School of Economics, Administration and Computer Science

Explainable AI Code Generation Assistant

Iuliia Liutova

Supervisor: Alexander Avdiushenko

May 2026

School of Economics, Administration and Computer Science

Explainable AI Code Generation Assistant

Dissertation which was submitted for obtaining a BSc degree in Applied Computer Science at Neapolis University Pafos

Iuliia Liutova

Examiners Committee

May 2026

Copyrights

Copyright (c) Iuliia Liutova, 2026. All rights reserved.
The approval of the Dissertation by Neapolis University does not necessarily imply the acceptance of the author's views on behalf of the University.

Contents

Validity Page

Student's Name & Surname: Iuliia Liutova

Dissertation Title: Explainable AI Code Generation Assistant

This Dissertation was prepared in the context of the studies for obtaining a BSc degree in Applied Computer Science at Neapolis University Pafos and was approved by the members of the Examiners' Committee.

Examiners' Committee

First Supervisor (Neapolis University Pafos): Alexander Avdiushenko

Committee member

Committee member

Declaration

I, Iuliia Liutova, being fully aware of the consequences of plagiarism, declare responsibly that this paper entitled "Explainable AI Code Generation Assistant", is strictly a product of my own personal work and all sources used have been duly stated in the bibliographic citations and references. Where I have used ideas, text and/or sources of other authors, they are clearly mentioned in the text with the appropriate citation and the relevant reference is included in the bibliographic references section with a full description.

The Declarant

Summary

English Summary

Large language models can generate and modify code for non-trivial programming tasks, but their explanations often remain generic or weakly connected to the implementation. When an agent edits several files, the developer must understand what changed and where each explanation is grounded in the code. The main objective of this dissertation is to design, implement, and evaluate an explainable AI code generation assistant that connects natural-language explanations to source-code lines.

The system was implemented as an IntelliJ plugin. It combines code generation through Junie CLI with explanation generation, multiple detail levels, structured and unstructured formats, explanation-to-code mappings, editor highlighting, and optional insertion of mapped comments. The methodology includes two evaluation directions. First, a mini-dataset based on annotated labml.ai implementations compares generated explanations with human-written explanations using an LLM-as-a-judge score. Second, a paired SWE-bench Verified Mini experiment compares normal and explanation-enriched repositories.

The findings show that the plugin can produce useful explanation artifacts for realistic generation and repair workflows. In the explanation-quality evaluation, the best OpenAI configuration reached a mean score of 0.786, while the Junie-based setting reached 0.752. In SWE-bench Verified Mini, explanation-enriched repositories solved 33 out of 50 tasks, compared with 32 out of 50 in the regular condition. The results do not prove a statistically significant improvement, but they indicate that grounded explanations can support review without harming agent performance. The dissertation concludes that explanation-to-code mapping is most valuable as a verification layer for AI-generated code, while broader claims about trust and productivity require larger benchmarks and human studies.

Keywords: explainable AI, code generation, coding agents, IntelliJ plugin, explanation to code mapping.

Greek Summary

Τα μεγάλα γλωσσικά μοντέλα μπορούν πλέον να παράγουν και να τροποποιούν κώδικα για σύνθετες προγραμματιστικές εργασίες, όμως οι εξηγήσεις τους συχνά παραμένουν αφηρημένες, γενικές ή ανεπαρκώς συνδεδεμένες με την πραγματική υλοποίηση. Αυτό δημιουργεί πρόβλημα επαλήθευσης: όταν ένας πράκτορας τροποποιεί πολλά αρχεία, ο προγραμματιστής πρέπει να κατανοήσει όχι μόνο τι άλλαξε, αλλά και σε ποιες γραμμές κώδικα βασίζεται κάθε εξήγηση. Ο βασικός στόχος της παρούσας διπλωματικής εργασίας είναι ο σχεδιασμός, η υλοποίηση και η αξιολόγηση ενός εξηγήσιμου βοηθού παραγωγής κώδικα, ο οποίος συνδέει εξηγήσεις φυσικής γλώσσας με συγκεκριμένες γραμμές πηγαίου κώδικα.

Το σύστημα υλοποιήθηκε ως πρόσθετο για το IntelliJ IDEA. Συνδυάζει παραγωγή κώδικα μέσω του Junie CLI με παραγωγή εξηγήσεων, πολλαπλά επίπεδα λεπτομέρειας, δομημένες και μη δομημένες μορφές παρουσίασης, ρητές αντιστοιχίσεις ανάμεσα σε εξηγήσεις και κώδικα, διαδραστική επισήμανση στον επεξεργαστή και προαιρετική εισαγωγή επεξηγηματικών σχολίων στα αρχεία. Η μεθοδολογία περιλαμβάνει δύο κατευθύνσεις αξιολόγησης. Πρώτον, χρησιμοποιείται ένα μικρό σύνολο δεδομένων βασισμένο σε επισημειωμένες υλοποιήσεις του labml.ai, ώστε οι παραγόμενες εξηγήσεις να συγκριθούν με ανθρώπινες παιδαγωγικές εξηγήσεις μέσω μετρικής LLM-as-a-judge. Δεύτερον, ένα ζευγαρωμένο πείραμα βασισμένο στο SWE-bench Verified Mini συγκρίνει κανονικά αποθετήρια με αποθετήρια εμπλουτισμένα με εξηγήσεις σε εργασίες αυτόματης επιδιόρθωσης.

Τα ευρήματα δείχνουν ότι το πρόσθετο μπορεί να δημιουργεί χρήσιμα τεχνουργήματα εξήγησης για ρεαλιστικές ροές εργασίας παραγωγής και επιδιόρθωσης κώδικα. Στην αξιολόγηση ποιότητας εξηγήσεων, η καλύτερη διαμόρφωση OpenAI πέτυχε μέση βαθμολογία 0.786, ενώ η διαμόρφωση Junie πέτυχε 0.752 και έδειξε σταθερή συμπεριφορά στα αξιολογημένα αρχεία. Στο SWE-bench Verified Mini, τα εμπλουτισμένα αποθετήρια έλυσαν 33 από τις 50 εργασίες, σε σύγκριση με 32 από τις 50 στο κανονικό σενάριο. Τα αποτελέσματα δεν αποδεικνύουν στατιστικά σημαντική βελτίωση, αλλά δείχνουν ότι οι θεμελιωμένες εξηγήσεις μπορούν να υποστηρίξουν τον έλεγχο χωρίς να βλάπτουν συστηματικά την απόδοση του πράκτορα. Η εργασία καταλήγει ότι η αντιστοίχιση εξήγησης και κώδικα είναι ιδιαίτερα χρήσιμη ως επίπεδο επαλήθευσης και επιθεώρησης για κώδικα που παράγεται από τεχνητή νοημοσύνη.

Λέξεις-κλειδιά: εξηγήσιμη τεχνητή νοημοσύνη, παραγωγή κώδικα, προγραμματιστικοί πράκτορες, πρόσθετο IntelliJ, αντιστοίχιση εξήγησης και κώδικα.

Abstract

Large language models are capable of generating code for solving non-trivial programming tasks. However, the explanations that accompany generated code often remain overly abstract and poorly related to specific implementation decisions. This slows down verification, debugging, and maintenance, especially when an agent modifies several files in different parts of a repository. This dissertation presents an explainable AI code generation assistant implemented as an IntelliJ plugin. The system combines code generation through Junie CLI with multi-level code explanations, explicit mappings between explanation components and source-code lines, interactive highlighting, and optional insertion of explanation comments into the edited code.

1. Introduction

Programming assistants using artificial intelligence are increasingly able to solve non-trivial tasks, but the quality of explanations remains the main obstacle. In many tools, users receive either a short explanation that is difficult to relate to the implementation, or a long narrative without a clear justification for the modified lines.

This issue is becoming more relevant as programmers move from autocomplete to task-level repository editing. Recent work highlights the value of natural language representations for code modification [1], documentation creation [2], outlines-based code understanding [3]. However, in practice, there remains a gap between creating explanations and bringing them in line with the code.

