Integrating Conversational Assistants within Software Development Environments: An Extensible Approach

Albert Contreras[0009−0006−6887−9826], Esther Guerra[0000−0002−2818−2278], and Juan de Lara[0000−0001−9425−6362]

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

Abstract. Recent advances in generative artificial intelligence are reshaping our daily lives. Large language models (LLMs) – the technology underlying chatbots like ChatGPT – are able to produce coherent text responses upon user prompts. For this reason, LLMs are being used to automate tasks in many disciplines, like law, human resources, marketing, or media content creation. Software development is no exception to this trend, and conversational assistants based on LLMs have started to appear. However, there is still the need to understand the integration and interaction possibilities of these assistants within *integrated development environments* (IDEs), enabling the addition of new assistive tasks in a simple manner, coordinating multiple assistants, and tracing the assistants' contributions to the project under development.

We tackle this gap by exploring alternatives for integrating assistants within IDEs, and proposing a general architecture for conversational assistance in IDEs. The architecture features extensibility mechanisms to add new assistive tasks externally without resorting to programming, a rich traceability model of the user-assistant interaction, and a multi-assistant coordination model.

We have realised our proposal within Eclipse, building an assistant for Java development called CARET. The assistant supports tasks like code completion, documentation, code comprehension, maintenance and testing, but can be easily extended with additional ones. Finally, we present an evaluation for one of these tasks: method renaming. The evaluation results are promising since the recommendations of our assistant were generally perceived as more appropriate than the original method names and a baseline.

Keywords: Software Development · Conversational Assistant · Large Language Model · IDE · Eclipse · Java · Method Renaming.

1 Introduction

Since its beginning, software engineering has been continually striving to achieve higher levels of productivity and quality [35]. This goal has been pursued by different strategies, including automation techniques [6], the use of development languages with high-level of abstraction [38], knowledge bases and FAQs documenting development expertise [1], powerful integrated development environments (IDEs, like Visual Studio

Code[1], IntelliJ IDEA[2], or Eclipse[3]), catalogues of design patterns [13], or development assistants and recommender systems [29,34]. In this work, we are interested in the latter approaches.

Recent advancements in natural language processing (NLP), deep learning and generative artificial intelligence have resulted in the emergence of *open domain* conversational agents, which are able to produce sensible responses upon arbitrary user inputs (called *prompts*). These agents, also called chatbots[4], are currently being explored to solve many types of tasks in domains like law, human resources, marketing, media content creation, and software development, among others. They are powered by large language models (LLMs), which are transformer-based neural networks trained on vast amounts of (text) data [44].

Specific to software development, LLMs *fine-tuned* on code have been proposed [40], such as Codex[5], Code Llama[6], and StarCoder [19] (see [45] for a survey). Some of them are even integrated into IDEs, such as GitHub Copilot[7]. Still, assistant-based development is in its childhood, with many challenges to address and assistance strategies to assess [24]. This way, researchers working on assistants for software development may wonder: *What are the possible ways to integrate assistants into IDEs? Is it possible to store and retrieve past developer-assistant interactions, e.g., for version control? Can new assistive tasks be provided externally? How can multiple assistants be coordinated? How can the assistant effectiveness be assessed?*

Our work aims at answering the previous questions. For this purpose, we first present a taxonomy – using a feature diagram [16] – that describes the possibilities for integrating conversational assistants into IDEs. Then, we propose a general, extensible architecture for conversational assistance within IDEs. Our architecture enables the specification of new assistive tasks externally, and can coordinate the recommendations of several chatbots, not necessarily built using LLMs. Our approach reifies and persists the interactions between the developers and the assistant as a traceability model. This allows tracking the decisions made, storing them under standard version control systems, and 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). CARET[8] helps in a wide range of development tasks, including code completion, refactoring, maintenance, documentation, unit test generation and program comprehension. It has an extensible architecture that supports the incorporation of new assistive tasks externally without the need for coding. It also features a bidirectional traceability model from reified user-agent interactions to code, and vice versa, via code annotations.

[1] https://code.visualstudio.com/
[2] https://www.jetbrains.com/idea/
[3] https://www.eclipse.org/
[4] 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
[8] See https://caretpro.github.io/.

This paper also presents an evaluation of the suitability of one of the development tasks supported by CARET: method renaming. In the experiment, 12 participants evaluated the suitability of 96 method names of Java projects retrieved from GitHub. The results show that the names proposed by CARET were generally perceived as more appropriate than the original ones, which confirms the usefulness of assistants like CARET.

This paper is an extension of a previous version presented at the ENASE'2024 conference [9]. The new contributions of this paper are as follows. First, we have expanded the feature model that describes the dimensions of conversational assistance for software development (cf. Figures 1–4). Second, we have made the architecture of CARET extensible to facilitate the addition of new assistive tasks externally. Such an extensibility mechanism profits from the extension points of Eclipse, and we showcase its use by the definition of a new assistive task to refactor existing Java code into lambda expressions [22]. Finally, we perform a more comprehensive revision of related works.

The rest of this paper is organised as follows. Section 2 provides background on conversational assistants, and Section 3 analyses current research in the area. Next, Section 4 presents the four main components of our proposal: the analysis of the assistant-IDE integration possibilities, the traceability model, the new extensibility mechanism, and the coordination of multiple conversational agents into a unified assistant. Section 5 introduces CARET, our Java assistant for Eclipse. Afterwards, Section 6 reports on the evaluation of one representative task that relies on the use of LLMs: method renaming. Finally, Section 7 ends with the conclusions and open research lines.

2 Background on Conversational Assistants

Depending on their scope, conversational agents (or *chatbots*) can be either *task-oriented* or *open-domain*. In the reminder of this section, we describe the main concepts of each chatbot type.

Task-oriented chatbots. These chatbots are a popular means to access software services using natural language. Their use has risen because they can help users access services like customer support, banking or shopping, and can be easily embedded into social networks (e.g., Telegram, Slack), websites or intelligent speakers. They are called *task-oriented* because they help users in performing a specific task.

Many technologies to build task-oriented chatbots are in use nowadays [27]. Some prominent examples are Google's Dialogflow[9], the IBM Watson Assistant[10], Microsoft's Bot Framework[11], Amazon Lex[12] or Rasa[13]. They support the specification of the user *intents* that the chatbot should recognise (e.g., ordering food, setting a medical appointment). Intents declare training phrases, which are used to train a natural language understanding (NLU) engine. This way, when the user inputs an utterance, the NLU engine selects the most likely intent with a certain confidence. If the confidence is below a

[9] https://dialogflow.com/
[10] https://www.ibm.com/cloud/watson-assistant/
[11] https://dev.botframework.com/
[12] https://aws.amazon.com/en/lex/
[13] https://rasa.com/

threshold, then a *fallback* intent is selected, if one is available. Fallback intents are an indication of user requests that the chatbot could not handle.

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

Open-domain chatbots. 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 ChatGPT[14] or Google's Gemini[15] (formerly known as Bard). LLMs are transformer-based neural networks trained on vast amounts of textual data [37,40]. They can deliver a sensible text output on arbitrary user *prompts* without the requirement to pre-define permissible user intents.

