

Conversational Assistants for Software Development: Integration, Traceability and Coordination

Albert Contreras^a, Esther Guerra^b and Juan de Lara^c

Computer Science Department, Universidad Autónoma de Madrid, Spain
{albert.contreras, esther.guerra, juan.delara}@uam.es

Keywords: Software Development, Development Assistant, Large Language Model, Conversational Agent, Chatbot, IDE, Eclipse, Java, Method Renaming.


Abstract: The recent advances in generative artificial intelligence are revolutionising our daily lives. Large language models (LLMs) – the technology underlying conversational agents like ChatGPT – can produce sensible text in response to user prompts, and so, they are being used to solve tasks in many disciplines like marketing, law, human resources or media content creation. Software development is also following this trend, with recent proposals for conversational assistants tailored for this domain. However, there is still a need to understand the possibilities of integrating these assistants within *integrated development environments* (IDEs), coordinating multiple assistants, and tracing their contributions to the software project under development. This paper tackles this gap by exploring alternatives for assistant integration within IDEs, and proposing a general architecture for conversational assistance in software development that comprises a rich traceability model of the user-assistant interaction, and a multi-assistant coordination model. We have realised our proposal building an assistant (named CARET) for Java development within Eclipse. The assistant supports tasks like code completion, documentation, maintenance, code comprehension and testing. We present an evaluation for one specific development task (method renaming), showing promising results.


1 INTRODUCTION


Software development has been striving for higher levels of productivity and quality from its inception. This goal has been pursued by several strategies, such as the use of higher-level development languages (Wasowski and Berger, 2023), automation techniques (Brambilla et al., 2017), powerful integrated development environments (IDEs like Eclipse¹, Visual Studio Code², or IntelliJ IDEA³), knowledge bases and FAQs documenting development expertise (Abdalkareem et al., 2017), catalogues of design patterns (Gamma et al., 1994), or recommenders and development assistants (Rich and Waters, 1988; Savary-Leblanc et al., 2023). In this paper, we are interested in the latter approaches.

The recent advances in deep learning, natural language processing and generative artificial intelligence

have triggered the appearance of *open domain* conversational agents able to produce sensible responses upon arbitrary user prompts. These agents, also called chatbots⁴, are currently being explored to solve all sorts of tasks in domains like marketing, law, human resources, media content creation, and software development, among many others. They are powered by large language models (LLMs), which are transformer-based networks trained on vast amounts of text data. Some specific LLMs for code exist (Xu et al., 2022a), such as Codex⁵, Code Llama⁶, and StarCoder (Li et al., 2023). Some of them are even integrated into IDEs, such as GitHub Copilot⁷. Still, assistant-based development is in its childhood, with many problems to solve and assistance strategies to assess (Ozkaya, 2023a). This way, researchers working on development assistance may wonder: *What are the possible ways to integrate assistants into IDEs?*

^a  <https://orcid.org/0009-0006-6887-9826>

^b  <https://orcid.org/0000-0002-2818-2278>

^c  <https://orcid.org/0000-0001-9425-6362>

¹ <https://www.eclipse.org/>

² <https://code.visualstudio.com/>

³ <https://www.jetbrains.com/idea/>

⁴ In this paper, we use the terms *conversational agent* and *chatbot* interchangeably.

⁵ <https://openai.com/blog/openai-codex>

⁶ <https://ai.meta.com/blog/code-llama-large-language-model-coding/>

⁷ <https://github.com/features/copilot>

Is it possible to retrieve past developer-assistant interactions? How can multiple assistants be coordinated? How can the assistant effectiveness be assessed?

This paper aims to answer the previous questions. For this purpose, we first present a taxonomy of the possibilities for integrating conversational assistance into IDEs, in the form of a feature diagram (Kang et al., 1990). Then, we propose a general and extensible architecture for conversational assistance in software development, able to coordinate the recommendations of several chatbots, not necessarily built using LLMs. The architecture reifies and persists the interactions between the developers and the assistant as a traceability model. This allows tracking the decisions made, as well as supporting queries about which parts of the code generated the assistant, why, when, and who invoked the assistant.

To validate these ideas, we present a specific conversational assistant for Java development within Eclipse called CARET (Conversational Assistant for softwaRE developmenT). The assistant helps in a wide range of development tasks, including code completion, documentation, maintenance, program comprehension and unit test generation. It features a bidirectional traceability model from reified user-agent interactions to code, and vice versa, via code annotations. We present an evaluation of the suitability of one of the development tasks (method renaming), which yields very promising results.

In the remainder of this paper, Section 2 provides background on chatbots and analyses the state of the art. Next, Section 3 describes the three main ingredients of our approach: the analysis of the assistant-IDE integration possibilities, the traceability model, and the coordination of multiple conversational agents into a unified assistant. Section 4 introduces CARET, our Java/Eclipse assistant. Then, Section 5 reports on the evaluation. Finally, Section 6 finishes with the conclusions and prospects for future work.

2 STATE OF THE ART

Next, we provide some background and state of the art on conversational agents (Section 2.1) and their use for software development tasks (Section 2.2).

2.1 Conversational agents

Conversational agents (or *chatbots*) are being increasingly used to access software services using natural language. Their popularity has risen because they reduce the entry level to services like customer support,

banking or shopping, and can be easily embedded into social networks (e.g., Telegram), websites or intelligent speakers. These chatbots are called *task-oriented* as they help users in performing a specific task.

Many technologies to build task-oriented chatbots exist (Pérez-Soler et al., 2021), like Google’s Dialogflow⁸, Amazon Lex⁹, Microsoft’s Bot Framework¹⁰, the IBM Watson Assistant¹¹ or Rasa¹². They allow defining the user *intents* that a chatbot aims at recognising (e.g., ordering a pizza, setting an appointment with a technician). Intents declare training phrases, which are used to train a natural language understanding engine. This way, when the user inputs an utterance, the engine selects the most likely intent with a certain confidence. If the confidence is below a threshold, then a *fallback* intent is selected, if available. Fallbacks are an indication of user requests that the chatbot cannot handle.

Intents may have *parameters*, which are pieces of information required from the user (e.g., type of pizza, appointment date), and whose value is extracted from the user utterance. When the chatbot detects an intent, it performs the actions associated to the intent, usually accessing an external information system and composing a response. Finally, the user-chatbot conversation *flows* are explicitly designed by setting *paths* of intents that a user may follow to perform a task.