Moreover, modern programming assistants are no longer limited to offering individual code snippets: they are acting as autonomous agents who plan, modify, test, and improve software systems. In this context, explanations supported by the agent's thoughts can help not only human developers understand the implemented changes, but also other agents to use these explanations as a better context for further generation. Such explanations can become a form of structured communication between agents, allowing them to reason about previous modifications, reuse solutions, and create improved code in subsequent tasks.

This dissertation bridges this gap with an assistant developed based on three properties: code generation, explanation, and explicit mapping of explanation elements to code changes. The assistant generates code and explanations aligned with the code.

2. Literature Review

Recent advances show that AI-based programming assistants can efficiently generate code, but often cannot make their outputs easy to verify. In this dissertation, explainability is not considered as a separate property. It covers at least four interrelated levels: an explanation of the generated code artifact, an explanation of the agent's observed behavior over time, data on the model's input-output dependencies, and an explanation that supports appropriate human trust. This distinction is important because modern programmers are moving from autocomplete to task-level agents that check repositories, edit multiple files, launch tools, and pursue a goal described by the user. In such circumstances, a brief explanation is not enough. The developer must understand what has changed, why this change is plausible, where the explanation is grounded in the code, and how much the result should be trusted.

2.1 Explainability in Human-Agent Systems

The general literature on XAI provides a conceptual framework for this work. Rosenfeld and Richardson [4] describe explainability in human-agent systems using questions such as why an explanation is needed, who gets it, what is explained, when it is provided, and how it is presented. This taxonomy is useful for programming agents because it prevents a common simplification: using any natural language text next to the code as an explanation. The explanation for the programming assistant should be tied to a specific user need, such as checking a patch, checking a changed function, or understanding why a test failure led to a certain edit.

Miller [5] argues that AI explanations should be based on how people generate, select, and evaluate explanations in social settings. This is important for software development because developers usually explain code selectively. They do not describe each token; instead, they point to a key design decision, an appropriate limitation, or an unexpected line. Similarly, Langer et al. [6] emphasize stakeholder desiderata: different users require different forms of explanation. A novice developer may need a conceptual description, a maintainer may need a line-level rationale, and a reviewer may need proof that the generated patch matches the context of the repository.

Agent-oriented XAI also emphasizes the importance of time and process. Alzetta et al. [7] argue that explanations in multi-agent systems should be available to the user in time so that they can act in accordance with them. For a programming assistant, this means that explanations are most useful during review and debugging, and not just after the patch has already been accepted. Sado et al. [8] review explainable goal-driven agents and robots, and show that explanations can relate to goals, beliefs, intentions, plans, and actions. This is directly related to programming agents: the repository editing agent not only creates text, but also forms subgoals, selects files, invokes tools, and reviews its work. Therefore, a complete explanation of the coding agent using artificial intelligence should eventually cover both the modified code and the observed process that led to it.

2.2 Trust and Verification in AI-Assisted Programming

The software engineering literature makes the need for an explanation more specific. Wang et al. [9] studied developers' trust in AI-based code generation tools and found that trust is determined by expectation setting, configuration, and validation of suggestions. Their findings are important because they show that explanation is not just a function of usability. This supports calibrated trust: the developer should not blindly accept the generated code or reject useful suggestions due to the opacity of the tool.

As programming tools become more autonomous, the verification burden increases. Chen et al. [10] report that programming agents can reduce effort compared to less autonomous tools, but wider adoption depends on whether users understand the agents' behavior. This point distinguishes agentic code generation from the usual code completion. When the assistant modifies several files, the developer needs to check not only the final diff, but also the relationship between the task, the selected files, the generated implementation and the checks performed. Thus, the literature suggests that an explainable coding assistant should provide artifacts that can be used in the code review process.

2.3 Natural-Language Representations of Code

Several recent articles have addressed this issue using natural language as an interface between humans and code. NaturalEdit [1] is the most appropriate reference for this dissertation. It treats natural language summaries as interactive objects related to code. Its key features are an abstraction gradient that allows summaries to be displayed at different levels of detail; interactive matching of natural language phrases

and lines of code; and bidirectional synchronization so that changes in natural language and code remain consistent. The technical assessment of the authors and user research show that developers can use such representations to improve understanding, formulate intentions, and verify validity.

Natural Language Outlines [3] offers a corresponding structured interface. With this approach, each function is accompanied by an outline: a set of short instructions in natural language that summarize the main ideas of the code. The article presents outlines as a modern form of literate programming supported by LLMs. Outlines can help with navigation, maintenance, code generation, and summarizing code review results. Compared to NaturalEdit, the focus is not on direct editing using natural language, but on presenting the structure and purpose of the code in a form that matches the implementation.

DocAgent [2] expands the discussion from individual functions to repository-wide documentation. It builds a dependency graph, processes code components in a topological order, and coordinates the work of specialized agents who read, search, write, verify, and organize documentation creation. The system evaluates documentation in terms of completeness, usefulness, and truthfulness. This is relevant to this thesis because it shows that high-quality explanations of code bases require context management and validation, not just a single file query. At the same time, DocAgent is primarily a documentation generation system. It is not focused on explaining a specific patch created by the agent, or linking explanatory components to modified lines in the IDE.

2.4 Explanation as a Generation Context

Another line of work considers explanations not only as post-hoc descriptions for people, but also as a context that can improve generation. Geng et al. [11] show that code comments can serve multiple purposes for a developer, such as explaining what a piece of code does, why it exists, and how it should be used. This is an important correction to the simple presentation of explanations with low, medium, and high levels of detail. Different levels of detail help, but the quality of the explanations also depends on which question the developer is answering.

The MANGO approach [12] asserts that comments can act as natural logic pivots between problem statements and code. By prompting or training models to create inline logic comments, the authors improve code generation performance in benchmarks such as HumanEval and MBPP. This supports the intuition behind repositories enriched with explanations: grounded explanations can help future programming agents decompose the task and understand the surrounding implementation. However, this

advantage depends on correctness. Macke and Doyle [13] show that incorrect documentation can harm LLM code understanding to a greater extent than the absence or incompleteness of documentation. In this dissertation, this result is a warning. Inserted explanatory comments are only useful if they match the code. A plausible but incorrect explanation can mislead both the human reviewer and the future agent.

2.5 Positioning of This Dissertation

The literature reviewed suggests that explainable AI coding assistance should be evaluated using four questions. First, is the explanation clear to the stakeholders and is it useful for solving the developer's problem? Secondly, is it available at the right time in the workflow? Third, is it actionable in the code interface? Fourth, is it faithful enough to support calibrated trust?

The system developed in this dissertation mainly focuses on the third question and partly on the others. It provides multi-level summaries, explicit mappings of explanatory components to lines of source code, interactive highlighting, and optional insertion of explanatory comments. This is closely consistent with the principles of NaturalEdit and Natural Language Outlines, since it treats natural language as a structured representation of code, rather than as an ungrounded response in a chat. It also relies on DocAgent, treating explanation generation as a structured pipeline that can use different providers and stored artifacts.

At the same time, the limits of this contribution are explained in the literature. The current system explains the generated code artifact more clearly than the internal logic of the underlying coding model. It uses modified segments, prompts, and filtered agent logs to provide context, but does not prove a causal relationship between the hidden state of the model and the final patch. Thus, the most accurate framing for the dissertation is an auditable, explanation-grounded AI coding assistant. Its contribution is to simplify the verification of AI-generated code in an IDE environment, leaving deeper research on model attribution and calibrating people's trust for the future.

3. Methodology

3.1 Methodological Overview

The main hypothesis of this thesis is that the generated code becomes easier to verify when explanations are viewed as structured artifacts rather than as simple descriptive text added after the fact. Thus, the developed system follows three methodological principles. First, explanations should be generated at several levels of abstraction, since users need both a brief overview and a detailed understanding of the implementation. Secondly, each component of the explanation should be linked to specific lines of code so that the user can check whether the explanation is based on the implementation. Thirdly, explanations should not only explain which code is written, but also why it is implemented in this way.