Rather than being specific to performing a certain task, LLMs are typically open-domain. Nevertheless, some of them have been *fine-tuned* on specialised data, like code [40], or tasks, like compiler optimisation [10]. Fine-tuning permits repurposing an LLM pretrained on generic text data for particular downstream tasks (e.g., question-answering) or domains (e.g., programming). However, since LLMs lack fallbacks, assessing the accuracy of the produced output, or asserting when an LLM does not know the answer, becomes hard. Hence, LLMs may incur in so-called *hallucinations* [8], i.e., inaccurate or false answers presented as a fact. In this respect, the unpredictability of LLMs can be regulated by the *temperature* hyperparameter. This way, a high temperature results in more imaginative but less predictable LLM outputs upon the same utterance, while lower temperatures entail more deterministic answers.

3 Related Works

The seminal ideas behind assistants for software development can be traced back to the 80's with the *programmer's apprentice* [29]. 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 [13]).

Nowadays, the focus of artificial intelligence has dramatically 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 [42].

Next, we examine the state-of-the art on conversational assistants for programming and other software development activities, and briefly report on approaches for the specific assistance task of method renaming.

Conversational assistants for programming. The appearance of LLMs [44] has prompted their use for software engineering. Several LLM-based programming assistants have been proposed. One of the first ones was GitHub Copilot, which was originally

[14] `https://openai.com/chatgpt`
[15] `https://gemini.google.com/`

built atop Codex [7], an LLM based on GPT-3 and fine-tuned on code. Copilot is integrated into several IDEs, like Visual Studio Code and JetBrains, and offers autocompletion assistance as the developer types. Being initially free, now Copilot is a paid feature. Its code completion capabilities have been recently integrated into the Eclipse IDE as a plugin[16]. This plugin provides autocompletion for several languages using GitHub Copilot, and therefore requires a subscription. Subsequently, other AI-powered conversational assistants for software development akin to Copilot have appeared, such as Amazon Q Developer[17] or Bito[18], to name a few. 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, companies or developers may like to extend the range of supported assistance tasks – which are fixed – and future assistant-enabled IDEs may need to coordinate several agents.

Gemini Code Assist[19] is a recent programming assistant powered by AI and natural language conversation, which overcomes some of the mentioned problems. It supports multiple IDEs and programming languages, and in addition to code completion, it has contextual actions that automate other development tasks such as test generation and code documentation. Moreover, it features a preliminary facility to customise the code suggestions based on proprietary codebases. Still, the assistive tasks and their activation mode are predetermined and fixed, it lacks traceability of the assistant-generated code, and it does not support incorporating or coordinating several agents to accomplish domain-specific tasks.

Interestingly, Barke and collaborators have recently used *grounded theory* to analyse how programmers interact with GitHub Copilot [4]. They detected two main usages of the assistant. The first one is for the *acceleration* of known tasks (i.e., autocompletion). The second is 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.

The Programmer's Assistant [31] is a recent conversational assistant for Python based on Codex. Users interact with it via natural language, and the context for the assistance can be provided by selecting code. The authors have documented the evolution of their engineered prompts to achieve the desired persona and behaviour, e.g., to overcome the LLM reluctance to answer some questions, or to diminish the didacticism of the LLM answers [32]. A user study revealed the utility and good acceptance of this assistant by developers [31]. 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 extensibility and traceability mechanisms are missing.

Xu and collaborators [41] propose two systems for code generation and retrieval from natural language, both integrated in the PyCharm[20] IDE for Python programming. The systems were evaluated to understand if they led to improved efficiency and qual-

[16] https://www.genuitec.com/products/copilot4eclipse/
[17] https://aws.amazon.com/q/developer/
[18] https://bito.ai/product/ai-chat-developers/
[19] https://cloud.google.com/products/gemini/code-assist
[20] https://www.jetbrains.com/pycharm/

ity. While they obtained mixed results, overall, developers declared enjoying the experience.

Robe and Kuttal explored design options for PairBuddy, a conversational assistant for pair programming, with a 3D embodiment [30]. A Wizard of Oz methodology [17] was used, where a human was controlling the assistant. The work is justified by the fact that interaction with development assistants is still in its beginnings, and hence 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.

Conversational assistants for software development beyond programming. The works analysed so far focus on coding. In addition, there are proposals of conversational assistants that help in other development-related tasks, such as version control [5], IDE usage [14], or code debugging [3].

Devy is a voice-based assistant for software development tasks related to version control [5]. Devy is an intent-based chatbot, and so, it maps high-level user intents into low-level commands. Intents may have parameters for required information, and Devy asks for their value if they are not provided. 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.

ROBIN is a multi-agent conversational assistant for code debugging in Visual Studio [3]. It leverages the context where an error occurs, and interactively requests missing information from the user to improve the accuracy of the assistance. The authors have evaluated ROBIN through a with-in subject study, where employing the assistant resulted in improved bug localisation and resolution. While our current focus is on programming assistive tasks whose input is code, in the future, we plan to extend CARET to include other data sources such as debugging information, error logs or unit test results.

In relation to the development process, LCG is a code generation framework that emulates different software process models, with LLM agents playing specific roles such as requirements engineer, developer, tester, and the like [20]. The agents continuously refine themselves to enhance code quality. This way, the framework assists in refining the code generation process.

Other types of assistants have been included into IDEs, such as a recommender for commands within Eclipse [14].

Method renaming. Section 6 evaluates our proposed assistant on one particular task: method renaming. Different approaches exist for this task. For example, Alon and collaborators represent snippets of code as an attention-based neural network using a learned fixed-length continuous vector, and use this representation to predict method names, among other applications [2]. Liu and collaborators describe a classifier based on a deep learning architecture that first identifies method names that are inconsistent with the code in the method body, and then suggests a new name for them [21]. Instead, Zhang and collaborators use the code history to train a random forest classifier that signals whether a method needs renaming, and in such a case, returns a name suggestion [43]. Our assistant uses LLMs to suggest new method names, but needs to be explicitly invoked by the user, i.e., it lacks a monitor that detects the need for renaming.

Like us, some recent approaches leverage LLMs for method renaming. Recio and collaborators use the OpenAI Chat API to name new methods obtained by a refactor-

ing that extracts existing code into the new method without altering the overall behaviour [28]. They create prompts that include both the original and the extracted code, and ask the model to produce a method name for the extracted code. Our assistant works similarly but is not restricted to method extraction, and is accessible within the IDE. In the same vein, Nazari and collaborators use an LLM to produce candidate method names, and another LLM combined with a program verifier to validate them [23]. Their goal is improving the understandability of the code produced by program synthesisers, which tends to be unidiomatic and difficult to understand. Instead, our approach is simpler, targets Java, and integrates into Eclipse.

Thus, there is a wide variety of techniques to rename methods, some based on LLMs. We do not claim that our own renaming proposal surpasses all of them. Instead, our contribution lies in an extensible architecture that enables developers to effortlessly integrate their preferred conversational assistants (including the previous ones and ours) into their IDE (Eclipse in our proof of concept). This way, they can seamlessly use these assistants in their regular development process, via natural language without the need to change tools.

