data, and can introduce overhead costs.

Second, proactive support should move beyond code quality and could expand into domains such as debugging and testing, where timely assistance could further improve developer productivity.

Third, long-term behavioral impacts of proactive AI remain underexplored. Future studies should investigate how developers' trust, habituation, and reliance evolve over extended periods, as well as potential downstream effects on code quality and maintainability — especially in industrial and team-based settings.

Moreover, understanding individual differences in how developers accept, reject, or modify suggestions can inform more personalized and adaptive intervention strategies. Future work could include studies with larger and more diverse samples to assess the influence of factors such as experience level, role, and language preferences. Controlled deployments, such as A/B tests, can be particularly valuable here: they could systematically compare different proactive strategies, validate personalization approaches, and complement our in-the-wild findings by providing clearer comparative insights.

9

Conclusion

This thesis contributes to the growing body of research on proactive AI assistance in software development by designing, implementing, and evaluating a proactive chat assistant integrated within a commercial IDE. Our five-day in-the-wild study with 18 professional developers demonstrated that timing is a critical factor for engagement: suggestions delivered at workflow boundaries (e.g. post-commit) achieved highest engagement rates, while mid-task interventions were often dismissed. Importantly, we observed sustained engagement patterns over time, with half of the participants continuing to use the assistant beyond the study period — highlighting strong potential for integration into real-world development workflows.

Developers valued contextually relevant suggestions, particularly when the AI acted as a “second pair of eyes” to surface overlooked issues, supporting both perceived productivity and code quality. The wide variation in user preferences also underscores the need for adaptive, personalized proactive systems. While our study relied on heuristic-based triggers and a modest sample size, these point toward rich future directions in advanced intent modeling and longitudinal studies, possibly in A/B settings.

In conclusion, our work bridges the gap between experimental prototypes and practical integration, providing actionable design insights and empirical evidence for integrating proactive AI into everyday software engineering practice. This work lays a solid foundation for future research and industry adoption, supporting more seamless developer-AI collaboration.

References

[1] Brian P. Bailey and Shamsi T. Iqbal. “Understanding changes in mental workload during execution of goal-directed tasks and its application for interruption management”. In: *ACM Trans. Comput.-Hum. Interact.* 14.4 (Jan. 2008). ISSN: 1073-0516. DOI: 10.1145/1314683.1314689. URL: https://doi.org/10.1145/1314683.1314689.

[2] Shraddha Barke, Michael B. James, and Nadia Polikarpova. “Grounded Copilot: How Programmers Interact with Code-Generating Models”. In: *Proceedings of the ACM on Programming Languages* 7 (2022), pp. 85–111. URL: https://api.semanticscholar.org/CorpusID:250144196.

[3] Nancy K. Baym et al. “INTELLIGENT FAILURES: CLIPPY MEMES AND THE LIMITS OF DIGITAL ASSISTANTS”. In: *AoIR Selected Papers of Internet Research* (2019). URL: https://api.semanticscholar.org/CorpusID:242874985.

[4] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. “RepairAgent: An Autonomous, LLM-Based Agent for Program Repair”. In: *Proceedings of the 47th International Conference on Software Engineering (ICSE)*. 2025.

[5] Tianyi Chen. “The Impact of AI-Pair Programmers on Code Quality and Developer Satisfaction: Evidence from TiMi studio”. In: *Proceedings of the 2024 International Conference on Generative Artificial Intelligence and Information Security*. GAIIS ’24. Kuala Lumpur, Malaysia: Association for Computing Machinery, 2024, pp. 201–205. ISBN: 9798400709562. DOI: 10.1145/3665348.3665383. URL: https://doi.org/10.1145/3665348.3665383.

[6] Valerie Chen et al. *Need Help? Designing Proactive AI Assistants for Programming*. 2025. arXiv: 2410.04596 [cs.HC]. URL: https://arxiv.org/abs/2410.04596.

[7] Bhavya Chopra et al. *Conversational Challenges in AI-Powered Data Science: Obstacles, Needs, and Design Opportunities*. 2023. arXiv: 2310.16164 [cs.HC]. URL: https://arxiv.org/abs/2310.16164.

[8] Bhavya Chopra et al. *Exploring Interaction Patterns for Debugging: Enhancing Conversational Capabilities of AI-assistants*. 2024. arXiv: 2402.06229 [cs.HC]. URL: https://arxiv.org/abs/2402.06229.

[9] Umut Cihan et al. *Evaluating Large Language Models for Code Review*. 2025. arXiv: 2505.20206 [cs.SE]. URL: https://arxiv.org/abs/2505.20206.

[10] CognitionLabs. *Devin AI*. https://devin.ai/. 2024.

[11] Cursor. *Cursor: The AI-first Code Editor*. https://cursor.sh. 2023.

[12] George Demiris and Brian K Hensel. “Technologies for an aging society: a systematic review of “smart home” applications”. In: *Yearbook of medical informatics* 17.01 (2008), pp. 33–40.

[13] Shuzheng Gao et al. *Search-Based LLMs for Code Optimization*. 2024. arXiv: 2408.12159 [cs.SE]. URL: https://arxiv.org/abs/2408.12159.

[14] GitHub. *GitHub Copilot*. https://github.com/features/copilot. 2021.

[15] Google. *Google Jules*. https://jules.google/. 2025.

[16] Karen Holtzblatt and Hugh Beyer. *Contextual design: defining customer-centered systems*. Elsevier, 1997.

[17] Eric Horvitz. “Principles of mixed-initiative user interfaces”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI ’99. Pittsburgh, Pennsylvania, USA: Association for Computing Machinery, 1999, pp. 159–166. ISBN: 0201485591. DOI: 10.1145/302979.303030. URL: https://doi.org/10.1145/302979.303030.

[18] ICC/ESOMAR. *ICC/ESOMAR International Code on Market, Opinion and Social Research and Data Analytics*. Accessed on: June 09, 2025. 2022. URL: https://esomar.org/uploads/attachments/ckqtawvjq00uukdtrhst5sk9u-iccesomar-international-code-english.pdf.

[19] Shamsi T. Iqbal et al. "Towards an index of opportunity: understanding changes in mental workload during task execution". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI '05. Portland, Oregon, USA: Association for Computing Machinery, 2005, pp. 311–320. ISBN: 1581139985. DOI: 10.1145/1054972.1055016. URL: https://doi.org/10.1145/1054972.1055016.