NaturalEdit[1] is the main design reference. This demonstrates that natural language summaries can become an interactive representation of code, not just documentation. The developed plugin retains this idea, but changes the environment and workflow. NaturalEdit is an extension for VS Code with React web browsing, editing summaries, and synchronization between edited summaries and code. The implementation in this thesis is the IntelliJ IDEA plugin, the main use case of which is the generation of agent code using the Junie CLI, followed by an automatic explanation of the modified code. Thus, the developed plugin focuses on the actions of the agent, the integration of IntelliJ, and the artifacts of linked explanations that can be inserted into the codebase.

3.2 Ethics and Data Statement

This dissertation does not involve human participants, interviews, questionnaires, medical data, or personal data collection. The empirical part of the work uses publicly available open-source repositories, public benchmark datasets, generated explanations, and local execution logs produced during software experiments. Therefore, bioethics approval was not required for the conducted study. The evaluation data were used only for software engineering experimentation, and the official gold patches from SWE-bench were not provided to the repair agent during the repair phase.

3.3 System Architecture

The implementation is located in explainable-ai-plugin. It is a Kotlin-based IntelliJ Platform plugin built with Gradle, JDK 21, Kotlin 2.1.20, the IntelliJ Platform Gradle plugin, OkHttp, Kotlin serialization, and Kotlin coroutines. The plugin registers an IntelliJ tool window named AI assistant, a settings page under Tools -> Explainable AI, project-level services, and editor popup actions.

The architecture is organized around the following components:

- MyToolWindow builds the main user interface, stores the current explanations and mappings, displays generation logs, and coordinates user actions.
- OpenAIService wraps OpenAI API calls for direct explanation and mapping generation.
- JunieCliService executes Junie CLI for code generation and for Junie-based explanation generation.
- CodeChangeDetector captures pre-generation snapshots of files and computes changed line ranges after Junie modifies files.
- The comment insertion service inserts high-detail structured explanations into source files as comments using the language-specific comment style.
- OpenAISettings and OpenAISettingsConfigurable store the selected explanation provider, model, temperature, token limit, OpenAI key, and Junie token.

This separation is important because the plugin has two different artificial intelligence providers. The OpenAI path is useful for faster and cheaper explanations and mappings using a selectable model such as gpt-5.5, gpt-5.4, etc. The Junie path is useful when an explanation needs to be prepared with a deeper analysis of the repository context. The configuration level allows the user to switch between these providers without changing the rest of the workflow.

3.4 Agent-Based Code Generation

The first implemented section of the plugin is the workflow of agent-based code generation. This section is responsible for converting the user's natural language request into specific changes in the repository. It is intentionally separated from the explanation section, since code generation and explanations have different technical limitations. Code generation is an invasive operation: It can edit files, create new files,

perform internal searches, and change the status of the project. The generation of explanations, on the contrary, should take into account the code and create structured descriptions without changing the implementation, unless the user explicitly uses the tool to insert comments. Separating these two tasks makes the user interface clearer and also makes it easier to interpret the evaluation results.

The main design decision was to use Junie as an external agent instead of re-implementing the agent inside the plugin. It was a pragmatic choice. The agent needs to check a lot of files, monitor the import, analyze the test results and decide which part of the repository should be changed. Junie already provides this tool usage cycle. If a different agent is used in a future version of the system, the replacement should mainly affect the service that runs the external command and analyzes its logs, rather than the data display model or interaction with the editor.

The user-oriented workflow starts on the Code Generation tab. The tab contains a task entry area, a create button, a streaming log area, and controls for creating or inserting explanations after generation. The request is intentionally transmitted to the agent in the form in which it was written by the user. The plugin does not rewrite the task into a hidden instruction format, as such rewriting would make the evaluation less transparent.

The same tab also configures how the generated changes will be explained after the agent run. Figure 3.1 shows the code-generation tab in both provider modes. In OpenAI mode, the interface exposes the model selector in addition to detail and format controls. In Junie mode, the explanation provider is selected directly while the generation request remains the same.

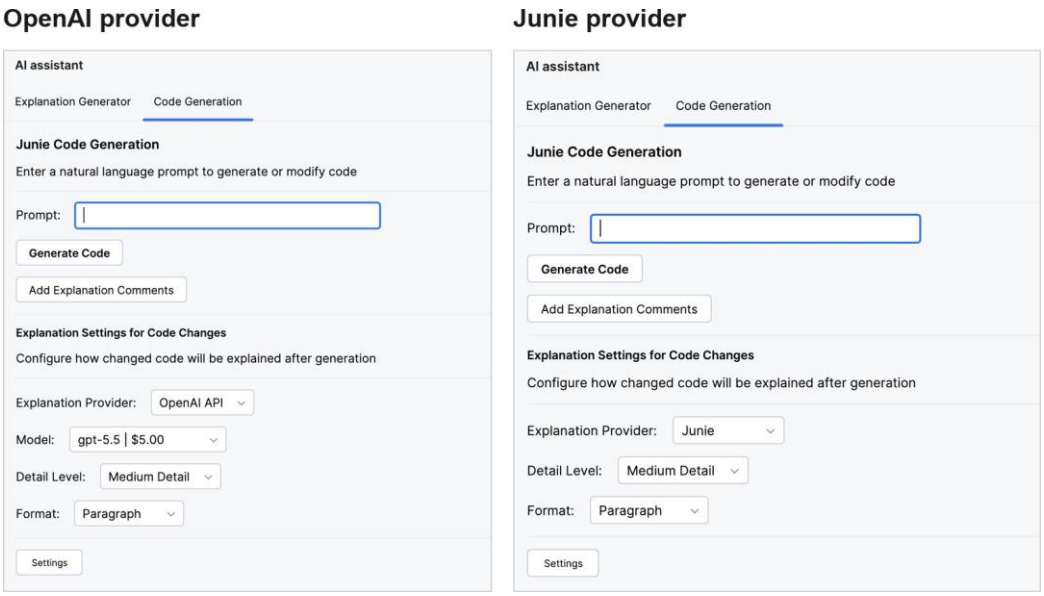

Figure 3.1: Code-generation tab settings for OpenAI and Junie explanation providers.

Before launching the external agent, the plugin creates a snapshot of the files. This step is important because the explanation stage should describe the code that actually changed. The implemented design saves the code before generation and later compares it with the code created after generation. The resulting modified segments contain the old code, the new code, the row ranges, and the type of change. These segments become compact explanatory blocks. They also reduce the use of tokens, as an explanatory request can focus on the region being edited and an adjustable amount of contextual information (only some files, full repo, augmented by an agent files, or without any).

The agent's log, shown in Figure 3.2, is used for two reasons. Firstly, it is useful for the user. The user can see which attempts the external agent has made, which files it has checked, and where the launch may have failed. Secondly, it improves the context of the explanation. This does not automatically make the explanation correct, but it does make it more informative than if it used only the differences.

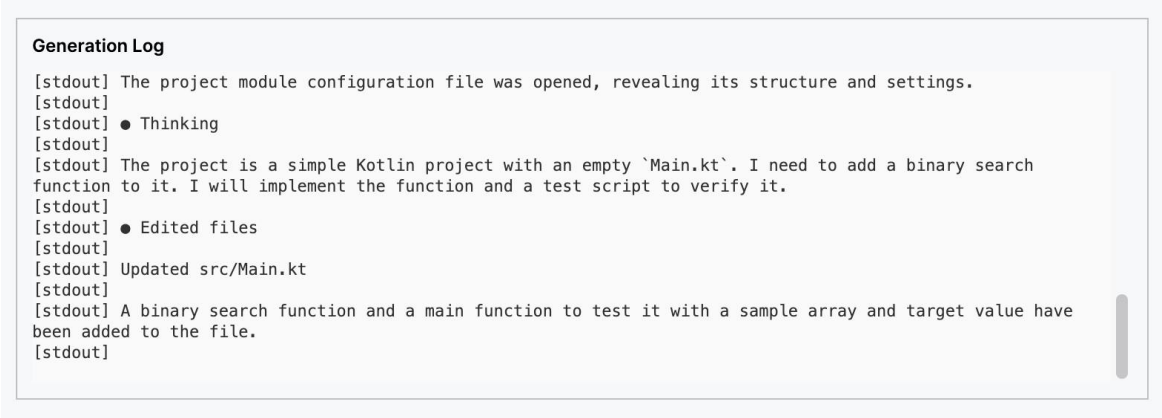

Figure 3.2: Junie generation log for the prompt `add function for binary search to Main.kt`.