Different from task-oriented chatbots, the recent advances in generative artificial intelligence have promoted the appearance of *open-domain* chatbots based on LLMs, like OpenAI’s ChatGPT¹³ or Google’s Gemini¹⁴ (formerly known as Bard). LLMs are large neural networks with a transformer-based architecture that are trained on vast amounts of textual data (Xu et al., 2022a). They are able to provide a sensible text output upon arbitrary user *prompts* without the need to predefine admissible user intents.

Rather than being task-specific, LLMs are typically open-domain, although some of them have been *fine-tuned* on specialised data, like code (Xu et al., 2022a). Fine-tuning enables repurposing an LLM pretrained on generic text data for specific downstream tasks (e.g., question-answering), or domains (e.g., programming). However, since LLMs have no fallbacks, it can be difficult to assess the accuracy of the produced output, or to assert when an LLM does

⁸<https://dialogflow.com/>

⁹<https://aws.amazon.com/en/lex/>

¹⁰<https://dev.botframework.com/>

¹¹<https://www.ibm.com/cloud/watson-assistant/>

¹²<https://rasa.com/>

¹³<https://openai.com/chatgpt>

¹⁴<https://gemini.google.com/>

not know the answer, leading to so-called *hallucinations* (Chen et al., 2023) (i.e., inaccurate or nonsensical answers presented as fact). Related to this issue, the *temperature* hyperparameter of LLMs is used to regulate their unpredictability. This way, the higher the temperature, the less predictable the LLM’s outputs become upon the same input.

2.2 Conversational assistance for software development

The idea of development assistants can be traced back to the *programmer’s apprentice* (Rich and Waters, 1988) in the 80’s. This system used symbolic artificial intelligence – knowledge representation based on frames – to describe and reason about programs with the help of design *clichés* (a.k.a. design patterns (Gamma et al., 1994)).

Today, the focus of artificial intelligence has shifted to machine learning. In particular, deep learning is being increasingly used to help software developers in tasks related to requirements, software design and modelling, coding, testing, and maintenance (Yang et al., 2022).

Recently, the advent of LLMs (Zhao et al., 2023) has prompted their use also for software engineering. Several LLM-based programming assistants have been proposed. One of the first ones was GitHub Copilot, originally built on Codex, an LLM based on GPT-3 and fine-tuned on code. Copilot is integrated in several IDEs such as Visual Studio Code and JetBrains, and offers autocompletion assistance as the developer types. Although Copilot was initially free, currently it is a paid feature. Its code completion capabilities have been recently integrated into the Eclipse IDE as a plugin¹⁵. This plugin provides autocompletion for several languages using GitHub Copilot at the back, and hence requiring a subscription. While these assistants are valuable for developers, a deeper integration with the IDE beyond code completion would be desirable. They also lack traceability information to understand which parts of the code created the assistant and why. Moreover, future assistant-enabled IDEs may need to coordinate several agents.

The programmer’s assistant (Ross et al., 2023) is a conversational assistant for Python based on Codex. It is interacted with using conversation, and the context can be provided by selecting code. A user study revealed the utility and good acceptance of this assistant by developers. However, the assistant is not integrated into a fully-fledged IDE, so it does not take

advantage of the possibilities of integration via commands, and traceability mechanisms are missing.

In (Xu et al., 2022b), the authors present two systems for code generation and retrieval from natural language, both integrated in the IDE PyCharm¹⁶ for Python programming. They evaluated the systems for improved efficiency and quality with mixed results, but developers declared enjoying the experience.

Barke *et al.* used *grounded theory* to study how programmers interact with GitHub Copilot (Barke et al., 2023). They detected two main usages of the assistant: for the *acceleration* of known tasks (i.e., autocompletion) and for the *exploration* of options that may be used as the starting point to reach a solution. We claim that exploration can be improved by the availability of several agents, and that the assistant contributions should be properly traced.

Robe and Kuttal explored design options for Pair-Buddy, a conversational assistant for pair programming, with a 3D embodiment (Robe and Kuttal, 2022). They used a Wizard of Oz methodology, where a human controls the assistant. The work is justified by the fact that interaction with development assistants is still in its infancy, and so different design options need to be explored. We agree with this, but in addition, we propose including traceability support and the possibility to coordinate multiple agents.

Devy is a voice-based assistant for development tasks related to version control (Bradley et al., 2018). Devy is an intent-based chatbot, and so, it maps high-level user intents into low-level commands. Intents may have parameters modelling required information, and Devy asks for their value when absent. In our work, we also found that intent-based agents are suitable to map user intentions into complex IDE commands, but in addition, we can combine LLM- and intent-based agents. Other types of assistants have been included into IDEs, such as a recommender for commands within Eclipse (Gasparic and Ricci, 2020).

Section 5 will evaluate our proposed assistant on one particular task: method renaming. Different approaches have been proposed for this task. For example, Liu *et al.* report on a classifier based on a deep learning architecture that first identifies method names that are not consistent with the code, and then proposes a new name for them (Liu et al., 2019). Instead, Zhang *et al.* use the code history to train a random forest classifier to state whether a method needs renaming, and if so, produce a name suggestion (Zhang et al., 2023). Our assistant uses LLMs to suggest new method names, but needs to be explicitly invoked by the user, i.e., it lacks a classifier that detects the need for renaming.

¹⁵<https://www.genuitec.com/products/copilot4eclipse/>

¹⁶<https://www.jetbrains.com/pycharm/>

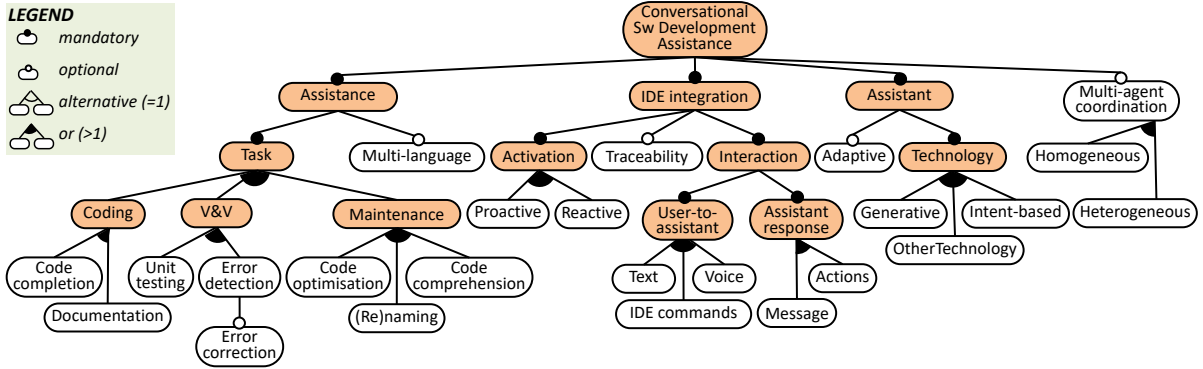


Figure 1: Dimensions of conversational assistance for software development.