[20] Imen Jaoua, Oussama Ben Sghaier, and Houari Sahraoui. *Combining Large Language Models with Static Analyzers for Code Review Generation*. 2025. arXiv: 2502.06633 [cs.SE]. URL: https://arxiv.org/abs/2502.06633.

[21] JetBrains. *About JetBrains AI service*. Accessed: May 30, 2025. 2025. URL: https://www.jetbrains.com/help/ai/jetbrains-ai.html.

[22] JetBrains. *JetBrains AI Assistant*. https://www.jetbrains.com/junie/. 2025.

[23] JetBrains. *JetBrains Fleet AI Assistant Features*. https://www.jetbrains.com/help/fleet/ai-assistant.html. 2024.

[24] Ellen Jiang et al. "Discovering the Syntax and Strategies of Natural Language Programming with Generative Language Models". In: *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI '22. New Orleans, LA, USA: Association for Computing Machinery, 2022. ISBN: 9781450391573. DOI: 10.1145/3491102.3501870. URL: https://doi.org/10.1145/3491102.3501870.

[25] Neal Lathia et al. "Smartphones for large-scale behavior change interventions". In: *IEEE Pervasive Computing* 12.3 (2013), pp. 66–73.

[26] James R Lewis. "The system usability scale: past, present, and future". In: *International Journal of Human–Computer Interaction* 34.7 (2018), pp. 577–590.

[27] Nelson F. Liu et al. *Lost in the Middle: How Language Models Use Long Contexts*. 2023. arXiv: 2307.03172 [cs.CL]. URL: https://arxiv.org/abs/2307.03172.

[28] Abhinav Mehrotra, Robert Hendley, and Mirco Musolesi. "PrefMiner: mining user's preferences for intelligent mobile notification management". In: *Proceedings of the 2016 ACM international joint conference on pervasive and ubiquitous computing*. 2016, pp. 1223–1234.

[29] Christian Meurisch et al. "Exploring User Expectations of Proactive AI Systems". In: *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.* 4.4 (Dec. 2020). DOI: 10.1145/3432193. URL: https://doi.org/10.1145/3432193.

[30] Microsoft. *IntelliSense*. https://learn.microsoft.com/en-us/visualstudio/ide/using-intellisense?view=vs-2022. 2023.

[31] Microsoft. *Visual Studio IntelliCode*. https://visualstudio.microsoft.com/services/intellicode/. 2018.

[32] A Cristina Mihale-Wilson, Jan Zibuschka, and Oliver Hinz. "User preferences and willingness to pay for in-vehicle assistance". In: *Electronic Markets* 29 (2019), pp. 37–53.

[33] Tim Miller. *Explanation in Artificial Intelligence: Insights from the Social Sciences*. 2018. arXiv: 1706.07269 [cs.AI]. URL: https://arxiv.org/abs/1706.07269.

[34] Aral de Moor, Arie van Deursen, and Maliheh Izadi. "A Transformer-Based Approach for Smart Invocation of Automatic Code Completion". In: *Proceedings of the 1st ACM International Conference on AI-Powered Software*. AIware '24. ACM, July 2024, pp. 28–37. DOI: 10.1145/3664646.3664760. URL: http://dx.doi.org/10.1145/3664646.3664760.

[35] Hussein Mozannar et al. *Reading Between the Lines: Modeling User Behavior and Costs in AI-Assisted Programming*. 2024. arXiv: 2210.14306 [cs.SE]. URL: https://arxiv.org/abs/2210.14306.

[36] Hussein Mozannar et al. *The RealHumanEval: Evaluating Large Language Models' Abilities to Support Programmers*. 2024. arXiv: 2404.02806 [cs.SE]. URL: https://arxiv.org/abs/2404.02806.

[37] Hussein Mozannar et al. “When to Show a Suggestion? Integrating Human Feedback in AI-Assisted Programming”. In: *AAAI Conference on Artificial Intelligence*. 2023. URL: https://api.semanticscholar.org/CorpusID:259108906.

[38] Daye Nam et al. *Using an LLM to Help With Code Understanding*. 2024. arXiv: 2307.08177 [cs.SE]. URL: https://arxiv.org/abs/2307.08177.

[39] OpenAI. *OpenAI Codex*. https://openai.com/index/introducing-codex/. 2025.

[40] Zhenyu Pan et al. *Do Code LLMs Understand Design Patterns?* 2025. arXiv: 2501.04835 [cs.SE]. URL: https://arxiv.org/abs/2501.04835.

[41] Veljko Pejovic and Mirco Musolesi. “InterruptMe: designing intelligent prompting mechanisms for pervasive applications”. In: *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. 2014, pp. 897–908.

[42] Dorin Pomian et al. “EM-Assist: Safe Automated ExtractMethod Refactoring with LLMs”. In: *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*. FSE ’24. ACM, July 2024, pp. 582–586. DOI: 10.1145/3663529.3663803. URL: http://dx.doi.org/10.1145/3663529.3663803.

[43] Kevin Pu et al. “Assistance or Disruption? Exploring and Evaluating the Design and Trade-offs of Proactive AI Programming Support”. In: *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. CHI ’25. ACM, Apr. 2025, pp. 1–21. DOI: 10.1145/3706598.3713357. URL: http://dx.doi.org/10.1145/3706598.3713357.

[44] Ranjan Sapkota, Konstantinos I. Roumeliotis, and Manoj Karkee. *Vibe Coding vs. Agentic Coding: Fundamentals and Practical Implications of Agentic AI*. 2025. arXiv: 2505.19443 [cs.SE]. URL: https://arxiv.org/abs/2505.19443.

[45] Agnia Sergeyuk et al. *Human-AI Experience in Integrated Development Environments: A Systematic Literature Review*. 2025. arXiv: 2503.06195 [cs.SE]. URL: https://arxiv.org/abs/2503.06195.

[46] Agnia Sergeyuk et al. *The Design Space of in-IDE Human-AI Experience*. 2024. arXiv: 2410.08676 [cs.SE]. URL: https://arxiv.org/abs/2410.08676.

[47] Tao Sun et al. *BitsAI-CR: Automated Code Review via LLM in Practice*. 2025. arXiv: 2501.15134 [cs.SE]. URL: https://arxiv.org/abs/2501.15134.