Another important detail is that in the generation section, explanations are not inserted into the code immediately. The first result is an explanation artifact displayed in the tool window. The user can view it, change the level of detail, click on the matched phrases and only then insert comments if they are useful.

3.5 Explanation Data Model

The explanation schema follows the NaturalEdit reference but is implemented in Kotlin data classes. The core explanation object contains a title and six explanation variants: `low_unstructured`, `low_structured`, `medium_unstructured`, `medium_structured`, `high_unstructured`, and `high_structured`. This creates two independent controls for the user: detail level and format. Low detail gives a compact

overview, medium detail gives a practical explanation, and high detail gives the most implementation-specific description. The structured variants use bullet points, while the unstructured variants use paragraph style.

Mappings are represented by dedicated mapping objects. Each mapping contains an explanationComponent, which must be an exact substring of the displayed explanation, and a list of CodeSegment objects. Each code segment contains a code fragment and an absolute line number. A mapping container stores mappings for all six explanation variants. Finally, the combined explanation-with-mappings artifact joins the explanation and its mappings into one object.

3.6 OpenAI-Based Explanation Generation

The OpenAI explanation generation is based on OpenAIService. The service reads the endpoint, model name, temperature, token limit, and API key from the plugin settings. The API key is extracted from the secure password store of IntelliJ, while the unclassified configuration is saved in the plugin state. This separation is important because the plugin is an IDE extension, not a short-lived script. Users can keep the plugin installed for a long time, and credentials should not be written to repository files or logs. On the same settings page, the user can also select an explanation provider. In practice, this allows you to configure the explanation subsystem without changing the code.

The OpenAI provider exposes the model choice directly in the plugin UI. The implemented list is stored in MyToolWindow as a model-pricing map and is used by the model combo boxes in both the selected-code explanation tab and the code-generation tab. The available OpenAI models are gpt-5.5, gpt-5.4, gpt-5.4-mini, gpt-4.1, gpt-4.1-mini, gpt-4.1-nano, o3, gpt-4o, and gpt-4o-mini. For generated-code explanations, the OpenAI request also includes the filtered agent trace as part of the context.

The six-option explanation format was chosen to support different reading situations. An unstructured explanation with a low level of detail is useful when the user wants to get a quick answer only to the question ”what does this code do?”. A structured explanation with a low level of detail is useful for scanning. Medium granularity increases the responsibility for implementation and data flow, while high granularity allows you to describe algorithms, conditions, edge cases, and interactions between functions. Structured options are especially important for display, as marker-like components provide natural units that can be connected to the code. Unstructured options still persist because prose can be easier to read when the user wants a coherent narrative rather than a checklist.

The matching request is more rigorous than the usual explanation request. The model is asked to select explanation components that are exact substrings of the displayed explanation. For each component, it should return one or more code segments with the code text and line numbers. This exact matching requirement is important for an interactive user interface. The plugin highlights explanatory phrases by viewing the displayed explanatory text. If the returned component is just a paraphrase, the user interface cannot reliably know which phrase should be highlighted in color. Thus, the strict display format binds the output of the model to a specific rendering behavior.

The implementation includes several response correction mechanisms, since language models often generate almost correct JSON rather than a valid JSON file. The service removes the limitations of markdown, searches for a top-level object, allows additional surrounding text, and tries to decode the result into Kotlin serialization data classes. After decryption, the plugin performs a semantic check. It checks whether the components of the explanation are displayed in the selected explanation option. It checks whether the line numbers match the selected range of codes. It corrects the relative line numbers when it seems that the model is counted from the beginning of the fragment, and not from the beginning of the file. It also performs a tolerant code search when the exact code fragment cannot be found in the claimed string. These steps do not allow you to completely hide model errors, but they prevent small formatting deviations that can lead to the destruction of a useful artifact.

3.7 Junie-Based Explanation Generation

3.7.1 Purpose of the Junie Explanation Provider

The Junie explanation provider connects repository-level code generation with structured review artifacts. After an agent run, the plugin detects the changed segments and sends the relevant context to the provider: the old code, the new code, the user request, and the filtered agent trace. Thus, the Junie provider also receives the agent trace as part of the explanation context. The purpose is not to create generic documentation, but to explain the concrete implementation that was produced by the agent.

In the current implementation, the provider uses `gemini-3.1-flash-lite-preview` for the explanation step. The model is asked to generate one combined JSON artifact that contains both `explanation` and `mappings`. This unified generation is important because mapping components must be exact substrings of the displayed explanation. When the same model response creates the text and the line-level grounding together, the artifact is more internally consistent than when the explanation and map-

ping are produced as unrelated calls.

The high-level flow of the Junie CLI service is shown in Figure 3.3. The original repository is modified by the external AI agent, the plugin then derives a compact diff and filtered agent trace from the original and updated repositories, and the explanation provider turns this context into explanations with line-level mappings.

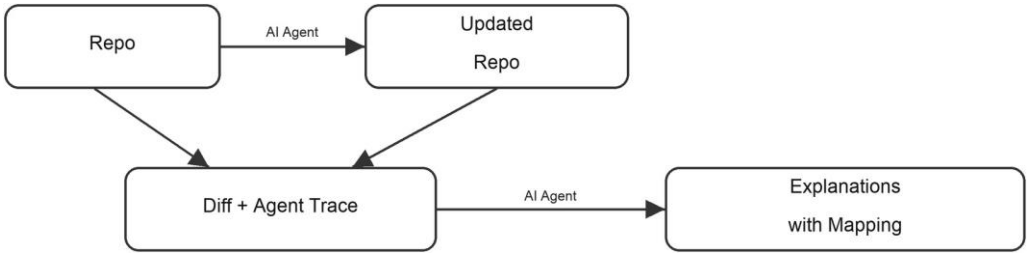

Figure 3.3: High-level architecture of the Junie CLI service workflow.

3.7.2 Input Context and Prompt Construction

The Junie explanation prompt is built from several pieces of context. For selected-code explanation, the most important input is the selected source code and its real line range in the file. For generated-code explanation, the prompt additionally contains the old code, the new code, the change type, the changed line range, the original user request, and the filtered agent trace. This richer prompt is necessary because a diff alone can be ambiguous. A small code edit may represent a bug fix, a compatibility change, a performance improvement, or a defensive check. The agent trace and user request help the explanation provider choose the interpretation that matches the actual generation task.

The prompt asks Junie to produce an artifact that follows the same schema as the rest of the plugin. The explanation must contain six variants: low-detail unstructured, low-detail structured, medium-detail unstructured, medium-detail structured, high-detail unstructured, and high-detail structured. The mappings must connect explanation components to concrete code segments and line numbers. The explanation component is required to be an exact substring of the explanation text displayed to the user. This requirement is not cosmetic. The UI highlights explanation components by locating them inside the rendered explanation. If Junie returns a paraphrased component, the plugin cannot reliably color it or connect it to a click action.

The prompt also instructs Junie to avoid unsupported speculation. The explanation should describe the provided code and the observed change, not invent hidden requirements or claim that tests passed unless that information is present in the con-

text. This constraint is especially important for generated code. An explanation that overstates correctness can reduce trust and slow review. The goal is to make code easier to inspect, not to certify that the patch is correct.

3.8 Interactive Mapping and Code Highlighting

The tool window displays the selected explanation according to the current detail controls and format. If comparisons are available, the explanation text is displayed as an interactive text area. Each mapped component of the explanation gets a background color from the overall palette. When the user clicks on the highlighted explanation phrase, the plugin opens the corresponding file, finds the matched code fragments, highlights their ranges in the editor and proceeds to the first matched line.

Figures 3.4 and 3.5 show the generated mapping artifact for the same Main.kt binary-search example in two display formats. The bullet-point view in Figure 3.4 separates the explanation into scan-friendly mapped components, while the paragraph view in Figure 3.5 keeps the explanation as a continuous text. In both layouts, the explanation phrase on the right is highlighted with the same color as the corresponding lines in the editor, making the natural-language explanation act as a navigational layer over the generated code.