Overall, different proposals of conversational assistants for software engineering can be found, which generally show 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, do not trigger IDE commands, and only a few proposals modify existing artefacts. Secondly, the assistant contributions, their provenance and their rationale are not persisted, and hence cannot be put under version control, to the detriment of the project monitoring. Third, the set of tasks in which the assistant may help is fixed and predefined. As a consequence, adding new assistive tasks or customising the predefined ones becomes problematic. 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 following, we present our approach to address these issues.

4 Designing Conversational Assistants for Software Development

This section describes the ingredients of our approach. First, Section 4.1 presents a feature diagram that captures the main dimensions of conversational assistance. Then, Section 4.2 introduces a traceability model for assistance-based development. Finally, Section 4.3 proposes extensibility mechanisms to enable the addition of assistive tasks without resorting to programming, and describes an execution and coordination model for orchestrating assistive tasks.

4.1 Dimensions of assistance for software development

Figure 1 shows the main dimensions relevant to conversational assistance for software development. They are expressed as a feature diagram, a notation commonly used in product lines to express the variability of a system [16]. The figure includes just the four

most general top-level dimensions to consider, which are further refined in Figures 2–4. Specifically, one can look at the Assistance (i.e., tasks the assistant helps with), the IDE integration (i.e., how the assistant integrates with and interact through the IDE), the Assistant (i.e., technical aspects of the assistant's implementation), and the Multi-agent coordination (i.e., options for the coordination of multiple assistive agents). These dimensions and their features, which we will explain next, were elicited from an analysis of the literature and our own experience.

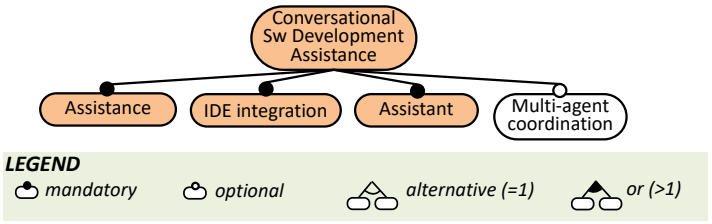


Fig. 1: General dimensions of conversational assistance for software development.

Assistance. This dimension, detailed in Figure 2, refers to the tasks for which the assistant provides help. The assistance can be for one or more development tasks (cf. feature Task in the figure), be available for one or several programming languages (cf. feature Multi-language), and optionally be extensible to accommodate new tasks (cf. feature Extensible). The feature diagram classifies tasks as related to coding (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). We do not claim that this list of tasks is exhaustive, but it is representative of the task types a conversational assistant can help with. For example, we have not included tasks not directly related to programming, like assistance for versioning [5], modelling [26] or debugging [3]. As for extensibility, it allows for the external addition of specialised or unforeseen tasks as needed (e.g., refactoring a block of Java code using lambda expressions), and can address specific company requirements and standards (e.g., generating Javadoc comments following certain convention, format or language).

IDE integration. Integrating a conversational assistant into an IDE must consider various aspects, as Figure 3 details. First, the Activation of the assistant may be Reactive (i.e., when the developer needs to explicitly ask for assistance) or Proactive (i.e., when the assistant monitors the developer activity in the background and provides assistance when it sees fit). Both styles are not mutually exclusive, and an IDE may offer assistance tasks with both types of activation.

Additionally, the developer–assistant Interaction (feature User-to-assistant) can be done through IDE commands (e.g., buttons or menus), natural language Text (e.g., comments

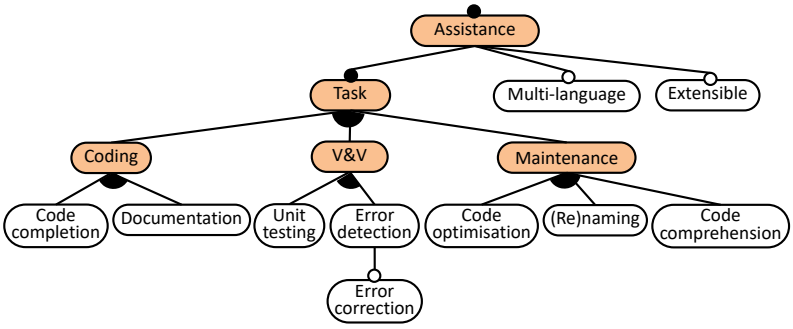


Fig. 2: Features of the Assistance dimension.

in the code like in GitHub Copilot, or through dedicated chat views), or Voice [5]. In the case of commands, the IDE needs to generate a textual prompt in natural language to send to the conversational assistant, together with the context of the assistance request (e.g., a code fragment currently 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., refactoring a code snippet, inserting new code or comments into a file). 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, optionally, the developer–assistant interaction may be traced (e.g., storing the query of the developer, the assistant answer, and whether the recommendation was applied) and the IDE may mark the code fragments added or modified by the assistant (e.g., for a more detailed testing).

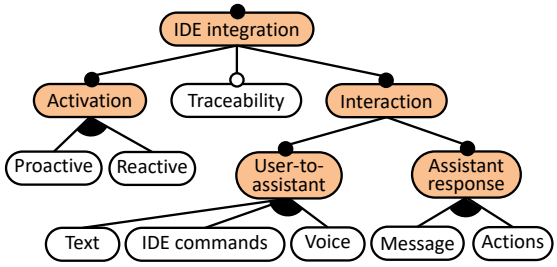


Fig. 3: Features of the IDE integration dimension.

Assistant. As Figure 4(a) depicts, the underlying Technology powering the conversational assistant can be generative based on LLMs, can be based on intents, or can rely on other technologies (e.g., rule-based natural language processing as in [26]). In addition, some assistants may be Adaptive to the context of use, for example, to check or

enforce coding standards and norms used within a company, or to learn from previous interactions with the developer.

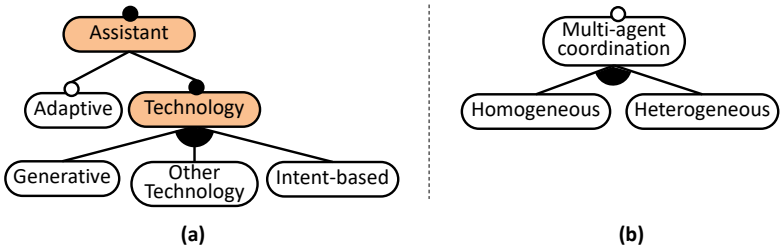


Fig. 4: Features of: (a) dimension Assistant and (b) dimension Multi-agent coordination.

Multi-agent coordination. As Figure 4(b) describes, an assistant may integrate different conversational agents helping in different Heterogeneous tasks (e.g., testing and coding) or offer alternative solutions for the same task (feature Homogeneous, e.g., several agents that use different LLMs and prompts to provide distinct code completions that the developer may choose from). If an assistant integrates several agents, then it requires mechanisms for their coordination.