[48] Tabnine. *Tabnine*. https://docs.tabnine.com/main. 2020.

[49] Xin Tan et al. *How far are AI-powered programming assistants from meeting developers’ needs?* 2024. arXiv: 2404.12000 [cs.SE]. URL: https://arxiv.org/abs/2404.12000.

[50] Nalin Wadhwa et al. “CORE: Resolving Code Quality Issues using LLMs”. In: *Proceedings of the ACM on Software Engineering* 1 (July 2024), pp. 789–811. DOI: 10.1145/3643762.

[51] Justin D. Weisz et al. “Perfection Not Required? Human-AI Partnerships in Code Translation”. In: *26th International Conference on Intelligent User Interfaces*. IUI ’21. ACM, Apr. 2021, pp. 402–412. DOI: 10.1145/3397481.3450656. URL: http://dx.doi.org/10.1145/3397481.3450656.

[52] Windsurf. *Windsurf AI Code Editor*. https://windsurf.ai. 2023.

[53] Di Wu et al. “iSMELL: Assembling LLMs with Expert Toolsets for Code Smell Detection and Refactoring”. In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. ASE ’24. Sacramento, CA, USA: Association for Computing Machinery, 2024, pp. 1345–1357. ISBN: 9798400712487. DOI: 10.1145/3691620.3695508. URL: https://doi.org/10.1145/3691620.3695508.

[54] Yisen Xu et al. *MANTRA: Enhancing Automated Method-Level Refactoring with Contextual RAG and Multi-Agent LLM Collaboration*. 2025. arXiv: 2503.14340 [cs.SE]. URL: https://arxiv.org/abs/2503.14340.

[55] Sebastian Zhao et al. *CodingGenie: A Proactive LLM-Powered Programming Assistant*. 2025. arXiv: 2503.14724 [cs.HC]. URL: https://arxiv.org/abs/2503.14724.

A

General Usage Stats from Telemetry

Below we provide an overview of general usage patterns, based on quantitative telemetry data (N=18). Note that we treat this as a separate dataset from the qualitative dataset (N=15), as the anonymous nature of telemetry collected made accurate mapping from machines to participant names cumbersome.

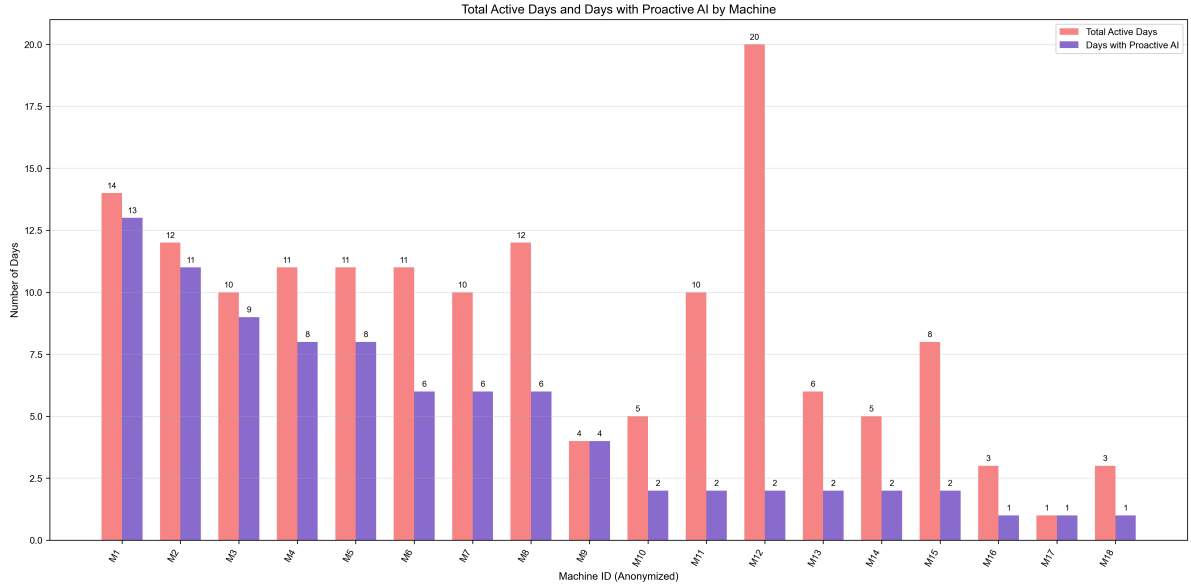


Figure A.1: The total number of days participants were active in terms of general usage of the IDE, and interacting with the proactive prototype under study specifically. Note that the type of interventions we introduced highly influence levels exposure to the proactive assistant.

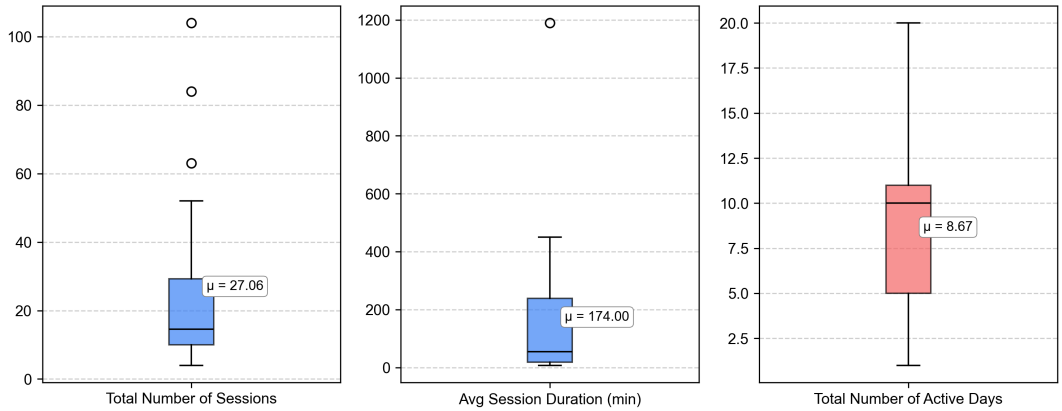


Figure A.2: Boxplots providing an overview participation in terms of total number of sessions, average session duration and total number of active days.