Overall, we find different proposals of conversational assistants for software engineering with a good acceptance among developers. However, we identify the following gaps in the state of the art. Firstly, the integration of the assistants in IDEs, when existent, is ad-hoc. Beyond autocompletion, the assistants’ responses are most often messages and do not trigger IDE commands or modify existing artefacts. Secondly, the assistant contributions, their provenance and their rationale are not persisted, to the detriment of the project monitoring. Lastly, to our knowledge, no assistant combines or coordinates the contributions of several LLM- and intent-based conversational agents, in order to exploit the benefits of each of them. In the remainder of the paper, we present our proposal to address these issues.

3 DESIGNING CONVERSATIONAL DEVELOPMENT ASSISTANTS

Next, we present the main concepts in our approach. Section 3.1 presents a feature model with the dimensions of conversational assistance, Section 3.2 proposes a traceability model for assistance-based development, and Section 3.3 describes an execution and coordination model for development conversational assistants.

3.1 Dimensions of assistance

Figure 1 shows the dimensions relevant to conversational assistance for software development. It was elicited based on an analysis of the literature, and our own experience. It comprises the following four main features:

- *Assistance.* The assistance may be for one or more development tasks (feature Task in the figure), and be available for one or several programming languages (feature Multi-language). The feature model classifies tasks into coding tasks (code completion, documentation), validation & verification tasks (unit testing, (semantic) error detection, error correction), and maintenance tasks (code optimisation, code comprehension, renaming of methods, classes or attributes). This list of tasks is not meant to be exhaustive, but it is representative of the task types a conversational assistant can help with. For instance, we leave out tasks not directly related to programming, like assistance for modelling (Pérez-Soler et al., 2017) or versioning (Bradley et al., 2018).
- *IDE integration.* The integration of a conversational assistant into an IDE must consider several aspects. First, the Activation of the assistant may be Reactive (the developer explicitly asks for assistance) or Proactive (the assistant monitors the developer activity and provides assistance when it sees fit). Both styles are not mutually exclusive. In addition, the Interaction of the developer with the assistant (feature User-to-assistant) can be done through IDE commands (e.g., menus or buttons), Text in natural language (e.g., code comments like in GitHub Copilot, or through dedicated views), or Voice (Bradley et al., 2018). In the case of commands, the IDE needs to produce a textual prompt in natural language to send to the conversational assistant, together with the context of the assistance request (e.g., code fragment selected on the editor). For text and voice, the IDE may need to extend the developer prompt with additional context information. The response of the assistant (feature Assistant response) can be a message, or it may involve actions that modify development artifacts (e.g., inserting new code or comments into a

file, refactoring a code snippet). In the latter case, the assistant takes an active role, while in the former case, it acts as an informer or recommender of information. Finally, the developer-assistant interaction may be optionally traced (e.g., storing the query of the developer, the answer by the assistant, whether the recommendation was applied) and the IDE may mark the code fragments added or modified by the assistant.

- *Assistant.* The underlying Technology of the conversational assistant can be an agent based on generative artificial intelligence (LLMs), an intent-based chatbot, or other technologies (e.g., rule-based natural language processing as in (Pérez-Soler et al., 2017)). In addition, some assistants may be Adaptive to the context of use, e.g., checking or enforcing coding standards and norms used within a company, or learning from previous interactions with the developer.
- *Multi-agent coordination.* An assistant may integrate several conversational agents that help in different Heterogeneous tasks (e.g., coding and testing) or provide alternative solutions for the same task (feature Homogeneous, e.g., several agents that use different LLMs and prompts to suggest distinct code completions that the developer may choose from). If an assistant integrates several agents, then mechanisms for their coordination are required.

As we will describe in Section 4, our assistant CARET supports all the tasks of the feature diagram. It is not multi-language, as it specifically targets Java. Its activation is reactive, interaction is through both natural language text and IDE commands, their responses comprise both text and IDE actions (e.g., creating new files, inserting code into files) and it offers traceability of the developer-assistant interaction. CARET internally uses generative and intent-based technologies, is not adaptive, and coordinates multiple heterogeneous conversational agents.

3.2 Tracing assistant contributions

Keeping a trace of the interactions with the assistant can be useful for project management. The trace would record the contributions of the assistant to the code, along with the developers' requests that originated that code. This can be exploited for code reporting and analysis purposes, making it possible to see what the assistant did, where, when and why. It would also be possible to undo/redo the assistant contributions for exploratory purposes. Besides, the assistant-produced code may require more thorough

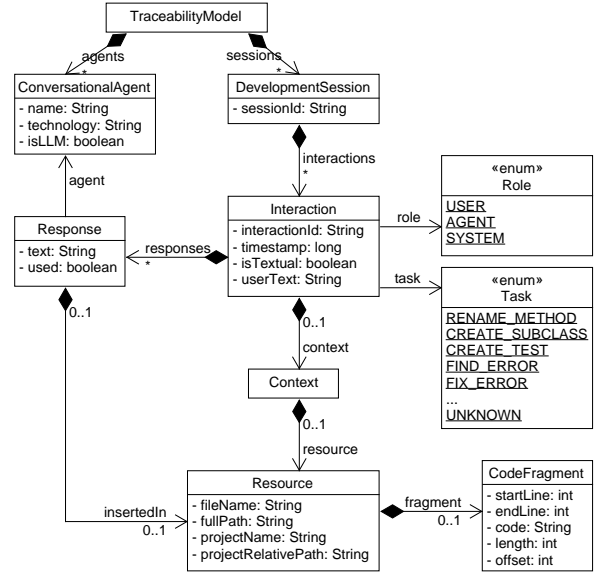


Figure 2: User-assistant interaction traceability model.

testing than the human-produced code, so tracking the former code would make it easier to identify and subsequently test.