As we will explain in Section 5, our assistant CARET supports all the tasks included in the feature diagram (cf. Figure 2). It is not multi-language, as it specifically targets Java, but it is extensible with new tasks. Its activation is reactive, the interaction is through both natural language text and IDE commands, its 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, coordinates multiple heterogeneous conversational agents, and it is not adaptive.

4.2 Tracing the contributions of the conversational assistant

Keeping a trace of the developer–assistant interactions and the actual assistant contributions can be useful for project management. The trace would record the contributions of the assistant to the project code, along with the developers' requests that originated that code. This way, the trace could be put under version control, as well as be exploited for code reporting and analysis purposes, enabling to check what the assistant contribution was, where, when and why. Moreover, it would also enable to undo/redo the assistant contributions for exploratory purposes. Besides, the assistant-produced code may require more thorough testing than the human-produced code, so tracking the former code would make it easier to identify and subsequently test it.

Our approach to trace the contributions of the assistant comprises two elements: a traceability model to store the interactions, and a set of code annotations to tag the code

fragments introduced by the assistant. As depicted in Figure 5, this enables bidirectional traceability: from past developer–assistant interactions into the assistant–injected code (label 1), and from the assistant-injected code back to the originating developer-assistant interaction (label 2).

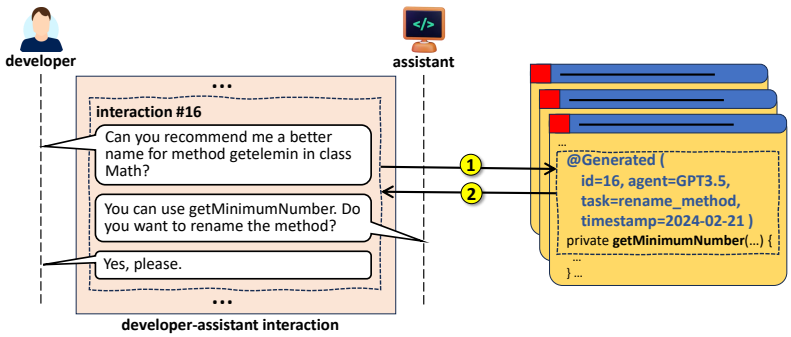


Fig. 5: Tracing developer-assistant interactions and their effect on the code.

Figure 6 displays a conceptual schema of the traceability model. The model records, for each DevelopmentSession, the Interactions between the developer and the assistant that take place during the session. The interactions have an identifier (InteractionId), a timestamp, the role of the interacting participant (user, agent or system, cf. Role enum), the development task resulting from the interaction (e.g., rename method, create subclass), and a Context that depends on the particular task. The interaction may start by a text message (class TextualInteraction) or an IDE command. In the former case, the text entered by the user is recorded. The traceability model assumes a catalogue of available assistive tasks (reference taskCatalogue). As we will see in Section 4.3, this catalogue is extensible via extension points.

The task context may include any code snippet 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" (i.e., the context in this case would comprise both the code of the method and the Java class file containing the method). Alternatively, the context can be a file (e.g., for requests like "create a class implementing interface IObservable"), a folder (e.g., for requests like "create a new sub-package called controllers"), or empty (e.g., for requests like "create a new Java project").

The Response to the interaction contains the text answered by the assistant (which may combine both code and textual explanations), the agent producing it, and whether the developer included the suggested code in the project. 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 subsections will show.

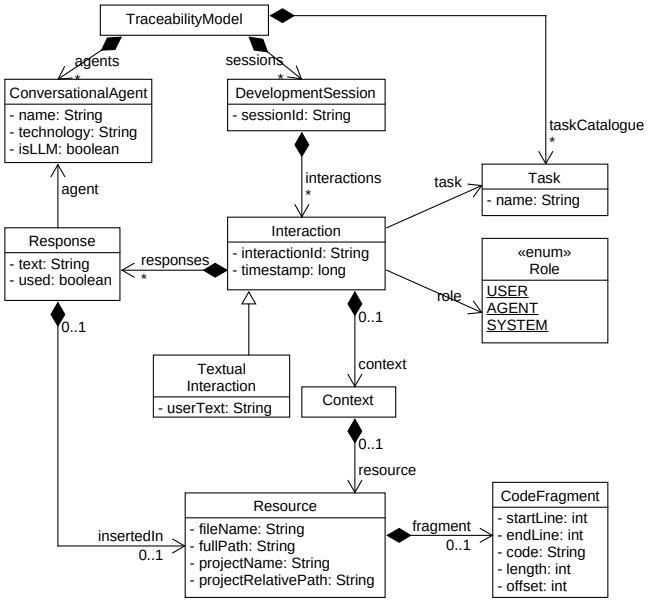


Fig. 6: Traceability model for the developer–assistant interaction (adapted from [9]).

To keep trace of the assistant-produced code within the program, we propose using code annotations [15] (cf. Figure 5). When a code snippet generated by the assistant is included in the project, 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 5.2, we will describe the Java annotation we have created for the contributions of CARET.

4.3 Extensibility and orchestration of assistive tasks

We have devised an extensible architecture that allows the external addition of new assistive tasks into the IDE without requiring programming. Assistive tasks are registered via an extension point, a mechanism present in many IDEs (e.g., Eclipse) and component-based frameworks. This way, arbitrary tasks (e.g., documenting a class, creating a class that satisfies some domain requirements) can be added to the IDE in a non-intrusive way.

Figure 7 shows a conceptual model of the extension point. It defines the data needed to register assistive tasks by subclassing from TaskGroup. This class groups related tasks, which can be bound to either menu commands, the usage context (e.g., when selecting

a code snippet), or none (i.e., the task is activated by natural language). Each Task defines named Parameters that have a description, can be required or not, and have a type (either a Java programming concept such as class or attribute, or other type, like "class description"). They may also indicate whether source code needs to be retrieved (to be passed to the LLM) or not (hasSource). For example, a task for creating a new class implementing an interface would declare a parameter for the interface name and the class name. The first parameter needs to retrieve code (the code of the interface), but the second parameter (for the new class) does not need to do so. Tasks have a unique code identifier, a name, a description, are carried out by an (LLM) Agent, and perform Actions. Section 5 will illustrate the registration of tasks using this extension point.

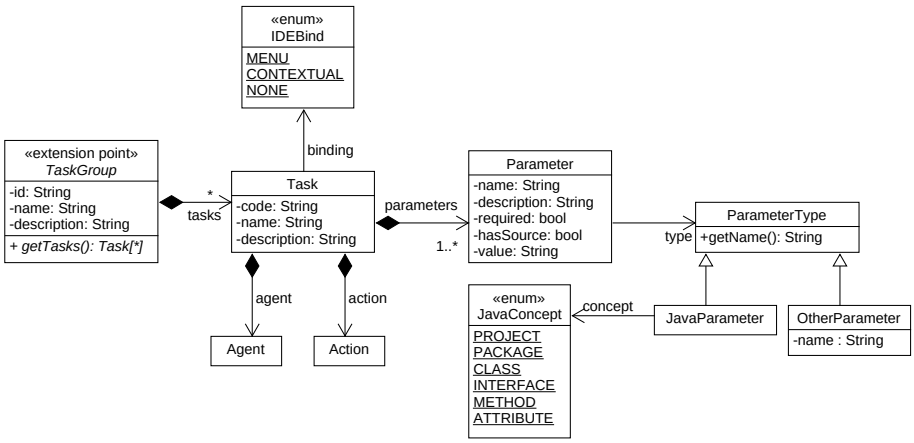


Fig. 7: AI-assistance extension point.