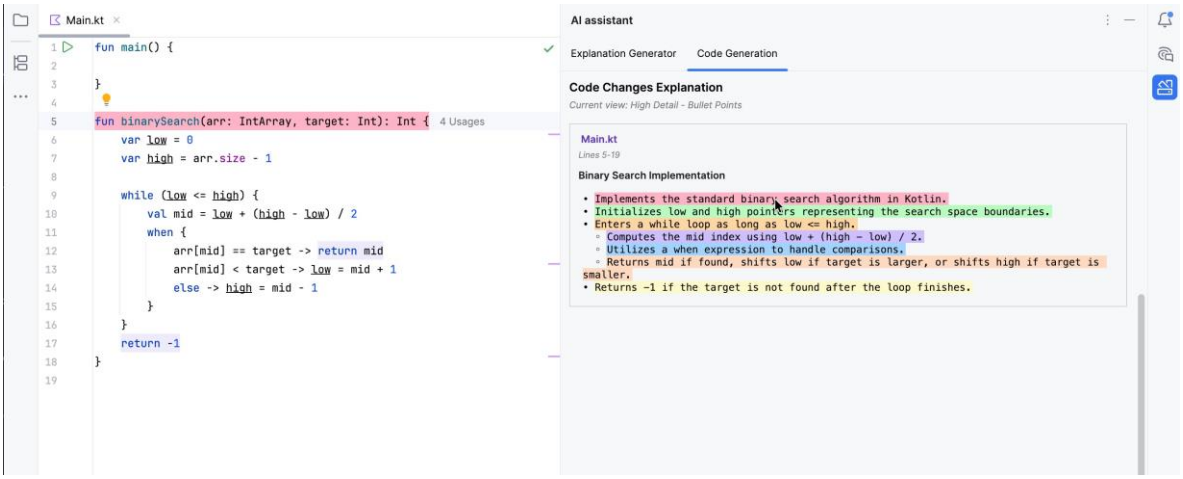

Figure 3.4: Interactive mapping for the generated binary-search explanation in bullet-point format.

3.9 Explanation Comments

The plugin can turn explanation mappings into source-code comments. For selected code, the Add Explanation Comments action generates a high-detail structured explanation and inserts mapped explanation components before their corresponding

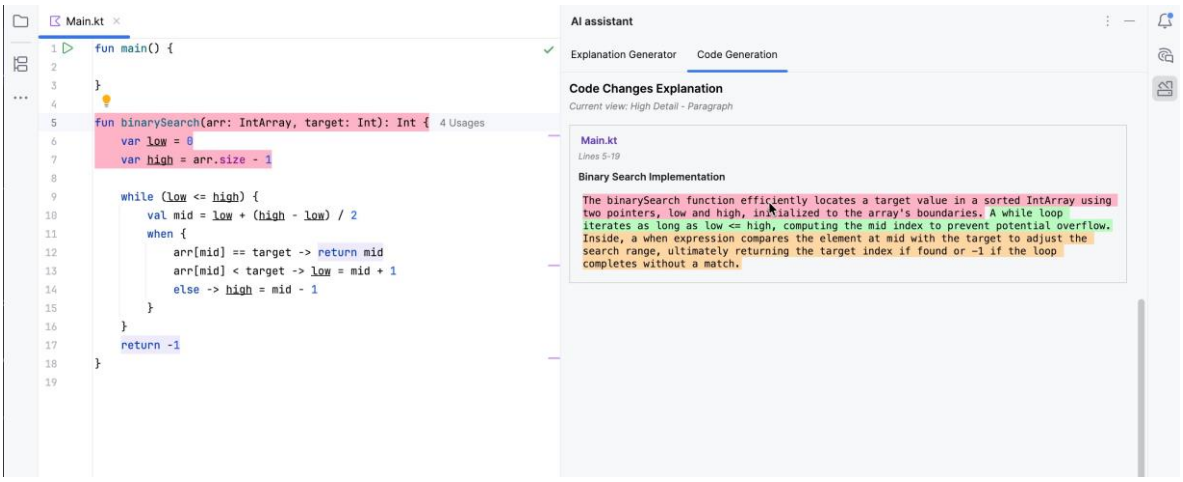

Figure 3.5: Interactive mapping for the generated binary-search explanation in paragraph format.

code chunks. For generated code, the Code Generation tab can insert comments for all explained change summaries.

ExplanationCommentInserter determines the comment style using IntelliJ language commenters. If a language-specific line comment prefix is available, it uses line comments. Otherwise, it falls back to block comments, with special handling for HTML, XML, and SVG files. The inserter groups multiple explanation components that target the same line and inserts comments in descending line order. This prevents earlier insertions from invalidating later offsets. After insertion, the plugin shifts the stored mapping line numbers so that interactive highlighting still points to the correct code after comments have been added.

3.10 Storage and Configuration

Credentials are not stored in plain configuration files. The plugin stores OpenAI and Junie tokens through IntelliJ's PasswordSafe API. General settings, such as endpoint, model, temperature, maximum tokens, and the selected explanation provider, are stored in the plugin state file. The default explanation provider is Junie, while the OpenAI API remains available as a legacy and controlled provider.

4. Evaluation and Results

4.1 Parallel Dataset and Comparison Metric

The first assessment required a dataset that would measure the quality of the explanations, not just the presence of comments next to the code. This distinction is important for the project. Many of the available code-comment datasets contain brief embedded comments, instructions for action, or descriptions at a superficial level. Such comments are useful for some documentation tasks, but they do not provide a reasonable, user-friendly explanation of what this system is aimed to create. The required dataset should contain the code along with human-written explanations that describe the conceptual role of the code snippets and can serve as a reference for comparing the generated explanations.

Finding such a dataset was a separate research task. After reviewing the possible code-comment and code-documentation datasets, I chose the labml.ai annotated deep learning paper implementations repository [14]. The repository is a collection of implementations of PyTorch neural networks and related machine learning methods. This format is extremely convenient for evaluating explanations: the same artifact contains executable code and detailed explanatory text designed to teach readers how the algorithm works. For this dissertation, this is more useful than the usual source code comments, since the reference text reflects a person's pedagogical understanding, rather than a brief hint at the implementation.

The resulting mini-dataset contains five representative labml.ai files: GAN losses, LSTM, Adam optimizer, batch normalization, and knowledge distillation. For each file, the original human-written explanatory components were saved as a reference. During generation, these explanations were removed from the input data, resulting in clean source code for the system. The system then generated a structured explanation with a high degree of detail and a mapping of the components of the explanation to the code segments. During evaluation, this generated artifact is compared with the original human reference. The reference and the generated mappings do not have to have exactly the same boundaries. Instead, the comparison checks whether the generated explanation retains the same logical decomposition, covers the same important concepts, and explains the code with comparable accuracy and depth.

The LLM-as-a-judge score is used as a metric. The local judge script uses o3 as the evaluation model. OpenAI describes o3 as a reasoning model for complex tasks, and its reasoning-oriented behavior is useful here because the judge must compare two structured explanations, reason about code coverage, and assign a calibrated quality score [15]. For each file, the judge receives generated components and expected components written by a human. The prompt suggests evaluating three aspects: segmentation, the quality of explanations, and the coverage of important concepts. It should return a JSON object with a numeric value metric from 0.0 to 1.0 and a brief text justification. In the rubric, values of 0.0–0.1 are rated as awful, 0.1–0.3 as poor, 0.4–0.6 as acceptable, 0.7–0.9 as good, and 1.0 as equivalent in quality to the reference.

This metric is noisy and should not be interpreted as a substitute for human evaluation. The judge may prefer a certain style of wording, may reward a certain degree of detail in segmentation, and may overlook minor algorithmic omissions. However, this provides a useful first quantitative signal. This allows the experimenter to compare the generated explanations with the same human reference, rather than evaluating them without a target. This is why the first part of the results uses the LLM-as-a-judge score as an approximate but informative indicator of the quality of the explanation.