Our approach to tracing the assistant contributions comprises two elements: a traceability model to store the interactions, and a set of code annotations that identify the code fragments introduced by the assistant. This enables bidirectional traceability: from past developer-assistant interactions into the modified code, and from the code to the originating interaction.

Figure 2 shows the traceability model. It records, for each *DevelopmentSession*, the *Interactions* between the user and the assistant that take place during the session. The interactions have an identifier, a timestamp, the role of the interacting participant (user, agent or system), whether the interaction was started by a text message or an IDE command (attribute *isTextual*), the text entered by the user, the development task resulting from the interaction (e.g., rename method, create subclass), and a *Context* that depends on the particular task. More in detail, the context may include any code fragment used to formulate the request to the assistant, in which case, the context stores both the *CodeFragment* and its container *Resource*. For example, this would be the context information stored for a request such as “document the behaviour of this method”. Alternatively, the context can be a file (e.g., for requests like “create a class implementing this interface”), a folder (e.g., for requests like “create a new sub-package called util”), or empty (e.g., for requests like “create a new Java project”).

The *Response* to the interaction comprises the text answered by the assistant (which may combine both code and textual explanations), the agent producing

it, and whether the developer actually used the suggested code. In the latter case, the model records the resource in which the code was inserted, and the position of the code in the resource. For each agent, the model stores its name, its technology, and whether it is based on LLMs. The latter information is relevant for coordinating multiple agents, as the next subsection will show.

To trace from the program the assistant-produced code, we propose the use of code annotations (Guerra et al., 2010). Whenever an assistant introduces a code snippet, the outer enclosing code block is automatically annotated to mark the interaction causing it (using the interaction identifier). In particular, if the assistant adds a method, this becomes annotated; if it adds a code fragment within a method, the enclosing method is annotated; and if it adds a class, interface or enumeration, these receive the annotation. In addition to the interaction identifier, the annotations carry additional meta-data, such as the task being solved and the agent that suggested the code.

In Section 4.2, we will describe the Java annotation we have created for the contributions of CARET.

3.3 Orchestrating conversational agents

As Section 3.1 discussed, a conversational assistant may integrate multiple conversational agents for the same or different tasks, built with different technologies, and invoked either by IDE commands (e.g., menus) or natural language text/voice requests. To coordinate the agents, we propose the scheme displayed in Figure 3.

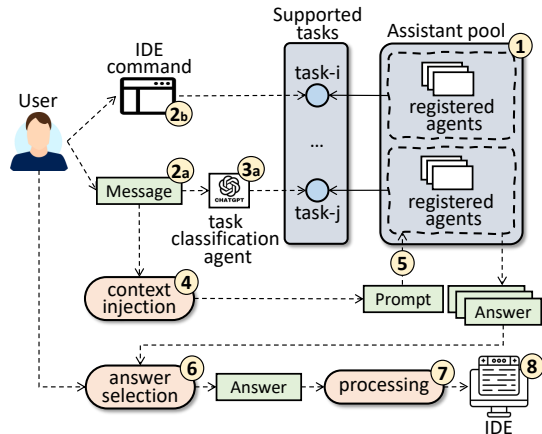


Figure 3: Orchestrating conversational agents.

Agents are registered for the tasks they are able to manage (label 1 in Figure 3). Users can make assistance requests via messages in natural language (label 2a) or commands of the IDE (label 2b). When

this happens, the first step is to select the agents that know how to handle the user request. The case of IDE commands is direct, since the command (e.g., create a class implementing an interface) is linked to a concrete task, from which the set of suitable (registered) agents can be retrieved. The case of natural language messages is more complex, as it requires the classification of the message into a task. To carry out this classification, we propose using an LLM agent (label 3a) with a prompt like:

You are a code assistant that helps software developers in programming tasks. Please classify into one of the next categories:

1. task-1

2. task-2

...

n. none of the above

the following request: “⟨user-message⟩”.

If the LLM agent responds “none of the above”, the user is informed that the assistant is not configured to assist with the requested task, and the current interaction is considered finished. Otherwise, if the LLM agent returns one of the listed tasks, the agents registered for that task shall be selected.

Next (label 4), a prompt is constructed using the user message or IDE command plus the context information (e.g., selected code in the IDE). This prompt is sent to the selected agents (label 5). Then, the user is presented the answers provided by the agents, and can select one of them (label 6). This answer needs to be processed (label 7) to extract the code and inject it as required into the software project, using the API of the IDE programmatically (label 8).

4 CARET

Next, we introduce CARET, a Java programming assistant we have built following the principles described in Section 3. Section 4.1 presents its architecture, and Section 4.2 describes its functionalities and showcases usage examples.

4.1 Architecture of CARET

CARET is a plugin for the Eclipse IDE that assists Java programmers in software development tasks. Figure 4 shows its architecture. The assistant integrates conversational agents of different technologies to process task requests, like OpenAI’s GPT-3.5 LLMs, Dialogflow and Rasa. In addition, it can be extended with other technologies by means of an extension point named AgentTechnology. Extension points

are the mechanism that Eclipse provides to allow adding functionality to a system externally (i.e., without changing its internal code).

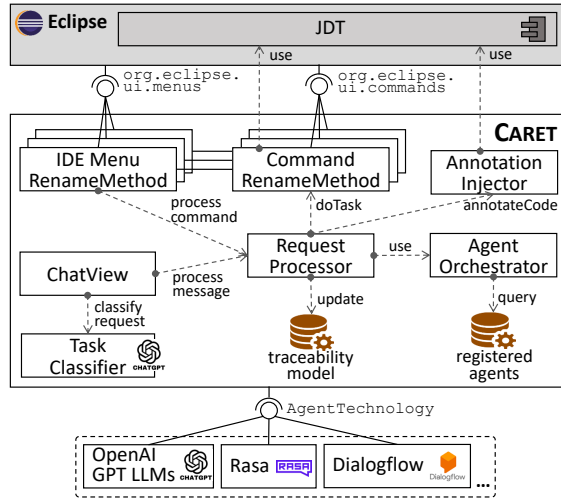


Figure 4: Architecture of CARET.

Users can request assistance in two ways: selecting in the IDE specific menus for each task, or writing a text request on a *Chat View*. In the former case, the selected menu determines unambiguously the task to perform. In the latter case, a *Task Classifier* tries to find the task that better fits the text request by sending the prompt drafted in Section 3.3 to a GPT-3.5 LLM.