Our architecture also involves a generic assistant which is in charge of interacting with the user (typically a developer), identifying the kind of assistance he/she requests, and performing the most appropriate registered task to address the request. Figure 8 shows the working scheme of this assistant.

First, as we have seen, the extension point for assistive tasks requires specifying, among other elements, how developers can request the assistance: in natural language (label 1a), or via the usage context or commands of the IDE (e.g., a menu or a button, label 1b). In the first case, the developer states to the IDE what he/she wants to achieve, and the assistant aims to deduce which registered task is suitable to accomplish it (label 2a). For this purpose, the assistant builds a prompt with the developer request and the set of registered tasks, and asking to classify the request into a task and to extract the relevant parameters from the request (label 3a). This prompt is sent to an LLM (label 4a), which outputs the classified task and the detected parameters (label 5a). For example, given the developer request "*I need a class Book with title and author*", the LLM classifies it into the task "*create a class*", and extracts the class name (*Book*) and description (*with title and author*) as parameters. The assistant can use these parameters as

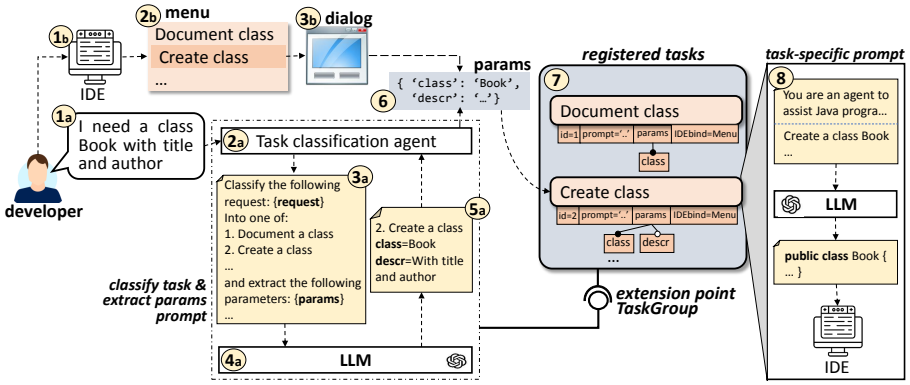


Fig. 8: Working scheme of the extensible assistant architecture.

context for the requested task. Conversely, if there is no registered task addressing the developer request, the assistant informs of this fact and the current interaction finishes.

Then, the assistant activates the identified task, passing to it the parameters (label 6). The task creates a prompt with two parts (label 8): a general one ("*You are an agent to assist Java programming*"), and a specific one providing the task description and parameters (specified in features Task.description and Task.parameters of the extension point, cf. Figure 7). This prompt is sent to the agent registered for the task (feature Task.agent in Figure 7), which typically though not necessarily will output code. For instance, in the previous example, the agent is an LLM that would return a Java class named *Book* with attributes *title* and *author*. Finally, the assistant applies the necessary actions in the IDE to complete the task (feature Task.action in Figure 7).

If the request comes from the usage context or an IDE command (label 2b), the task is uniquely determined. Hence, steps 1a to 5a are not necessary, but the developer is presented a dialog box to introduce the parameters needed for the task (label 3b), and the assistant proceeds from step 6.

5 CARET

This section presents CARET, a conversational assistant for Java programming that we have built following the principles described in Section 4. Section 5.1 introduces its architecture, and Section 5.2 describes its functionalities and showcases examples of use.

5.1 Architecture of CARET

CARET (`https://caretpro.github.io/`) is a plugin for the Eclipse IDE that assists Java programmers in software development tasks. Figure 9 shows its architecture. The assistant integrates conversational agents of different technologies to process assistive task requests, like OpenAI's GPT-3.5, Dialogflow and Rasa. In addition, it provides

two extensions points, called AgentTechnology and TaskGroup, which enable adding new agent technologies and assistive tasks to CARET, respectively. Extension points are the mechanism that Eclipse offers to allow adding functionality to a system externally, i.e., without changing its internal code.

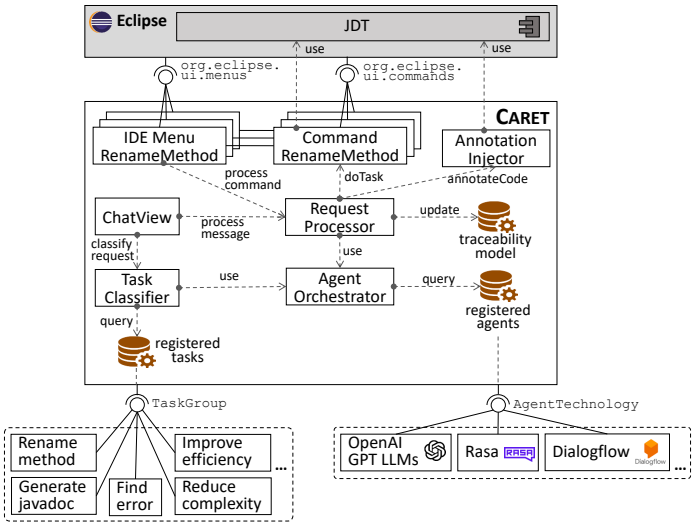


Fig. 9: Architecture of CARET (modified from [9]).

Eclipse users can request assistance to CARET in two ways: either 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. This is so as the extension point TaskGroup enables binding the registered assistive tasks to specific widgets of the IDE's GUI, like menus. Instead, in the case of a textual request, a *Task Classifier* tries to find the registered task that better fits the request. It does so by sending a prompt tailored to this classification problem to an LLM of the GPT family (label 3a in Figure 8).

Once identified the requested task, the *Request Processor* coordinates its accomplishment. It delegates the task execution to the command class that implements the task actions, passing the agent that will handle the task as a parameter. The *Request Processor* obtains the agents from the *Agent Orchestrator*. In turn, the orchestrator dynamically retrieves the agents that have been previously registered in the IDE using the AgentTechnology extension point. Thus, there may be agents of different technologies.

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 (label 8 in Figure 8). The response from the agents is processed through an interface that all agents conforming to the AgentTechnology extension point must implement. The response includes text, the matching intent (if any), context information, and code suggestions. The *Request Processor* displays the response – in the *Chat View* if the request

was textual, or in a popup window otherwise – and asks the user for confirmation to apply the code suggestion. If the answer is positive, the project code is modified using the Eclipse JDT[21], the modified code is annotated, and the interaction is traced and can be saved/retrieved in JSON format.

5.2 Functionality of CARET

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, defined via the extension point TaskGroup:

- 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 body 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 selected 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 6 will evaluate the suitability of such renaming suggestions.