The two generation settings were evaluated separately. The OpenAI service setting generated explanations using OpenAI models using the structured high-detail format. The Junie service setting used Junie generation with the structured high-detail format also; in this case, gemini-3.1-flash-lite was used. This distinction is important because the result of Junie is not just a choice of a different model.

Table 4.1 reports the evaluation results for each of the five algorithms. The results show that the complex files are not the same for each method. gpt-4o-mini does not work well with LSTM and batch normalization, while gpt-4.1 works best with Adam and remains stable in most topics. Junie successfully handles LSTM, Adam, batch normalization, and distillation, although their explanations are usually coarser than humans'.

Topic	Reference components	gpt-4o-mini	gpt-4.1-mini	gpt-4.1	Junie (gemini-3.1-flash-lite)
GAN losses	8	0.55	0.78	0.65	0.65
LSTM	23	0.25	0.83	0.83	0.78
Adam optimizer	30	0.50	0.50	0.85	0.75
Batch normalization	19	0.28	0.55	0.80	0.78
Knowledge distillation	47	0.50	0.82	0.80	0.80

Table 4.1: LLM-as-a-judge scores for each labml.ai mini-batch topic.

Table 4.2 shows the same results for all five topics. gpt-4.1 scores the highest average score. Junie is slightly behind with an average score of 0.752 and the lowest standard deviation among the evaluated methods. gpt-4.1-mini remains competitive, while gpt-4o-mini is clearly inferior in this dataset. The average number of reference

components was 25.4 per file, so all the generated outputs are coarser than human-generated annotations. The judges' explanations show that such a coarser segmentation is acceptable when the generated components still correspond to the basic structure of the code, but the score decreases if the explanations lack mathematical motivation, algorithmic rationale, or detailed conceptual background.

Service	Method	Mean	Median	Min	Max	Std.	Avg. generated components	Avg. reference components
OpenAI service	gpt-4o-mini	0.416	0.500	0.250	0.550	0.140	10.2	25.4
OpenAI service	gpt-4.1-mini	0.696	0.780	0.500	0.830	0.158	19.6	25.4
OpenAI service	gpt-4.1	0.786	0.800	0.650	0.850	0.079	16.6	25.4
Junie service	Junie (gemini-3.1-flash-lite)	0.752	0.780	0.650	0.800	0.060	8.4	25.4

Table 4.2: LLM-as-a-judge statistics for each labml.ai mini-batch topic.

4.2 SWE-bench Verified Mini Evaluation Methodology

4.2.1 SWE-bench Benchmark

SWE-bench is a benchmark to assess whether agents with a language model can solve real-world software development problems. The original benchmark contains 2294 issue-resolution tasks compiled from real-world issues on GitHub and pull requests from 12 popular Python repositories [16]. Each task provides the agent with information about the repository state and a description of the problem. The expected result is a patch that will change the repository so that the problem is fixed. This makes SWE-bench more realistic than isolated code generation benchmarks, since solving the problem may require navigating the repository, analyzing multiple files, editing existing code, and verifying fixes using project tests.

In a typical SWE-bench workflow, the agent is provided with a repository at a specific commit and problem_statement. It is assumed that the agent should not see the gold solution patch. It checks the codebase, edits the files, and sends the generated patch. The evaluator then applies this patch in a controlled environment and runs tests related to a specific task. The official SWE-bench evaluation guide describes evaluation as applying generated patches to real repositories and running repository tests in a Docker environment to ensure reproducibility [19]. The prediction sent to the official harness is a JSONL record containing the task instance_id, the model name, and the generated model_patch.

Table 4.3 shows the main task fields used in this dissertation. Some fields are part of the agent's input data, while others are for evaluation purposes only. This distinction is important because using patch or test_patch directly during repair will leak the answer or the evaluation criteria.

The main metric of SWE-bench is whether the problem has been solved. The task is considered resolved when the generated patch is successfully applied and passes

Field	Role in evaluation
instance_id	Unique task identifier used to compare regular and explained runs and to report results.
repo	GitHub repository containing the problem.
base_commit	Repository revision used as the starting point for the task.
problem_statement	Problem text sent to the agent as a repair prompt.
patch	Gold developer patch. It is used as reference metadata and should not be shared with the repair agent.
test_patch	Official evaluation-test patch applied after the agent completes its work.
FAIL_TO_PASS	Tests that should fail before the fix and pass after the correct fix.
PASS_TO_PASS	Regression tests that should continue to pass after the generated patch.

Table 4.3: Main SWE-bench task fields and their role in evaluation.

the official evaluation tests. The official evaluation results report the number of submitted instances, completed instances, resolved instances, and the resolution rate [19]. This binary signal, resolved/not-resolved, is more rigorous than checking whether the agent has created syntactically plausible code: the fix must satisfy the tests related to the real problem.

SWE-bench Verified is a specially curated subset of SWE-bench. OpenAI and the authors of SWE-bench introduced it after they found that some of the original SWE-bench tasks were insufficiently specified or contained tests that could unfairly reject acceptable solutions [17]. The Verified construction used a human annotation campaign with professional Python developers. OpenAI reports that 93 developers annotated 1699 random SWE-bench samples with three independent annotators for each sample. The samples were filtered out when the problem description was too incomplete, when FAIL_TO_PASS tests unfairly excluded acceptable solutions, or when annotators found other serious problems. The final SWE-bench Verified subset contains 500 expert-reviewed tasks and is listed in the official SWE-bench dataset guide as a high-quality evaluation option [18].

4.2.2 SWE-bench Verified Mini Benchmark

This dissertation uses SWE-bench Verified Mini, published as a Hugging Face dataset MariusHobbhahn/swe-bench-verified-mini [20]. The source code for creating the mini dataset is available in the public GitHub repository mariushobbhahn/SWEBench-verified-mini [21]. The mini dataset contains 50 tasks instead of the 500 tasks in SWE-bench Verified. Its goal is to reduce the cost and simplify conducting experiments, while maintaining approximately the same distribution of performance, test pass rates, and complexity as in the full Verified set. The dataset card contains

a test split of 50 rows with fields such as repo, instance_id, base_commit, patch, test_patch, problem_statement, FAIL_TO_PASS, and PASS_TO_PASS.

The GitHub repository explains how the mini subset was selected. The author compared several possible subsets of 50 tasks: the full 500-task set, a representative subset with k-means, a random subset, and a size-optimized subset. The ultimate goal was to select 50 data points whose score distribution remained close to the full Verified dataset while reducing storage requirements. During the construction, data points are first clustered using k-means, and then linear programming is used to select tasks that minimize storage size while keeping important parameters such as performance, number of tests passed, and complexity close to the full benchmark [21].

The local copy used in this experiment contains all 50 mini tasks. It is balanced between two repositories: 25 tasks from django/django and 25 tasks from sphinx-doc/sphinx.

4.2.3 Paired Evaluation Protocol

The experiment compares the behavior of an agent in two conditions: in a regular repository and in a repository supplemented with explanations. Both conditions use the same SWE-bench task, the same instance_id, the same base_commit, the same problem_statement, and the same official evaluation tests. The only intended difference is whether the repository contains generated explanation comments before the repair agent starts working.

The regular condition is run from a repository with clean source code copied from repos. The explained condition is run from the corresponding copy of the repository in the repos-explained section. Before launching the repair run, this explained repository contains comments derived from high-detailed structured explanations. For each task, the explanation-generation pipeline selects the files related to the task, generates explanations, maps the explanation components to the source-code ranges, and inserts comments before the mapped code regions. In the controlled explained repositories, explanations are generated for files that appear in the gold patch and for 10 additional randomly selected files. This reduces the emphasis on only the known target files while avoiding the cost of generating explanations for the entire repository. The repair agent is then run in this modified repository, so explanations are available in the normal context of the source code while the agent is reviewing the files.

The official gold patch is not provided to Junie during the repair phase. It is used offline only to prepare a repository enriched with explanations, mainly to select files related to the task in this controlled experiment. This project checks whether explanations next to the corresponding code can help in agentic repair. It should not

be interpreted as a search strategy in a production environment. In the workflow of a production plugin or agent, the relevant files must be selected through retrieval, static analysis, user selection, or an earlier agent pass.