The *Request Processor* coordinates the accomplishment of the tasks. It delegates the task execution to the command class that implements the task behaviour (there is one command class per task), passing the agents that will handle the task as parameters. The *Request Processor* obtains the agents from the *Agent Orchestrator*. Currently, the set of registered agents is predefined, and comprises one agent of each supported technology.

The above-mentioned command classes send to the agents a prompt tailored to the target task, which includes the user request and the necessary context information. The response from the agents is processed through an interface implemented by all agents conforming to the *AgentTechnology* extension point. The response includes text, the matching intent (if any), context information, and code suggestions. The *Request Processor* displays the response in the *Chat View*, and asks the user for confirmation to apply the suggestion. If the answer is positive, the project code is modified using the Eclipse JDT¹⁷, the interaction is traced (and can be saved/retrieved in JSON format), and the modified code is annotated.

¹⁷<https://projects.eclipse.org/projects/eclipse.jdt>

4.2 Tool support

The user can interact with the assistant by sending a message through the *Chat View* or using contextual menus that appear when right-clicking on the project files or a selected code fragment. Currently, CARET assists with the following tasks:

- **Code completion:** CARET is able to create a new project with the given name, a new class or interface with the given name in the current project, a class implementing a given interface, or a subclass of a given abstract class. It can also generate the code of a method, for which the user must provide either a description of the method, or the method name and its parameters.
- **Documentation:** It generates the Javadoc comments for a complete Java file. If the user does not provide a file but a code fragment, it can generate either Javadoc comments or line-by-line comments for the code.
- **Unit testing:** It creates a JUnit test for a given class.
- **Error detection and correction:** It can help detect simple semantic errors and propose corrections. Both functionalities rely solely on GPT-3.5 (i.e., the assistant does not integrate analysis or error detection/fix methods developed ad-hoc for Java).
- **Code optimisation:** CARET provides four optimisation options for a selected code fragment: efficiency improvement, readability improvement, complexity reduction, or general optimisation.
- **Code comprehension:** It produces an explanation in natural language of a selected piece of code.
- **Method (re)naming:** It renames a method to reflect its behaviour. Section 5 will evaluate the suitability of such renaming suggestions.

After processing the user request, the code of the suggested solution is displayed in a pop-up window, so that the user can decide to apply it or not. As an example, Figure 5 shows the response of CARET when the user selects the code of method “power” in the Java editor, and clicks on the menu option “Improve efficiency”. The suggested code improvement is displayed in a popup window. If the user accepts the suggestion, the suggested code is replaced in the Java editor, and the *Chat View* shows both the new code and its explanation.

In addition, accepting the assistant suggestion automatically adds a code annotation @Generated to the modified method, class or interface. This annotation – which we have designed for tracing CARET contributions – allows keeping track of the assistant-generated

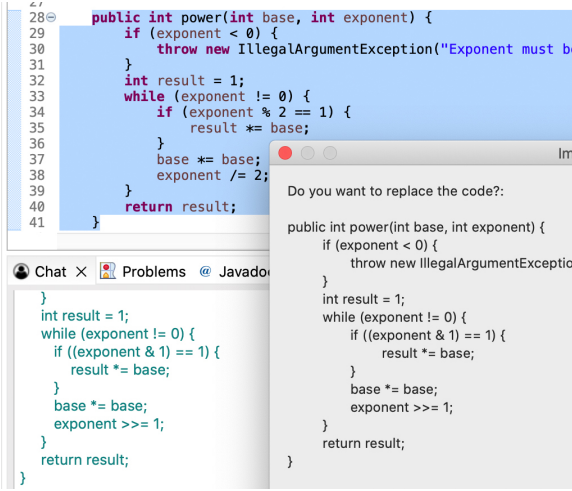


Figure 5: Screenshot of interaction with CARET. The popup window shows the code suggestion for improving the efficiency of method “power”.

code. It has four parameters: the name of the agent that produced the code, the performed task, the identifier of the interaction, and the timestamp (cf. Figure 2).

As an illustration, Figure 6 shows the code annotation added to the “factorial” method (lines 63–64). Its parameters indicate that the GPT-3.5 agent modified the method to reduce its complexity. For convenience, the *Chat View* at the bottom displays the introduced code, a “Copy code” shortcut button, and a “Go to” button which opens the file with the modified code and positions the cursor in the modified code. The latter information (modified resource and code fragment objects) is retrieved from the traceability model that stores the user-assistant interactions, as explained in Section 3.2.

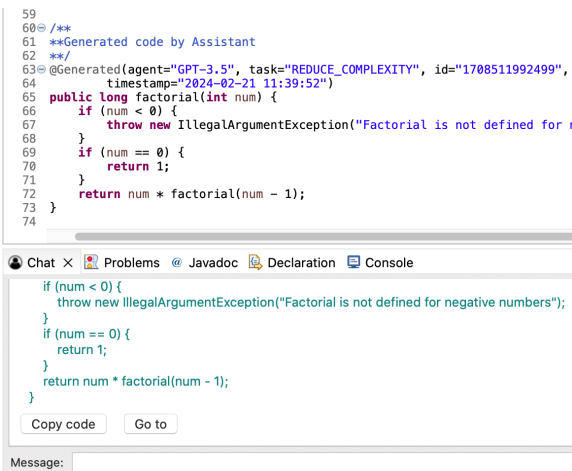


Figure 6: Screenshot of applied code suggestion for reducing the complexity of method “factorial”, and generated code annotation.

5 EVALUATION

This section evaluates the suitability of the assistance provided by CARET. Given the diverse range of tasks that CARET supports, we select to evaluate a representative one, which is method renaming, and leave the evaluation of the remaining ones for future work.

Method renaming is a common task during coding and maintenance. It seeks the alignment of the method name and its implementation. Good method names are important to make the code comprehensible – “if you have a good name for a method, you don’t need to look at the body” (Fowler, 1999) – while inconsistent method names make the code difficult to understand and maintain (Liu et al., 2019). As reviewed in Section 2.2, many different approaches have been investigated for this task. Our goal is to assess whether the LLM-based agents of CARET are fit for this task. Thus, our evaluation aims to answer the following research question (RQ):

Can CARET help to improve method names?

Next, Section 5.1 characterises the experimental setup, Section 5.2 describes the evaluation protocol, Section 5.3 analyses the results and answers the RQ, and Section 5.4 discusses the potential threats to validity.