After processing the user request, the code of the suggested solution is displayed either in the *Chat View* or in a pop-up window, depending on whether the request was issued textually or via commands. In both cases, the user can decide to apply the suggestion or not.

As an example, Figure 10 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". In this case, the suggested code improvement is displayed in a pop-up 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 code. It has four parameters: the name of the agent that produced the code,

[21] https://projects.eclipse.org/projects/eclipse.jdt

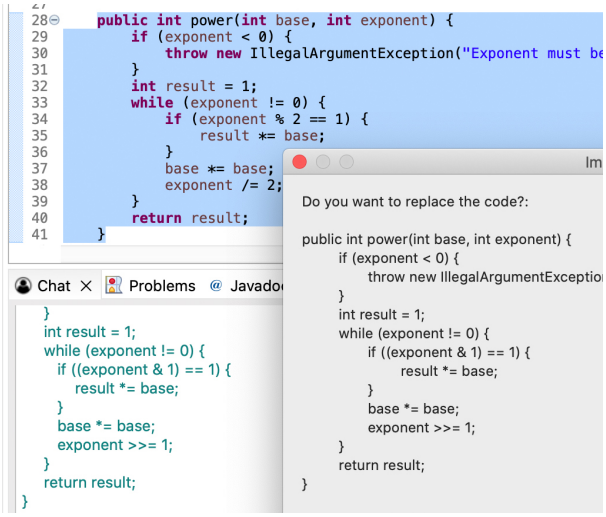


Fig. 10: Interaction with CARET. The pop-up window shows the code suggestion for improving the efficiency of method "power" (from [9]).

the performed task, the identifier of the interaction, and the timestamp (cf. Figures 5 and 6).

As an illustration, Figure 11 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 4.2.

CARET permits incorporating new assistive tasks as extensions of the extension point TaskGroup. For instance, Figure 12 shows the definition of a task group called "Lambdas", aimed at supporting activities related to the use of lambda expressions in Java. The group contains the definition of the specific task "Convert code using lambdas", which has a mandatory parameter called "methodName" and an optional parameter called "className". This way, to convert a piece of code into a lambda expression, the user will always have to provide the name of a method (and the assistant will prompt for its value if absent), but the class name can be provided or not. The right part of the figure shows the details of the definition of parameter "className". It comprises its name, description, whether it is required, whether it requires providing source code, and its type (one of those of the enumerate type JavaConcept in Figure 7). Overall, adding this task to CARET only requires completing the fields in the figure, but no lines of code.

Once a task has been defined in this way, it becomes readily available to be requested through the *Chat View* via a text message, or by selecting a dedicated context menu displayed when right-clicking on a code fragment. Figure 13 illustrates the use of the new task. By selecting the code of method "sortNameList" and right-clicking on it, a

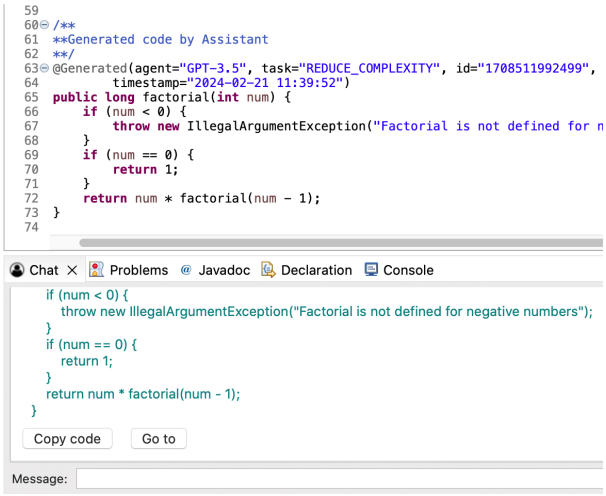


Fig. 11: Screenshot of applied code suggestion for reducing the complexity of method "factorial", and generated code annotation (from [9]).

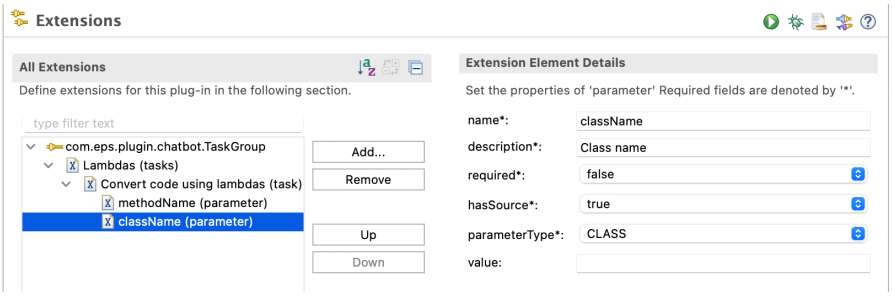
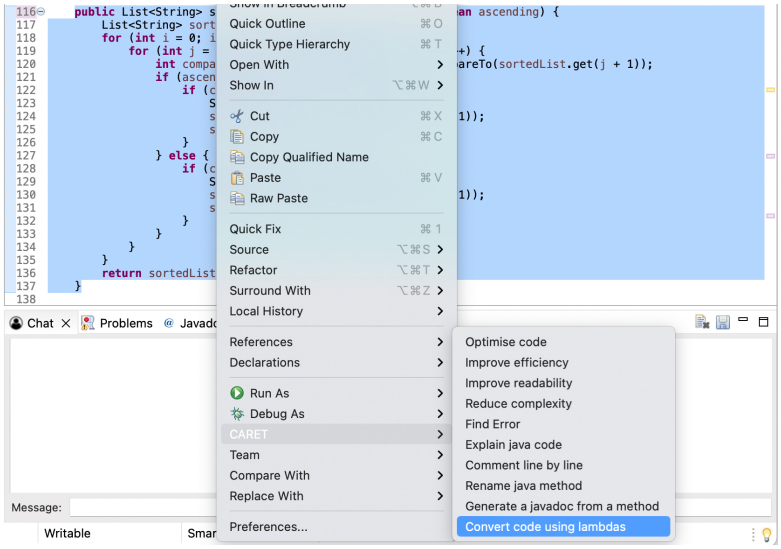


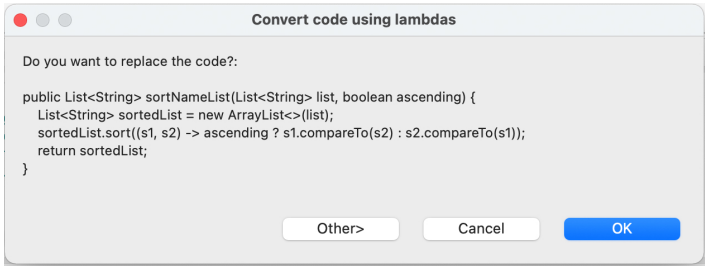
Fig. 12: Adding a new assistive task using the TaskGroup extension point.

list of CARET commands is displayed, which includes the task "Convert code using lambdas" (Figure 13a). When clicking on this command, CARET extracts the source code of the method, and sends a prompt that includes this code to the specified agent. The agent then responds with a code suggestion, which is presented to the user in a pop-up window (Figure 13b). If the developer accepts the suggested code, then it replaces the current code and is annotated with the interaction information (Figure 13c).