B

Qualification Screener Survey User Study

This Appendix contains the qualification survey that all invited participants (from JetBrains' research panel) underwent to avoid confounding factors in our study as much as possible. Specifically, eligibility criteria included:

- Having professional programming experience
- Being familiar with JetBrains IDEs
- Having regular experience with in-IDE AI tools in day-to-day development tasks

Welcome page

A. **Survey Introduction:**

Thank you for your interest in our research!

Please complete our five-minute survey, where we'll ask you about your professional background and experience.

B. **Please read the General Research Terms and Conditions**
   - ☐ I have read and agree to the General Research Terms and Conditions

C. **Please provide your contact details:**
   Note: Your email will only be used to match responses across study phases and will not be shared or used for other purposes.

   _____

D. **What type of reward would you like to receive if you fully participate in the study? (Radio button, select one)**
   - One-year JetBrains All Products Pack subscription
   - USD 150 Amazon Gift Card
   - Nothing, thank you

General questions

Which of the following JetBrains products do you currently use? (Checkbox, select multiple)

- None
- IntelliJ IDEA Ultimate
- IntelliJ IDEA Community Edition
- RubyMine
- PyCharm Professional Edition
- PyCharm Community Edition
- AppCode
- CLion

PhpStorm
WebStorm
GoLand
DataGrip
DataSpell
Rider
Fleet
ReSharper
ReSharper C++
dotCover
dotTrace
dotMemory
dotPeek
Space
TeamCity
YouTrack
Datalore
Upsource
Qodana
PyCharm Edu
IntelliJ IDEA Edu
JetBrains Academy
JetBrains Toolbox
Code With Me
JetBrains Gateway
MPS

IF FLEET CHOSEN:

How frequently do you use Fleet in software engineering projects?
Never
Less than once a month
A few times a month
About once a week
A few times a week

About once a day
Multiple times a day

How frequently do you use AI tools in your day-to-day development work?

Never
Less than once a month
A few times a month
About once a week
A few times a week
About once a day
Multiple times a day

IF AI [Never, Less than once a month, A few times a month] → end of survey.
IF AI [About once a week, A few times a week, About once a day, Multiple times a day]:

Which of the following AI tools do you use regularly?
None
Sourcegraph Cody
Visual Studio IntelliCode
Google Gemini (formerly Bard)
Codeium
Code Llama
GitHub Copilot
ChatGPT
Tabnine
JetBrains AI Assistant
Gemini Code Assist (formerly Duet AI for Developers)
CodeGPT plugin in an IDE
Microsoft 365 Copilot
Cursor
Anthropic Claude
Amazon Q Developer (previously CodeWhisperer)
Other, please specify:

IF AI [jb ai]:

What type of JetBrains AI Assistant license do you have? (Radiobutton, select one)

Trial
Pro

Enterprise (the license was purchased by my company)
Other, please specify: _____
I am not sure

Which of the following best describes your job role? (Checkbox, select multiple)

Developer / Programmer / Software Engineer
DevOps Engineer / Infrastructure Developer
Database Administrator
Architect
Tester / QA Engineer
Technical Support
Data Analyst / Data Engineer / Data Scientist
Business Analyst
Technical Writer
Team Lead
Systems Analyst
Product Manager / Marketing Manager
UX / UI Designer
CIO / CEO / CTO
Developer Advocate
Instructor / Teacher / Tutor
Other, please specify: _____

What are your primary programming languages? Choose no more than 3 languages. (Checkbox, select multiple)

Assembly
C
C#
C++
Clojure / ClojureScript
COBOL
CoffeeScript

Dart
Delphi
Elixir
F#
Go
GraphQL
Groovy
Haskell
HTML / CSS
Java
JavaScript
Julia
Kotlin
Lua
MATLAB
Objective-C
Perl
PHP
Platform-tied language (Apex, ABAP, 1C): _____
Python
R
Ruby
Rust
Scala
Shell scripting languages (bash/shell/powershell)
SQL (PL/SQL, T-SQL and other programming extensions of SQL)
Swift
TypeScript
Visual Basic
Other, please specify: _____*

How many years of professional coding experience do you have? (Radiobutton, select one)

- Less than 1 year
- 1–2 years
- 3–5 years
- 6–10 years
- 11–16 years
- 16+ years
- I don't have any professional coding experience

Which platform(s) do you use when working on your projects? (Checkbox, select multiple)

- Windows
- Linux
- macOS
- Other, please specify: _____

Thank you page

Thank you for completing this survey!
If your indicated experience matches our study criteria, you will receive a follow-up invitation within a week to participate in the main phase of the study.

If there's anything else you'd like to discuss, please contact us at surveys@jetbrains.com.

C

Daily Survey

Thanks for using the Fleet IDE with our new proactive AI feature!

This short form is part of an ongoing study to help us understand how proactive AI suggestions in the chat fit into your development workflow. It should take less than 2 minutes to complete.

There are no right or wrong answers – we're interested in your honest experience.

1. Which of the following AI Actions have you interacted with?
[ ] Edit Code
[ ] Review Changes With AI
[ ] I'm not sure / I don't remember
[ ] Other: _____

2. Which of the following were you involved in when the AI suggestion(s) appeared?
[ ] Fixing a bug
[ ] Improving quality or performance
[ ] Refactoring or cleaning up code
[ ] Reviewing my own commit
[ ] Reviewing someone else's changes
[ ] Writing new code
[ ] Other: _____

3. Did you interact with the AI suggestion(s) in the AI chat?
[ ] Yes, I applied and accepted them
[ ] Yes, but I modified them
[ ] No, I rejected them (actively dismissed or undid)
[ ] No, I ignored them (didn't respond to them at all)

4. How helpful were the AI suggestion(s) for your task?
Not at all helpful
Slightly helpful
Moderately helpful
Quite helpful
Very helpful

5. Was there anything about the suggestion(s) that stood out to you?
Were they surprising, confusing, helpful, unhelpful? Please describe some examples, like what the suggestion was and why you used or rejected it. If possible, please paste the chat messages or the generated code.

D

Post-Study Survey

Welcome page

A. **Survey Introduction:**

Thank you for your participation in our research into proactive AI features in Fleet!

Please complete our 15-minute survey, where we'll ask you about your experience during the study.

B. **Please provide your contact details:**
Note: Your email address will only be used to match responses across study phases. It will not be shared or used for any other purposes.