The experiment results are available at <https://github.com/caretpro/experiment>.

5.1 Experiment setup

The evaluation considers four Java projects. Table 1 shows a summary of them, detailing the number of compilation units (i.e., classes, interfaces, enums), the number of methods, and the lines of code (LoC).

Name	# Units	# Methods	# LoC
Tutorial-compiler	11	66	2216
JVector	96	646	5221
Log4J-detector	19	117	3008
Ramen	78	362	5114
Total	204	1191	15559

Table 1: Summary of selected projects.

The first three projects in Table 1 were taken from GitHub public repositories using the following query:

created:>2021-10-01 stars:>100 size:<3500
path:*/.project language:Java

The goal of this query was to find popular Java repositories (with more than 100 stars), of medium size (less than 3500 Kb), created after the release of GPT-3.5 (October 2021). Thus, from the top of the

list of retrieved projects, we discarded those either too small or hard to build due to their numerous dependencies. The fourth project is a student project from a programming course at our university, stored in a private repository. Overall, the domains of the selected projects are diverse, comprising compilers¹⁸, embedded vector search engines¹⁹, vulnerability detection due to the use of Log4J²⁰, and a social network with a swing graphical user interface.

5.2 Experiment design

To evaluate the suitability of the method names suggested by CARET, we have performed a user study that follows the scheme depicted in Figure 7.

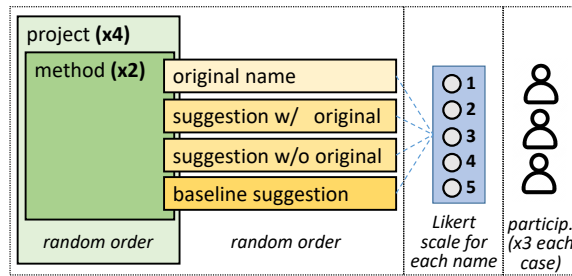


Figure 7: Scheme of the experiment design.

We first selected four Java projects as explained in Section 5.1. Then, we prepared a questionnaire with two parts: one collecting demographic data about the participants, and the other evaluating name suggestions for eight methods (two of each project). The method selection criterion was to have less than 20 LoC (to prevent participants from getting tired and to facilitate their understanding of the aim of the code) but not be trivial (e.g., getters and setters were excluded). For each method, the questionnaire presented its body and parameters, and suggested four names that participants had to rate using a 5-point Likert scale. The suggestions included the original method name, a baseline name made of the concatenation `<class-name>+“Method”`, and two names suggested by CARET using the GPT-3.5 agent with two variants of the prompt. The prompt of the first variant included the body and the original name of the method, while the second one included the body but not the method name. The GPT-3.5 agent used GPT-3.5-turbo with the parameter *temperature* set to 0.7. As an example, the next four names were presented for the same method: `insertNotDiverse` (orig-

¹⁸<https://github.com/wangjs96/>

A-tutorial-compiler-written-in-Java

¹⁹<https://github.com/jbellis/jvector>

²⁰<https://github.com/mergebase/log4j-detector>

inal), `concurrentNeighborSetMethod` (baseline), `insertNonDiverseNode` (CARET variant 1), and `updateNeighbors` (CARET variant 2).

Each evaluation case comprised 8 methods (2 from each project) and was evaluated by 3 participants. To avoid any bias, participants did not know how each name suggestion was generated, and the order of presentation of the methods and name suggestions was randomised. We recruited 12 participants in total, who evaluated 32 different methods, and 96 methods overall. The evaluation was conducted offline. Participants received the questionnaires by email and were given 5 days to submit their responses.

The questionnaires used are available at: <https://github.com/caretpro/experiment>.

5.3 Results and answer to RQ

Demographics of participants. The age of the participants ranged from 21 to 41 (31.9 years on average). Figure 8 summarises their demographic data. Overall, 83% of participants were men and 27% were women. In terms of educational level, 50% had a PhD, 34% had a master’s degree, 8% had a bachelor’s degree, and 8% were undergraduate students.

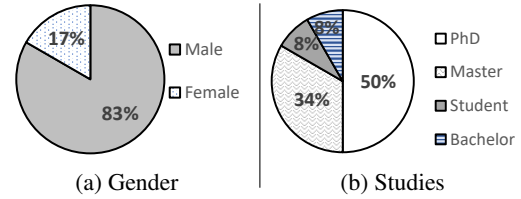


Figure 8: Demographic data of participants.

As Table 2 shows, the participants had an average of 9.75 years of experience in software development, and 4.75 years in Java development. They rated their knowledge of Java from 1 (none) to 5 (expert), obtaining an average of 3.42. Hence, overall, the participants declared to have good experience in software and Java development, and a fair knowledge of Java.

Experience	Average (years)
Software development	9.75
Java development	4.75

Table 2: Years of experience in development.

Evaluation results. Before analysing the responses to the questionnaires, it is worth noting that all method names generated by CARET are valid (e.g., they do not start with a number or special symbol) and follow the Java naming convention of being in lower camel-case.

With regard to the questionnaires, the box plots

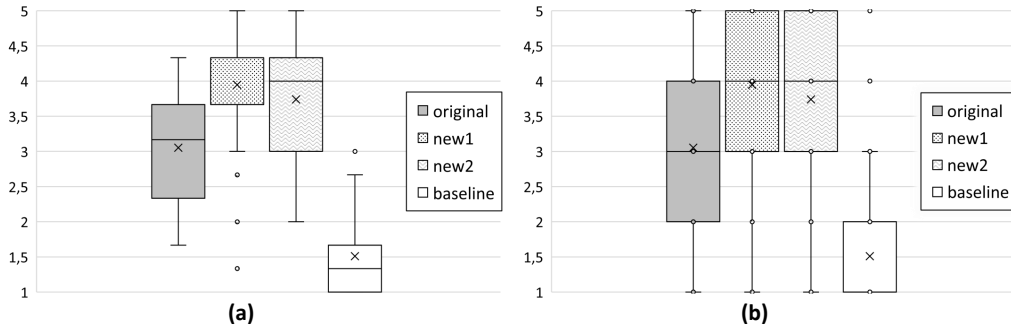


Figure 9: Distribution of scores of the suggested method names. (a) Distribution of the averages of the three scores received by each method. (b) Distribution of all scores (i.e., without averaging per method).

in Figure 9 depict the distribution of scores that each method renaming strategy received. In the box plots, *new1* corresponds to the assistant suggestions produced with the prompt that includes the original method name, and *new2* to those produced omitting the original method name. As Section 5.2 explained, 3 participants evaluated each method. Thus, Figure 9(a) shows the distribution of the average score values of each method (i.e., 32 data points per series), and Figure 9(b) shows the distribution of all scores without averaging per method (i.e., $32 \times 3 = 96$ data points per series).