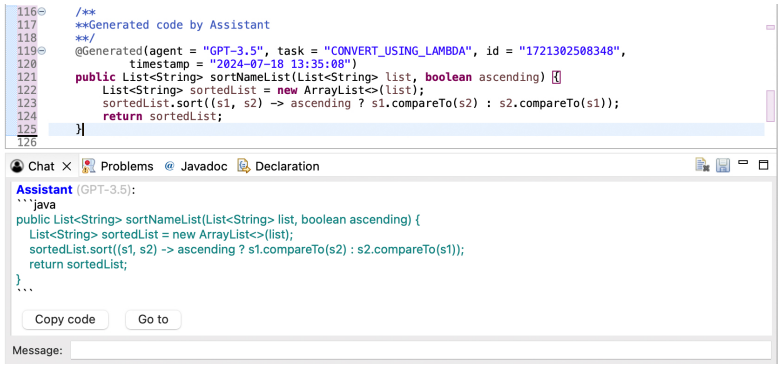
Finally, Figure 14 shows the configuration page of CARET. The field "Saved sessions folder" specifies the folder for storing the assistance sessions. The two lists below this field display the agents registered via the AgentTechnology extension point, and permit setting a priority order for their use as task classification agents and as task processing agents. This distinction is made to allow users prioritise a certain agent technology for task classification and use a different technology for task processing, so that they can choose the agents that best meet their needs. CARET uses this priority to decide the next agent to try in case a given one does not respond adequately or is not available.



(a) Contextual menu to apply new task on the selected method “sortNameList”.



(b) Pop-up window showing the code suggestion.



(c) The suggested code replaces the old one, and the method is annotated.

Fig. 13: Using the new assistive task to convert Java code into lambda expressions.

The last list in the configuration page makes it possible to configure which agents can provide content assistance, as well as the priority order in which they display their suggestions. Moreover, each registered agent can add a preference sub-page within the CARET preferences page, where users can customise the parameters of the agent.

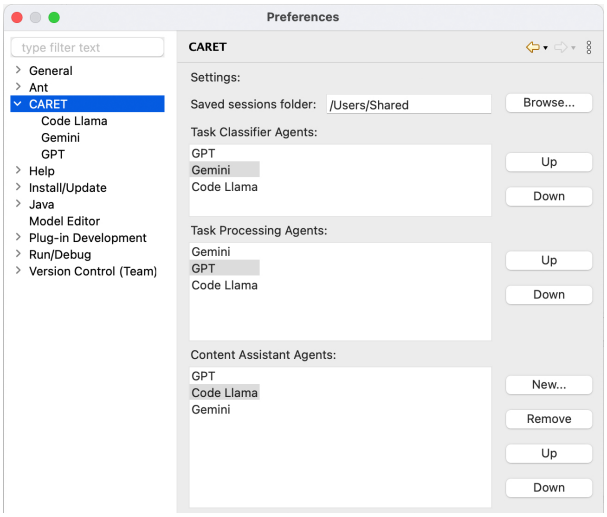


Fig. 14: Configuration of CARET.

6 Evaluation

This section evaluates the suitability of the assistance provided by CARET. Given the diverse range of tasks that CARET supports, we chose to evaluate a representative one – 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"* [12] – while inconsistent method names make the code difficult to understand and maintain [21]. As reviewed in Section 3, many 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 6.1 characterises the experimental setup, Section 6.2 describes the evaluation protocol, Section 6.3 analyses the results and answers the RQ, and Section 6.4 discusses the potential threats to validity.

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

6.1 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 (from [9]).

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 compilers[22], embedded vector search engines[23], vulnerability detection due to the use of Log4J[24], and a social network with a swing graphical user interface.

6.2 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 15.

We first selected four Java projects as explained in Section 6.1. 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

[22] https://github.com/wangjs96/A-tutorial-compiler-written-in-Java
[23] https://github.com/jbellis/jvector
[24] https://github.com/mergebase/log4j-detector

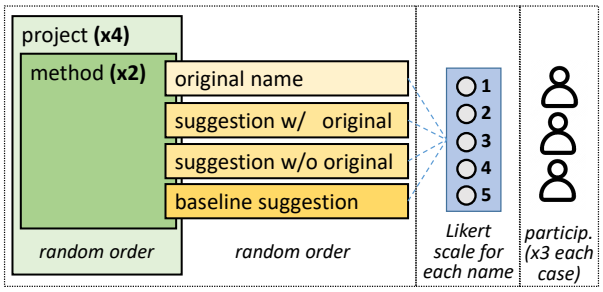


Fig. 15: Scheme of the experiment design (from [9]).

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 (original), 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 therefore 96 methods overall. The evaluation was conducted offline. Participants received the questionnaires by email and were given 5 days to submit their responses.

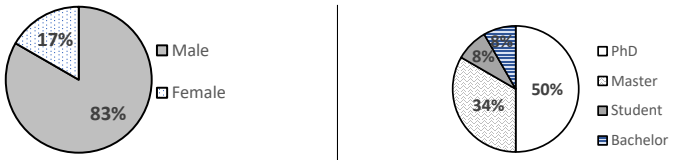
The questionnaires used for the evaluation are available on-line at: `https://github.com/caretpro/experiment`.

6.3 Results and answer to RQ

Demographics of participants. The age of the participants ranged from 21 to 41 years (31.9 years on average). Figure 16 summarises the collected demographic data. Regarding gender, 83% of participants were men and 17% were women. In terms of educational level, 50% had a PhD degree, 34% had a master's degree, 8% had a bachelor's degree, and 8% were undergraduate students.

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), which yield an average of 3.42. Hence, overall, the participants declared having good level of experience in software and Java development, and a fair knowledge of the Java language.

Evaluation results. Before analysing the responses to the questionnaires, it is worth stressing that all method names generated by CARET were valid (e.g., they do not start



(a) Gender (b) Studies

Fig. 16: Demographic data of participants (from [9]).

Experience	Average (years)
Software development	9.75
Java development	4.75

Table 2: Years of experience in development (from [9]).

with a number or special symbol) and followed the Java naming convention of being in lower camel-case (e.g., like aMethodName).

With regard to the questionnaires, the box plots in Figure 17 depict the distribution of scores that each method renaming strategy received. In the box plots, the series new1 corresponds to the assistant suggestions produced with a prompt that includes the original method name, and new2 to those produced omitting the original method name. As Section 6.2 explained, 3 participants evaluated each method. Thus, Figure 17(a) shows the distribution of the average score values of each method (i.e., 32 data points per series), and Figure 17(b) shows the distribution of all scores without averaging per method (i.e., 32×3=96 data points per series).