_____

Questions

All of the questions in this survey are focused on your experience with the proactive code improvement suggestions from AI.

Please indicate your level of agreement or disagreement with each of the following statements about the system, meaning the proactive code improvement suggestions from AI.

I think that I would like to use this system frequently.
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree

I found the system unnecessarily complex.
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree

I thought the system was easy to use.

Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree

I think that I would need the support of a technical person to be able to use this system.
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree

I found the various functions in this system were well integrated.
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree

I thought there was too much inconsistency in this system.
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree

I would imagine that most people would learn to use this system very quickly.
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree

I found the system very cumbersome to use.
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree

I felt very confident using the system.
Strongly disagree
Disagree

Neither agree nor disagree
Agree
Strongly agree

I needed to learn a lot of things before I could get going with this system.
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree

Please indicate how much you agree or disagree with the following statements about the proactive code improvement suggestions provided in the AI chat.

“I feel comfortable accepting the proactive code improvement suggestions provided in the AI chat.”
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree

“I trust that proactive code improvement suggestions provided in the AI chat are reliable.”
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree

“It was easy to understand how proactive code improvement suggestions provided in the AI chat were related to what I was trying to do.”
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree

“The proactive nudges for code improvement suggestions felt well-timed and unobtrusive.”
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree

“Using the proactive code improvement suggestions provided in the AI chat made it easier to complete my tasks.”
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree

“Using the proactive code improvement suggestions provided in the AI chat helped me complete my tasks more quickly.”
Strongly disagree
Disagree
Neither agree nor disagree
Agree
Strongly agree

Could you please describe how the proactive suggestions affected the improvement of your code?
Please include specific examples if possible, such as improvements in quality or performance, or suggestions that were unhelpful or misleading. If you strongly agreed or disagreed with any of the earlier statements, feel free to explain why here.
(open-ended)

What would you change about the proactive code improvement suggestions to make them more useful, reliable, or better integrated into your workflow?
(open-ended)

E

Semi-Structured Interview Pilot Study

Hello, and thanks for joining us today! I'm ____, and this is ____. We are researchers in the Human-AI Experience team at JetBrains.

Today, we'd like to explore a new AI feature prototype in Fleet that supports improving code. Have you participated in a study like this before?

For this session, you'll use a prototype to complete a small coding task. We'll ask you to think aloud as you work so we can understand your impressions and thoughts. There are no right or wrong answers — our goal is to learn from your experience and refine the prototype. Your feedback, whether positive or negative, is incredibly valuable.

Does that sound good? Great!

Were you able to start the app successfully?
→ If yes: Great!
→ If no: Let's quickly troubleshoot before proceeding.

Also, we'd like to record this session for analysis. The recordings will remain confidential, used only to create anonymized summaries, and deleted after the research is complete. Is that okay with you?

→ If yes: I'll now start the recording and ask for your permission once more.

[start recording]

Can you please confirm that you consent to this session being recorded?
→ If yes: Thank you!

With all the technical setup done, let's move on to the main part.

To start, could you tell me a bit about your experience with Fleet and generative AI for coding?

- How often do you use Fleet?
- Overall, how long have you been using Fleet?
- How familiar you are with AI actions in Fleet?
- Which AI-assisted coding tools did you try?
- Overall, how often do you use AI for coding-related tasks?

Thank you! That's really helpful to have more context.

Now, let's open Fleet and navigate to the project you have selected.

Your task is to optimize any function of your choice using AI actions in Fleet.

As you work, please think aloud — share what you're noticing, what you're trying to do, and any thoughts or reactions you have. If something is confusing or unexpected, let us know. If you pause or feel unsure, describe what's on your mind.

Let's begin! First, describe what you see on the screen.

→ Let the participant explore and talk freely.

Now, go ahead and try to optimize code using AI actions in Fleet. When you're done, just let us know.

Thanks for completing the task! Now, let's talk about your experience.

- How do you think this prototype AI feature works?
- How did you feel using it?
- What did you like about this feature?
- What didn't you like about it?
- How would you improve this feature to make it more useful for you?

- What other triggers would be useful (and not intrusive)?

Colleagues, do you have any questions here?

Is there anything else you'd like to add? Do you have any questions for us?

Thank you for your time and insights! Your feedback is incredibly valuable in shaping this feature. If you have any further thoughts later, feel free to reach out to us.

Thanks again, and have a great day!

[say goodbye]
[stop the recording]

User Study Setup Guide

Setup Instructions: Proactive AI Chat Assistance Study

Welcome!

This guide walks you through the required steps to ensure your Fleet IDE is setup correctly to successfully participate in the study on proactive AI assistance for code quality improvement. Note: all screenshots in this setup guide were taken on a *MacOS X (15.3.2, aarch64)* device, but the required setup steps are universal across different OS versions.

In case you have questions or run into issues, please check the "*Troubleshooting Tips*" section — or contact Nadine Kuo [nadine.kuo@jetbrains.com].

1. Installing Your Fleet Build

Download the special Fleet build (v1.49.204) using any of the installers below, depending on your OS.

- Mac ARM (M series) installer
- Windows ARM installer
- Linux ARM installer
- Mac x86_64 (Intel) installer
- Windows x86_64 installer
- Linux x86_64 installer

Note: MacOS users may experience "*Apple could not verify "Fleet.app" is free of malware that may harm your Mac or compromise your privacy*" when attempting to open the application. To bypass the above, you can execute the following in a terminal to clear extended attributes:

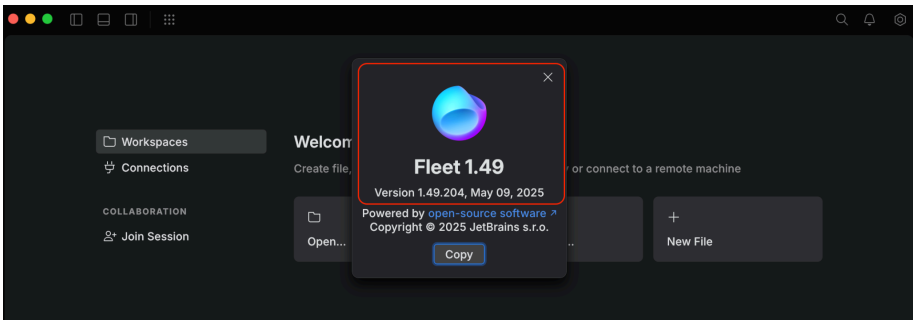
```
xattr -c <path/to/your/installed/Fleet.app>
```