We can see that the average score (marked with a cross in the box plots) is 3.05 (out of 5) for the original method names, 3.95 for strategy *new1*, 3.74 for strategy *new2*, and 1.51 for the baseline names. As expected, the baseline names were the lowest rated. Furthermore, in Figure 9(b), the median of the scores for the original method names is 3, while for the two assistant-generated method names is 4.

Figure 10 shows the results disaggregated by project, for the average scores (as in Figure 9(a)). Across all projects, the average and median of both *new1* and *new2* are higher than those of original, and baseline is consistently the worst.

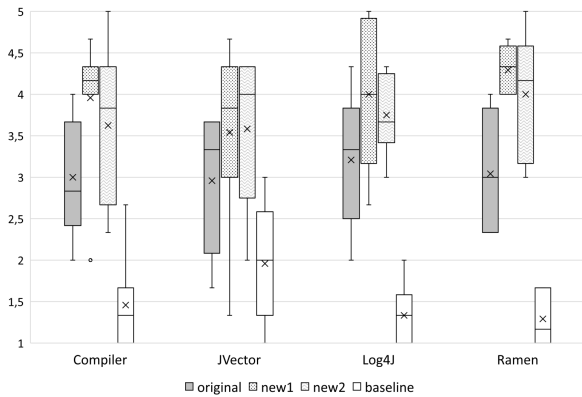


Figure 10: Distribution of scores disaggregated by project.

Now, we delve into the difference in score between the original method names and those suggested by the assistant. The left bar of Figure 11 shows the percentage of method names for which the average score of both *new1* and *new2* is higher than the average score of original. Overall, both *new1* and *new2* scored higher in more than half of the methods. The bar on the right shows the percentage of methods where either *new1* or *new2* is ranked higher than original. In this case, either *new1* or *new2* was ranked higher than the original name for more than 93% of the methods.

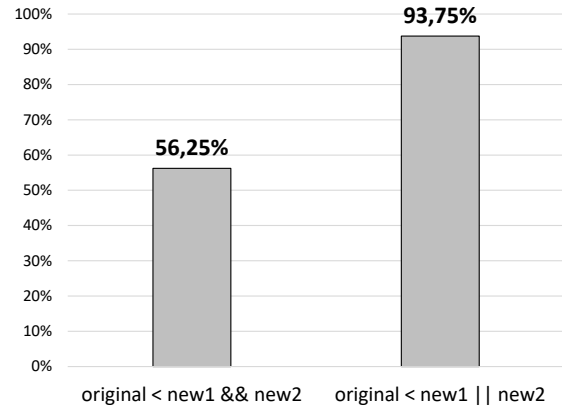


Figure 11: Comparison of scores between the original method names and the assistant suggestions.

Finally, to analyse if the difference in scores of *new1*, *new2* and original is statistically significant, we use the Wilcoxon Signed-Rank Test (Wilcoxon, 1945) to compare sample groups by pair ratings.

First, we define the null hypothesis H_0 as “there is no difference between the median scores of the original names and the *new1* suggestions”. This test results in $W = 616$, $Z_{(cal)} = -4.917$, $\alpha = 0.05$, $Z_{(\alpha/2)} = 1.96$, and $p - value = 0.0000008$. Since $p - value < \alpha$, we reject H_0 and state with 95% confidence that there is a significant difference between the medians of the scores of the original names and the *new1* suggestions.

Second, we set H_0 to “*there is no difference between the median scores of the original names and the new2 suggestions*”. This test results in $W = 797.5$, $Z_{(cal)} = -3.356$, $\alpha = 0.05$, $Z_{(\alpha/2)} = 1.96$, and $p - value = 0.0007897$. Since $p - value < \alpha$, we reject H_0 and state with 95% confidence that there is a significant difference between the median scores of the original names and the new2 suggestions.

Finally, we set H_0 to “*there is no difference between the median scores of the new1 and new2 names*”. This results in $W = 1264$, $Z_{(cal)} = -1.227$, $\alpha = 0.05$, $Z_{(\alpha/2)} = 1.96$, and $p - value = 0.2196459$. Since $p - value > \alpha$, we accept H_0 and state with 95% confidence that there is no significant difference between the median scores of the new1 and new2 names.

Answering the RQ. For the used dataset, the participants perceived the original method names as less appropriate than the suggestions new1 and new2 produced by CARET. Hence, we can answer that CARET suggestions could have helped to improve the method names in this study.

5.4 Threats to validity

Internal validity refers to the extent to which there is causal relationship between the conducted experiment and the resulting conclusions. We attempted to avoid any bias in the data by selecting Java projects developed by third parties, which were not present in GPT3’s training data. We also tried to prevent bias in the experiment by randomising the order of the projects and method names in the questionnaires, and by not revealing to the participants which mechanism was used for each presented method name.

External validity concerns the generalisability of the results. The study involved 12 participants who evaluated 384 alternative method names for 96 method blocks (32 unique ones) coming from 4 projects. While this is a fair amount of data, more evidence would be obtained with larger sets of participants and methods. Moreover, the participants rated methods with less than 20 LoC, so the results may differ for longer methods. Our study used GPT-3.5 with a temperature value of 0.7, but we cannot claim that this is the best value for solving the method renaming task. In the future, we will experiment with other temperature values to assess the quality of the output.

Construct validity is the extent to which an experiment accurately measures the concept it intends to evaluate. Since our evaluation is based on a subjective assessment of the appropriateness of the method names, we compiled 3 evaluations per method and averaged the scores. We did not consult the original

project developers (e.g., via pull requests as in (Liu et al., 2019)), but 12 independent developers evaluated the method names. To validate that the opinion of the participant developers was aligned and there were no outliers, we measured the inter-rater reliability using Fleiss’ kappa (Fleiss, 1971). The level of agreement between the participants was between 0.2 and 0.4 in all projects which, according to (Landis and Koch, 1977), can be considered fair.