For each task, run_junie_swebench.py performs the following procedure:

1. Copy the source repository to an isolated temporary worktree.
2. Check out the task's base_commit, unless you are using an already prepared repository with explanations.
3. Create a local baseline commit if comments are already included in the repository with explanations.
4. Run Junie with a prompt based on SWE-bench problem_statement.
5. Save Junie's stdout, stderr, generated patch, return code, timing information, and working directory artifacts.
6. Restore agent edits under tests/ and restore files affected by the official test_patch.
7. Apply the official test_patch.
8. Resolve framework-specific test labels for Django and Sphinx.
9. Run the FAIL_TO_PASS tests and, if they are not disabled, the PASS_TO_PASS tests.
10. Save a text file result.json for each task, then summarize the result files into CSV comparison tables.

The main metric is the task success value: whether the generated patch passes the evaluation command. A secondary process metric is the number of visible Junie actions. This number of actions is not an ideal indicator of internal reasoning, but it is constantly displayed in the execution logs and gives an approximate idea of the agent's observed effort. Therefore, two questions arise when comparing. Firstly, does the explained repository solve more, fewer, or the same number of tasks as a regular repository? Secondly, does it change the number of visible actions that the agent performs when completing tasks? The paired design controls the identity of the task and the official tests, but it does not fully control the non-determinism of the external agent. Therefore, the comparison should be interpreted as an empirical mini-benchmark experiment rather than as a statistically reliable statement. The quantitative results for these questions are reported later in the *Quality and Agent Performance* section.

4.3 Plugin UI

The result of the implementation is a working IntelliJ IDEA plugin. The plugin provides a tool window where the user can generate code using Junie, view the generation logs, and then view explanations of the modified code. The presentation of explanations supports multiple levels of detail and both paragraph and structured format. For each explanation displayed, the plugin saves explicit mappings of the explanation components to the source-code lines.

In addition to explanations of generated code, the plugin includes an explanation-only workflow. The separate Explanation Generation tab allows the user to manually select a fragment of source code in the editor and press the explain action. In this mode, no coding task is launched and the agent tracer is not used, because the agent does not produce a patch. The explanation provider instead relies on the selected code, the repository context available to the plugin, and the surrounding source-code information to describe what the selected fragment does.

The user interface makes the mapping artifact interactive. When the user selects an explanatory component, the corresponding fragment of the source code is highlighted in the editor. The user can also insert explanatory comments into the source file. After insertion, the plugin shifts the saved line numbers so that the selection matches the edited file. This demonstrates that the presentation of explanations is not only an artifact of evaluation; it can be used in the development process.

The plugin also supports two explanation providers. The OpenAI path is useful for controlled model comparison, while the Junie path is closer to the repository-aware workflow. Settings for selecting a provider, model, endpoint, and credentials are available through the plugin configuration user interface and are stored in the IntelliJ platform services. Thus, a working user interface is one of the results of the project: it turns the generated explanations and mappings into an easy-to-view interface for developers.

4.4 Quality and Agent Performance

The generated SWE-bench artifacts contain 50 task directories. For this experiment, explanation artifacts were generated by the Junie explanation provider using gemini-3.1-flash-lite-preview, which corresponds to Gemini 3.1 Flash-Lite. Code generation in both conditions, the regular repositories and the explanation-enriched repositories, was performed by Junie using its default gemini-3-flash-preview model alias, corresponding to Gemini 3 Flash. They are stored under repos-explained. Across these tasks, the generation pipeline created 548 explained files and 3554 high-

detail mappings. The artifacts are distributed across two repositories: Django tasks contain 277 explained files and 1802 mappings, while Sphinx tasks contain 271 explained files and 1752 mappings. Table 4.4 reports this artifact coverage by repository.

Repository	Tasks	Explained files	Total mappings	Avg. mappings/task
django/django	25	277	1802	72.08
sphinx-doc/sphinx	25	271	1752	70.08
Combined	50	548	3554	71.08

Table 4.4: Explanation artifacts generated for SWE-bench Verified Mini tasks.

The generated repositories contain explanations and mappings for several files for each task, which makes the artifacts suitable for checking whether the agent benefits from the additional context of the code. At the same time, the artifacts themselves do not prove the correctness of the explanation; they only show the coverage of the generation. Correctness still depends on the quality of the mapping and whether the explanation helps in subsequent code generation.

The final results compare regular Junie runs and runs with additional explanations. Table 4.5 reports the pass rates.

Dataset	Regular solved	Explained solved	Regular rate	Explained rate
Django, 25 tasks	20	20	80%	80%
Sphinx, 25 tasks	12	13	48%	52%
Combined, 50 tasks	32	33	64%	66%

Table 4.5: SWE-bench Verified Mini pass rates for regular and explanation-enriched runs.

For Django, the same number of tasks were completed in both conditions: 20 out of 25. The result did not change: every Django task that passed in the regular condition also passed in the explained condition, and every failed task remained failed. For Sphinx, the explained condition solved one additional task. The modified task was sphinx-doc__sphinx-8638, which failed in a regular run and was completed in an explained run. Thus, across all 50 tasks, one additional success was achieved in the explained condition.

Table 4.6 shows secondary process indicators. The signal about the number of actions is ambiguous, but favorable for the condition with additional explanations. The average number of visible agent actions in the summary data remained virtually unchanged: 47.76 actions for regular runs and 47.80 for runs with additional explanations. However, the median number of actions decreases from 45.5 to 42.5. This means that the explained repositories solved one additional task without requiring a significant increase in the average observed effort and with fewer typical actions. For Django, the average number of actions in the explained condition is slightly lower. For

Sphinx, the average number of actions is slightly higher, but the median number of actions is lower. Timing data is less stable because the launch of an external agent can be affected by service latency, retries, local machine load, and rerun history. For this reason, pass rate and action count are considered as the main quantitative indicators in this experiment.

Dataset	Reg. mean actions	Expl. mean actions	Reg. median actions	Expl. median actions
Django	38.52	38.36	36.0	38.0
Sphinx	57.00	57.24	54.0	53.0
Combined	47.76	47.80	45.5	42.5

Table 4.6: Agent action counts in SWE-bench runs.

5. Discussion

The main practical result is that explanation-enriched repositories are suitable for solving SWE-bench-style tasks. The generated artifacts cover hundreds of files and thousands of mappings, and the performance of complex problem solving remains comparable to the regular baseline. The additional success of Sphinx alone is not enough to talk about a statistically significant improvement, but it indicates that mapped explanations can provide useful context without systematically reducing agent performance.

The current system is the most effective as a verification and inspection layer. The user can ask the assistant to generate the code, view the modified segments, read the explanations at different levels of detail, click on the explanatory phrases to see the corresponding code, and if desired, insert the explanation into the repository. This makes the agent's output more easily verifiable than a simple patch. The system is less effective as an autonomous means of improving productivity, since the amount of evaluation is small, and the quality of explanations strongly depends on the model and the reliability of the syntax analysis.

6. Conclusion

6.1 Key points

The key achievements of the work can be summarized as follows:

- A working IntelliJ IDEA plugin was implemented as a developer-facing tool for agentic code generation, explanation generation, line-level explanation-to-code mappings, interactive highlighting, and insertion of mapped comments into source files.
- The system supports two types of explanation generation: OpenAI-based generation for controlled model comparison and Junie-based generation for repository-aware explanations after agentic code changes.
- The explanation data model supports six explanation variants: three levels of detail combined with structured and unstructured presentation formats. This makes the same generated artifact usable both as a short overview and as a detailed review aid.
- A labml.ai-based evaluation methodology was developed for assessing explanation quality. The dataset was selected because it contains source code paired with detailed human-written pedagogical explanations. The generated high-detail structured explanations were compared with these references using an o3-based LLM-as-a-judge rubric that scores segmentation, explanation quality, and coverage of important concepts. This gives a repeatable quantitative signal while still preserving a qualitative justification for each score.
- A custom paired mini-benchmark methodology was developed on top of SWE-bench Verified Mini. Each task was run in two conditions: a regular repository and an explanation-enriched repository. The paired design kept the same instance_id, base_commit, problem_statement, and official evaluation tests, making it possible to compare pass rate and visible Junie action counts while keeping the task identity fixed.