2. Opening Your Workspace in Fleet

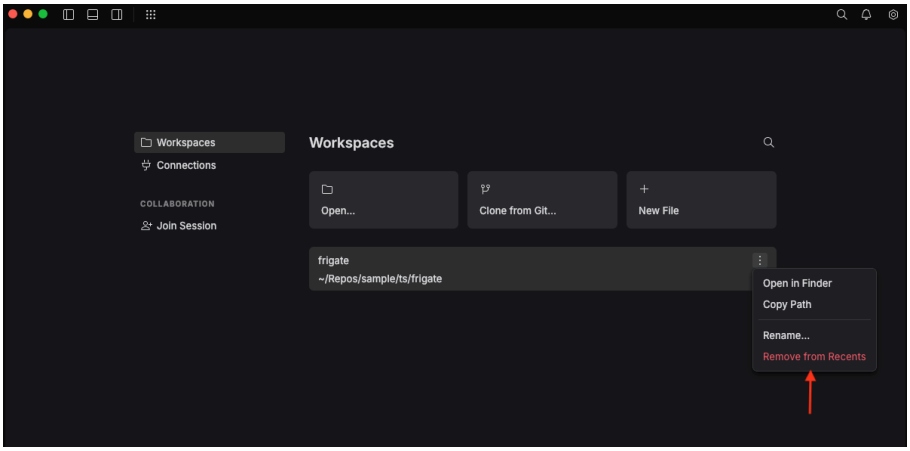
Upon opening the installed Fleet application:

- Check that the Fleet version is 1.49.204 (e.g. via *Fleet – About Fleet* in top menu on MacOS)
- Open any project that is representative of your day-to-day development work.

(see the screenshot on the next page)

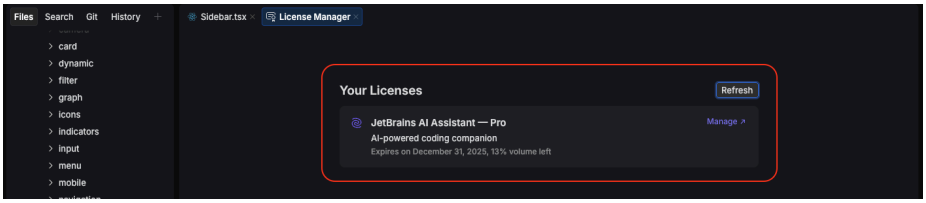


If you have workspaces listed already, please remove these from recents first to delete any cached data from other Fleet versions you may have worked with earlier.
Note: if it does not get erased from the list on clicking ***"Remove from Recents"***, you may have to retry.

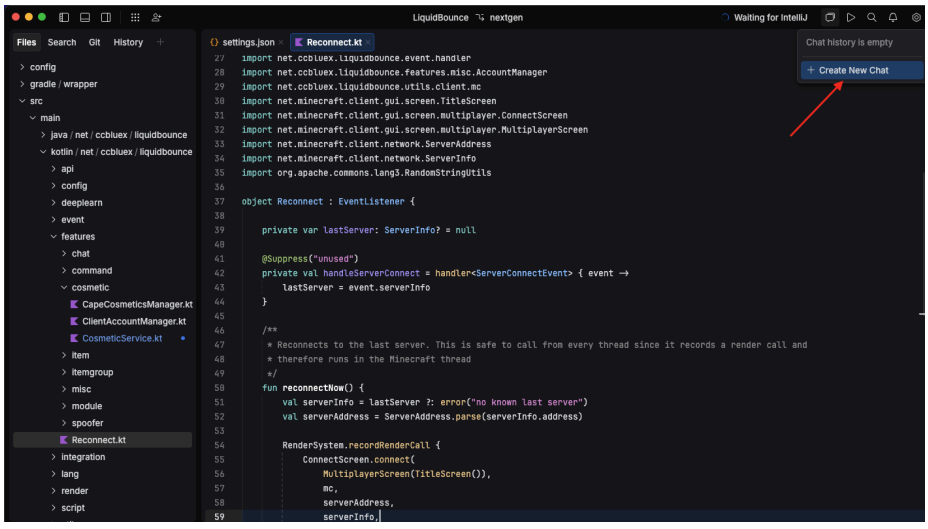


3. Check Your JetBrains AI Assistant License

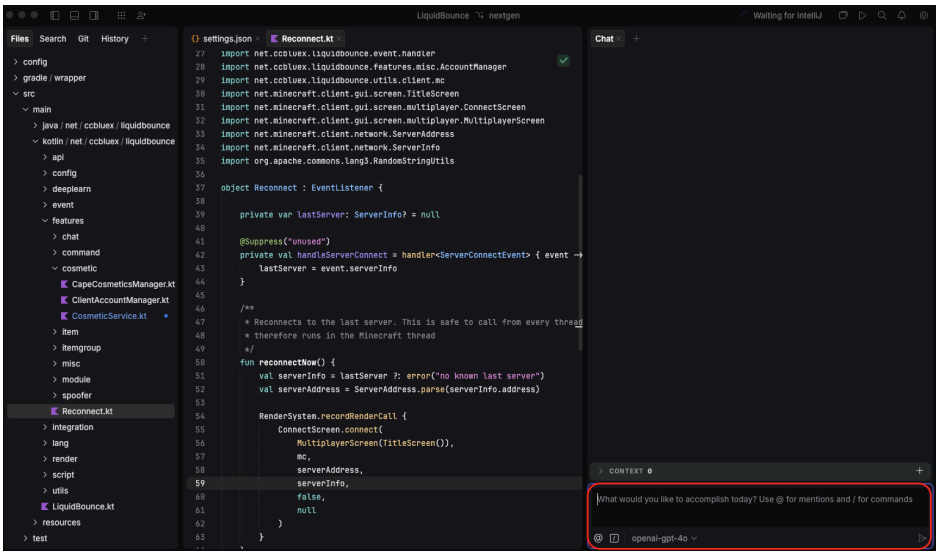
If you are not logged into your JetBrains account yet, please do so first. In case you have an active license already (Pro, Enterprise, Trial), check that your license shows up correctly under *Settings – License manager.* Else, you can request a 30-day Pro Trial which is activated immediately.