6 CONCLUSIONS AND FUTURE WORK

Intelligent conversational assistants will soon become an integral part of most development processes and environments (Ozkaya, 2023b). With this expectation, we have explored the option space for their integration within IDEs, and proposed a traceability model and a coordination scheme for multiple conversational agents. We have realised our proposal in CARET, a Java assistant for Eclipse that helps in tasks such as code completion, documentation, code optimisation, and unit test generation. Finally, we have conducted a user study of the method renaming task supported in CARET, with very promising results.

We are currently improving CARET to allow registering any number of conversational agents that would be orchestrated in combination with the predefined ones. We would like to add further homogeneous agents, e.g., focused on autocompletion using LLMs for code, such as GitHub Copilot. Our goal is to automate the creation of conversational assistants for other programming languages (e.g., Python or C++), Eclipse plugins, testing frameworks (e.g., Cucumber), or model-driven development (e.g., the Eclipse Modeling Framework (Steinberg et al., 2008)). We also plan to explore the possibility to inject additional context to the assistants by prompts that include, e.g., the user expertise or company-specific coding standards and guidelines. Finally, as with method renaming, we intend to evaluate the other tasks supported by CARET, taking as a basis works on evaluation of LLMs for code (Chen et al., 2021).

ACKNOWLEDGEMENTS

Work funded by the Spanish MICINN with projects SATORI-UAM (TED2021-129381B-C21), FINESSE (PID2021-122270OB-I00), and RED2022-134647-T.

REFERENCES

- Abdalkareem, R., Shihab, E., and Rilling, J. (2017). What do developers use the crowd for? A study using stack overflow. *IEEE Softw.*, 34(2):53–60.
- Barke, S., James, M. B., and Polikarpova, N. (2023). Grounded Copilot: How programmers interact with code-generating models. *Proc. ACM Program. Lang.*, 7(OOPSLA1):85–111.
- Bradley, N. C., Fritz, T., and Holmes, R. (2018). Context-aware conversational developer assistants. In *ICSE*, pages 993–1003. ACM.
- Brambilla, M., Cabot, J., and Wimmer, M. (2017). *Model-driven software engineering in practice, 2nd edition*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers.
- Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., et al. (2021). Evaluating large language models trained on code. *CoRR*, abs/2107.03374.
- Chen, Y., Fu, Q., Yuan, Y., Wen, Z., Fan, G., Liu, D., Zhang, D., Li, Z., and Xiao, Y. (2023). Hallucination detection: Robustly discerning reliable answers in large language models. In *CIKM*, pages 245–255. ACM.
- Fleiss, J. L. (1971). Measuring nominal scale agreement among many raters. *Psychological Bulletin*, 76:378–382.
- Fowler, M. (1999). *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley.
- Gamma, E., Helm, R., Johnson, R., and Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition.
- Gasparic, M. and Ricci, F. (2020). IDE interaction support with command recommender systems. *IEEE Access*, 8:19256–19270.
- Guerra, E. M., Cardoso, M., Silva, J. O., and Fernandes, C. T. (2010). Idioms for code annotations in the Java language. In *SugarLoafPLOP*, pages 7:1–7:14. ACM.
- Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, A. (1990). Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.
- Landis, J. R. and Koch, G. G. (1977). The measurement of observer agreement for categorical data. *Biometrics*, 33:159–174.
- Li, R. et al. (2023). StarCoder: May the source be with you! *CoRR*, abs/2305.06161. See also <https://huggingface.co/blog/starcoder>.
- Liu, K., Kim, D., Bissyandé, T. F., Kim, T., Kim, K., Koyuncu, A., Kim, S., and Traon, Y. L. (2019). Learning to spot and refactor inconsistent method names. In *ICSE*, pages 1–12. IEEE / ACM.
- Ozkaya, I. (2023a). Application of large language models to software engineering tasks: Opportunities, risks, and implications. *IEEE Softw.*, 40(3):4–8.
- Ozkaya, I. (2023b). The next frontier in software development: AI-augmented software development processes. *IEEE Softw.*, 40(4):4–9.
- Pérez-Soler, S., Guerra, E., de Lara, J., and Jurado, F. (2017). The rise of the (modelling) bots: Towards assisted modelling via social networks. In *ASE*, pages 723–728. IEEE Computer Society.
- Pérez-Soler, S., Juárez-Puerta, S., Guerra, E., and de Lara, J. (2021). Choosing a chatbot development tool. *IEEE Softw.*, 38(4):94–103.
- Rich, C. and Waters, R. C. (1988). The programmer’s apprentice: A research overview. *Computer*, 21(11):10–25.
- Robe, P. and Kuttal, S. K. (2022). Designing PairBuddy – A conversational agent for pair programming. *ACM Trans. Comput.-Hum. Interact.*, 29(4).
- Ross, S. I., Martinez, F., Houde, S., Muller, M., and Weisz, J. D. (2023). The programmer’s assistant: Conversational interaction with a large language model for software development. In *IUI*, pages 491–514. ACM.
- Savary-Leblanc, M., Burgueño, L., Cabot, J., Pallec, X. L., and Gérard, S. (2023). Software assistants in software engineering: A systematic mapping study. *Softw. Pract. Exp.*, 53(3):856–892.
- Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. (2008). *EMF: Eclipse Modeling Framework, 2nd edition*. Pearson Education.
- Wasowski, A. and Berger, T. (2023). *Domain-specific languages - Effective modeling, automation, and reuse*. Springer.
- Wilcoxon, F. (1945). Individual comparisons by ranking methods. *Biometrics*, 1:196–202.
- Xu, F. F., Alon, U., Neubig, G., and Hellendoorn, V. J. (2022a). A systematic evaluation of large language models of code. In *MAPS@PLDI*, pages 1–10. ACM.
- Xu, F. F., Vasilescu, B., and Neubig, G. (2022b). In-IDE code generation from natural language: Promise and challenges. *ACM Trans. Softw. Eng. Methodol.*, 31(2).
- Yang, Y., Xia, X., Lo, D., and Grundy, J. C. (2022). A survey on deep learning for software engineering. *ACM Comput. Surv.*, 54(10s):206:1–206:73.
- Zhang, J., Luo, J., Liang, J., Gong, L., and Huang, Z. (2023). An accurate identifier renaming prediction and suggestion approach. *ACM Trans. Softw. Eng. Methodol.*, 32(6):148:1–148:51.
- Zhao, W. X. et al. (2023). A survey of large language models. <https://arxiv.org/abs/2303.18223>.