- Using the labml.ai methodology, five annotated deep learning implementations were evaluated against human-written references. The best OpenAI configuration, gpt-4.1, reached a mean LLM-as-a-judge score of 0.786. The Junie-based setting reached a mean score of 0.752 and had the lowest standard deviation, 0.060, among the evaluated methods.
- For SWE-bench Verified Mini, the explanation-generation pipeline prepared artifacts for all 50 tasks. Across these tasks, it produced 548 explained files and 3554 high-detail mappings.
- In the paired SWE-bench experiment, the regular repositories solved 32 out of 50 tasks, or 64%. The explanation-enriched repositories solved 33 out of 50 tasks, or 66%. The difference came from one additional solved Sphinx task, sphinx-doc__sphinx-8638, while Django remained unchanged at 20 out of 25 tasks in both conditions.
- The explanation-enriched condition did not introduce a measured regression in the collected benchmark results. The average number of visible Junie actions stayed almost unchanged, from 47.76 in regular runs to 47.80 in explained runs, while the median decreased from 45.5 to 42.5 actions.

6.2 Limitations of the study

The study is limited by the size of the dataset, provider variability, evaluation noise, and the cost of the experiment. The parallel explanation-quality dataset contains only five files, so it is useful for targeted comparison, but not for broad generalization. The LLM-as-a-judge metric is also imperfect, as it can be sensitive to the wording, segmentation, and preferences of a particular model. The SWE-bench experiment is larger, but still limited to 50 tasks and two repositories. Performing larger-scale evaluations is expensive because the pipeline uses a lot of tokens in several stages: generating explanations, creating mappings, and evaluating the LLM as a judge. These stages also require paid model calls and lengthy launches by external agents. For an unfunded dissertation project, this creates real practical difficulties: a full-scale evaluation in many repositories, in many model configurations and repeated runs would require significantly more money and computing budget than was available. Time measurement is subject to noise, as agent startup may be affected by external service delays, local machine load, retries, and restart history. Finally, the explained SWE-bench condition allows comments to be inserted into repositories. This is useful so that explanations are visible to the command-line agent, but it is not identical to the explanation layer used only in the IDE. Another limitation is the lack of statistically

significant human-subject studies. The most direct way to assess whether explanations help people is to collect feedback from domain experts and plugin users. Such a study would require the involvement of participants, the development of controlled tasks or interviews, the payment of remuneration to participants, and the investment of time in qualitative and quantitative analysis. These requirements went beyond the time and financial constraints associated with the current work.

6.3 Suggestions for future research

Future work should evaluate the system based on the full SWE-bench Verified dataset and additional repositories, when this is made possible by financing and compute budget. The parallel explanation-quality dataset should be expanded with more files, more algorithm families, more programming domains, and human expert assessments. Larger datasets would allow us to check whether the observed results correspond to the five examples labml.ai and a mini-subset of SWE-bench, consisting of 50 tasks. The plugin could support an explanation-only context that does not insert comments into source files, which allows for a more accurate comparison of visible IDE explanations and explanations that modify the repository. Another promising area is the use of mappings as retrieval units for multi-agent workflows, so that subsequent agents can extract not only code, but also grounded explanations for previous changes. In future experiments, different file selection strategies should also be compared, since for production use, one cannot rely on gold patch files when choosing targets for explanation. Finally, future research should include user studies involving developers and domain experts. This study should find out whether the plugin makes it easier to check the generated code, whether the mapped explanations help users find the relevant code faster, whether the inserted comments are useful or distracting, and whether the tool changes trust in patches generated by the agent.

The public source repository and the build commands needed for installation are provided in Appendices A and B.

Bibliographic References

[1] Tang, N. et al.: NaturalEdit: Code Modification through Direct Interaction with Adaptive Natural Language Representation. arXiv preprint arXiv:2510.04494 (2025). arXiv link

[2] Yang, D. et al.: DocAgent: A Multi-Agent System for Automated Code Documentation Generation. arXiv preprint arXiv:2504.08725 (2025). arXiv link

[3] Shi, K. et al.: Natural Language Outlines for Code: Literate Programming in the LLM Era. arXiv preprint arXiv:2408.04820 (2024). arXiv link

[4] Rosenfeld, A., Richardson, A.: Explainability in Human-Agent Systems. Autonomous Agents and Multi-Agent Systems 33, 673–705 (2019). arXiv link

[5] Miller, T.: Explanation in Artificial Intelligence: Insights from the Social Sciences. Artificial Intelligence 267, 1–38 (2019). DOI link

[6] Langer, M. et al.: What Do We Want From Explainable Artificial Intelligence (XAI)? A Stakeholder Perspective on XAI and a Conceptual Model Guiding Interdisciplinary XAI Research. Artificial Intelligence 299, 103473 (2021). arXiv link

[7] Alzetta, F., Giorgini, P., Najjar, A., Schumacher, M., Calvaresi, D.: In-time Explainability in Multi-Agent Systems: Challenges, Opportunities, and Roadmap. Lecture Notes in Computer Science 12175, 39–53 (2020). DOI link

[8] Sado, F., Loo, C. K., Liew, W. S., Kerzel, M., Wermter, S.: Explainable Goal-Driven Agents and Robots: A Comprehensive Review. ACM Computing Surveys 55(10), Article 211 (2023). arXiv link

[9] Wang, R., Cheng, R., Ford, D., Zimmermann, T.: Investigating and Designing for Trust in AI-powered Code Generation Tools. Proceedings of the ACM Conference on Fairness, Accountability, and Transparency (FAccT) (2024). arXiv link

[10] Chen, V., Talwalkar, A., Brennan, R., Neubig, G.: Code with Me or for Me? How Increasing AI Automation Transforms Developer Workflows. arXiv preprint arXiv:2507.08149 (2025). arXiv link

[11] Geng, M., Wang, S., Dong, D., Wang, H., Li, G., Jin, Z., Mao, X., Liao, X.: Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. Proceedings of the IEEE/ACM International Conference on Software Engineering (ICSE) (2024). arXiv link

[12] Chen, Y., Liu, Y., Meng, F., Chen, Y., Xu, J., Zhou, J.: Comments as Natural Logic Pivots: Improve Code Generation via Comment Perspective. Findings of the Association for Computational Linguistics: ACL (2024). arXiv link

[13] Macke, W., Doyle, M.: Testing the Effect of Code Documentation on Large Language Model Code Understanding. arXiv preprint arXiv:2404.03114 (2024). arXiv link

[14] labml.ai: Annotated Deep Learning Paper Implementations. GitHub repository. Online source

[15] OpenAI: o3 Model. OpenAI API Documentation. Online source

[16] Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., Narasimhan, K.: SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv preprint arXiv:2310.06770 (2024). arXiv link

[17] OpenAI: Introducing SWE-bench Verified. OpenAI Blog (2024). Online source

[18] SWE-bench: SWE-bench Datasets Guide. Online source

[19] SWE-bench: SWE-bench Evaluation Guide. Online source

[20] Hobbhahn, M.: MariusHobbhahn/swe-bench-verified-mini. Hugging Face Datasets (2025). Dataset card

[21] Hobbhahn, M.: SWEBench-verified-mini. GitHub repository. Online source

A. Plugin Source Code

Due to the large volume of our code base, we decided not to include the implementation directly in this paper. However, the full source code is publicly available on GitHub at the following link:

https://github.com/yule44ka/explainable-ai-plugin/

This repository contains all the relevant code, documentation, and instructions necessary to reproduce the experiments and results discussed in this paper.

B. Plugin Build for Installation

The plugin can be built from the repository root as follows:

```
cd explainable-ai-plugin/plugin/explainable-ai-plugin
./gradlew build
```

For local development and manual testing in an IntelliJ sandbox, the following command starts an IDE instance with the plugin loaded:

```
./gradlew runIde
```

The implementation requires JDK 21 or later. For code generation, Junie CLI must be installed and available in PATH; for OpenAI-based explanations, an OpenAI API key must be configured in Settings -> Tools -> Explainable AI.