To check everything is working, invoke the AI Assistant via the chat icon in the top right corner:



Check that you are able to send requests, possibly using mentions (@) and commands (/):



Make sure that *GPT-4o* is selected at all times to ensure the most stable experience – we have only recently added support for the other models.

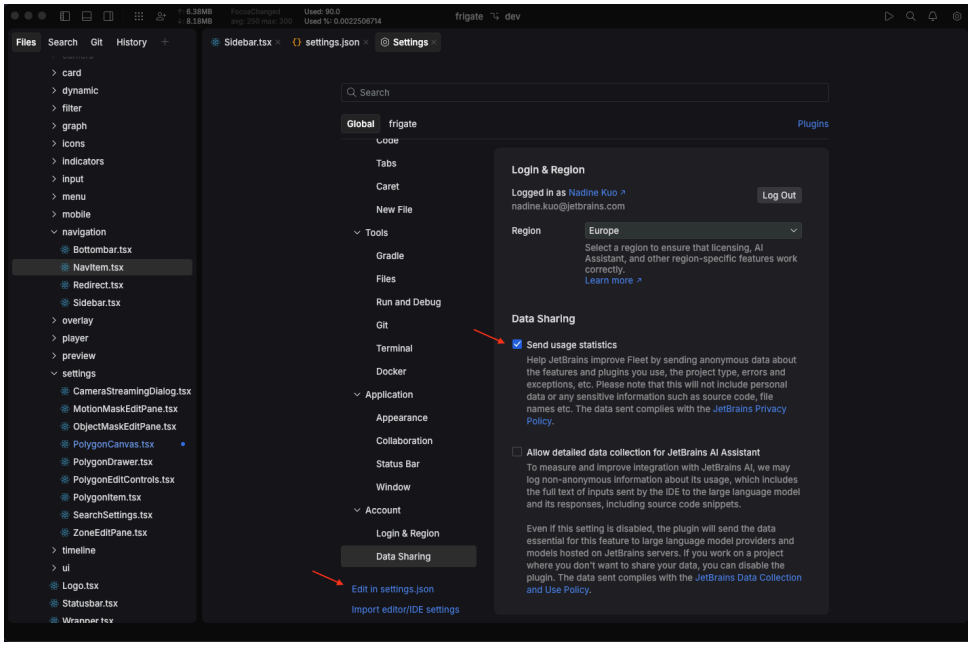
More information on JetBrains Fleet AI features can be found here:
https://www.jetbrains.com/help/fleet/ai-assistant.html

In the circumstance that you run out of volume or your license expires during the study, please shoot an e-mail to Nadine Kuo [nadine.kuo@jetbrains.com] with the following information:

- First and last name
- E-mail associated with your JetBrains account (see https://account.jetbrains.com/profile-details)

4. Configure Settings

Navigate to *Settings* – *Account* – *Data Sharing* and ensure that at least “Send usage statistics” is checked, which allows us to analyze user interactions with the features under study. Here, we also detail what kind of data will (not) be collected from our participants.



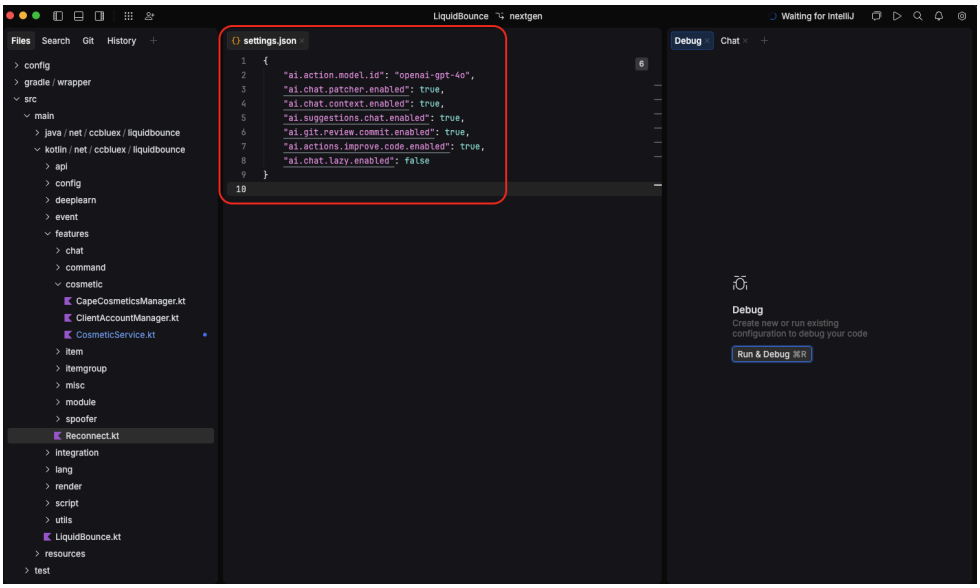
Next, navigate to “*Edit in settings.json*”.

In the `settings.json` file that is opened, copy over the following to enable/disable all necessary feature toggles:

```
{
 "ai.action.model.id": "openai-gpt-4o",
 "ai.chat.patcher.enabled": true,
 "ai.chat.context.enabled": true,
 "ai.suggestions.chat.enabled": true,
```

```
  "ai.git.review.commit.enabled": true,
  "ai.actions.improve.code.enabled": true,
  "ai.chat.lazy.enabled": false
}
```

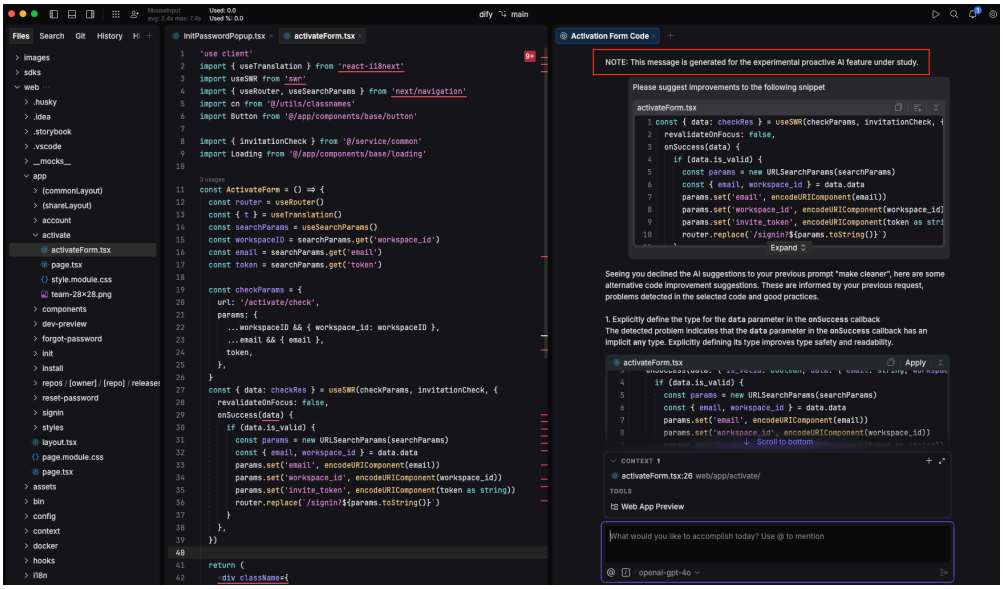
Your `settings.json` file should now look like:



Interacting with the AI Features Under Study

In this study, our aim is to understand how developers interact with several experimental proactive AI features for code quality improvement.

In the AI Assistant chat, these AI suggestions can be recognized by the following on top in the chat: *“NOTE: This message is generated for the experimental proactive AI feature under study*”. You may have to scroll a bit to the top to see this.



In order to invoke these chat suggestions, we ask you to at least perform the following during your work in the provided Fleet build:

- Iterating on code with AI – via the AI Action *“Edit Code*”, available upon selecting code via the floating toolbar, AltEnter popup or right-click menu [Link to GIF]
- Committing code – via the Git version control panel on the left [Link to GIF]

Note: this study is focused on the in-chat AI suggestions invoked via in-editor pop-ups as shown in the GIFs, not on the actions (e.g. *Edit Code*) themselves.

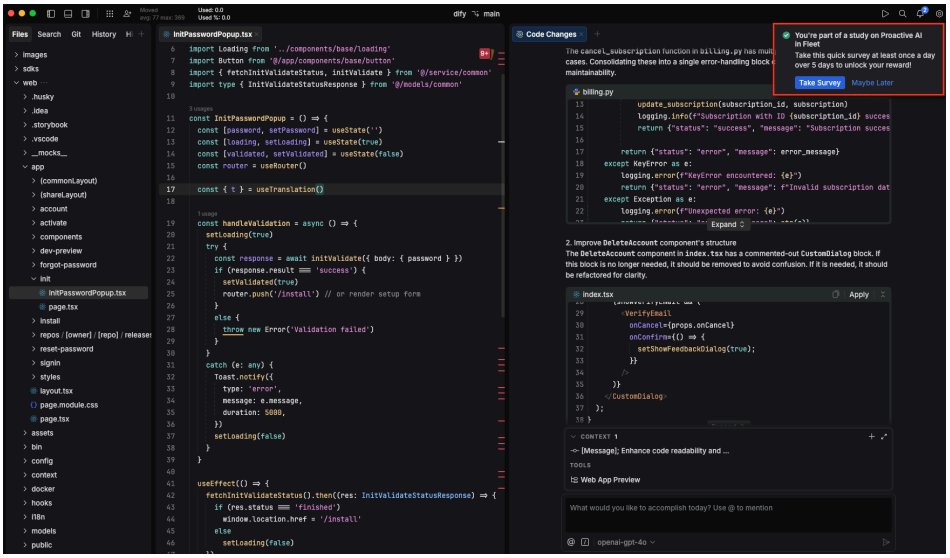
Of course, you are free to use any other AI Actions in Fleet, and interact with the AI Assistant in any way you wish. The most important to us, is that the work performed is representative of your day-to-day development work.
Again, make sure that *GPT-4o* is selected at all times to ensure the most stable experience.

Before you start, we highly suggest you to read the *Troubleshooting Tips* below.

How to Claim Your Reward?

You will receive survey notifications upon interacting with any of the AI features under study – at most once per Fleet session. In case you prefer to fill it out after your work on a given day, use this link: https://surveys.jetbrains.com/s3/HAX-Proactivity-Daily-Survey.

NOTE: in the top-right corner, check that your notifications are *not* muted (enabled by default).



We will only reward those participants that have filled out:

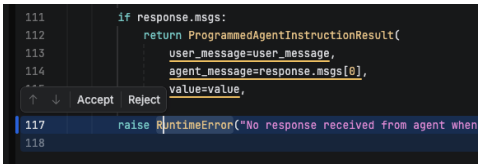
- Above survey at least once a day (~ 5 min.) across at least 5 days (need not be consecutive) between May 19, 2025 and May 28, 2025
- The post-study survey (~ 15 min.) which you will receive per e-mail on May 28, 2025

Troubleshooting Tips

Unfortunately, instability is expected as this Fleet build used for the study is not a Public Preview version. You may ignore error notifications that can possibly occur whilst working with the application, unless you cannot proceed with your work – in that case, please have a look if any of the tips below help, or contact Nadine Kuo [nadine.kuo@jetbrains.com].

I'm encountering exceptions or unresponsiveness when using the Accept / Reject options in the floating toolbar inside the editor

Please use the in-chat options to accept or reject snippets instead – our apologies for the inconvenience (this is a recent feature that is still in progress).



I'm encountering exceptions or unresponsiveness during interaction with in-chat code snippets (applying, accepting, rejecting, undoing)

1. Resend your prompt, or reinvoke the action that triggered chat invocation
2. If the issue persists, try another code snippet, file or project
3. Alternatively, you can manually copy over in-chat code snippets to your file

Note: we are aware of issues that may occur in case of undo operations following rejected snippets – please try to refrain from performing this action.

I'm getting "Client Error Exception during request to…" in the AI chat

1. Ensure that your active license shows up under *Settings - License Manager*
2. Restart the application and remove the workspace you are working in from recents (see step 2), after which you can try opening it again

I'm seeing an eternal progress spinner instead of model picker in the AI chat

Remove all contents from `settings.json`, save the file – after which you can paste the contents as shown above again.