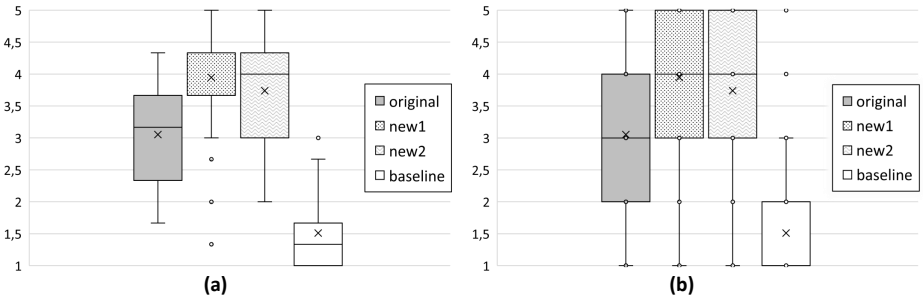


Fig. 17: Distribution of scores of the suggested method names. (a) Distribution of the averages of the three scores received by each method. (b) Distribution of all scores (i.e., without averaging per method). Figure from [9].

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,

by far. Furthermore, in Figure 17(b), the median of the scores for the original method names is 3, while for the two assistant-generated method names is 4.

Figure 18 shows the results disaggregated by project, for the average scores (as in Figure 17(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.

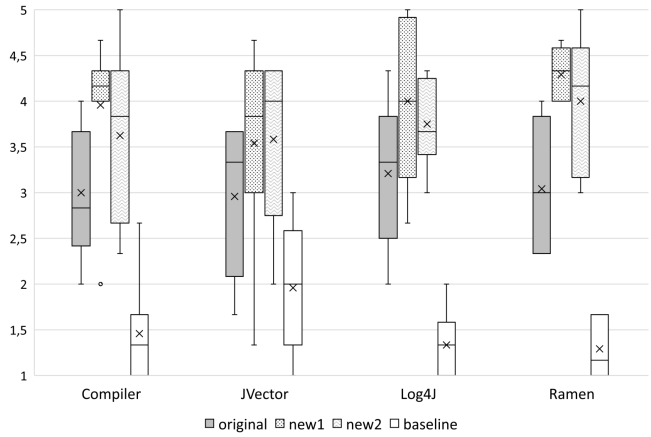


Fig. 18: Distribution of scores disaggregated by project (from [9]).

Now, we delve into the difference in score between the original method names and those suggested by the assistant. The left bar of Figure 19 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.

Finally, to analyse if the difference in scores of new1, new2 and original is statistically significant, we use the Wilcoxon Signed-Rank Test [39] 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 $W = 1264$, $Z_{(cal)} = -1.227$, $\alpha = 0.05$, $Z_{(\alpha/2)} = 1.96$,

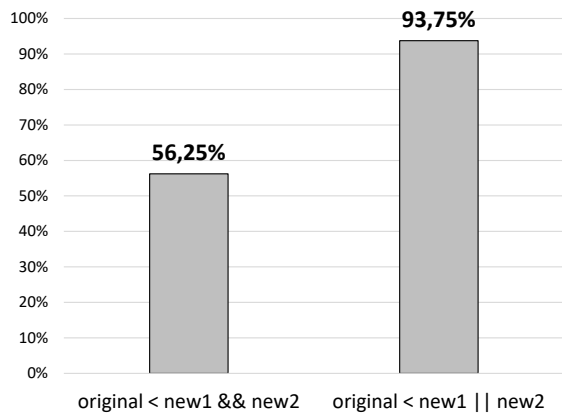


Fig. 19: Comparison of scores between the original method names and the assistant suggestions (from [9]).

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.

6.4 Threats to validity

Internal validity refers to the extent to which there is a causal relationship between the conducted experiment and the resulting conclusions. We attempted to avoid any bias in the experiment 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 execution by randomising the order of the projects and method names in the questionnaires, and by not revealing to the participants which mechanism was used to generate 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. This is a fair amount of data, but more evidence would be obtained with larger numbers 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, and other LLMs 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 [21]), 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 [11]. The level of agreement between the participants was between 0.2 and 0.4 in all projects which, according to [18], can be considered fair.

7 Conclusions and Future work

Intelligent conversational assistants will soon become an integral part of most development processes and environments [25]. With this expectation, we have explored the space of possibilities for their integration within IDEs. Then, we have proposed an extensible architecture that enables the incorporation of new agent technologies and assistive tasks externally, and proposed a traceability model and a coordination scheme for multiple conversational agents. We have realised our proposal within CARET, a conversational assistant for Java programming in Eclipse that helps in tasks such as code completion, code optimisation, documentation, and unit test generation. We have illustrated its extension with new tasks, and have conducted a user study of the supported method renaming task with very promising results.

We are currently working in the integration of new LLMs like Llama or Gemini into CARET, and extending CARET for its integration within the autocompletion facilities of Eclipse (i.e., activated with CTRL+*space* when writing code). We are also exploring proactive approaches to assistance, by supporting running the registered assistive tasks in the background. On the long term, our goal is to automate the creation of conversational assistants for other programming languages (e.g., Python or C++), domain-specific languages [38], Eclipse plugins, testing frameworks (e.g., Cucumber), or model-driven development (e.g., the Eclipse Modeling Framework [36]). Additionally, it would be possible to investigate the application of our architecture for assistance to low-code development platforms [33]. 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 [7].

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

References

1. Abdalkareem, R., Shihab, E., Rilling, J.: What do developers use the crowd for? A study using stack overflow. IEEE Softw. **34**(2), 53–60 (2017)
2. Alon, U., Zilberstein, M., Levy, O., Yahav, E.: code2vec: learning distributed representations of code. Proc. ACM Program. Lang. **3**(POPL), 40:1–40:29 (2019)

3. Bajpai, Y., Chopra, B., Biyani, P., Aslan, C., Coleman, D., Gulwani, S., Parnin, C., Radhakrishna, A., Soares, G.: Let's fix this together: Conversational debugging with GitHub Copilot. In: Proc. VL/HCC. IEEE (2024)
4. Barke, S., James, M.B., Polikarpova, N.: Grounded Copilot: How programmers interact with code-generating models. Proc. ACM Program. Lang. **7**(OOPSLA1), 85–111 (2023)
5. Bradley, N.C., Fritz, T., Holmes, R.: Context-aware conversational developer assistants. In: ICSE. pp. 993–1003. ACM (2018)
6. Brambilla, M., Cabot, J., Wimmer, M.: Model-driven software engineering in practice, 2nd edition. Synthesis Lectures on Software Engineering, Morgan & Claypool Publishers (2017)
7. Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H.P., Kaplan, J., et al.: Evaluating large language models trained on code. CoRR **abs/2107.03374** (2021), https://arxiv.org/abs/2107.03374
8. Chen, Y., Fu, Q., Yuan, Y., Wen, Z., Fan, G., Liu, D., Zhang, D., Li, Z., Xiao, Y.: Hallucination detection: Robustly discerning reliable answers in large language models. In: CIKM. pp. 245–255. ACM (2023)
9. Contreras, A., Guerra, E., de Lara, J.: Conversational assistants for software development: Integration, traceability and coordination. In: Kaindl, H., Mannion, M., Maciaszek, L.A. (eds.) Proc. ENASE. pp. 27–38. SCITEPRESS (2024)
10. Cummins, C., Seeker, V., Grubisic, D., Roziere, B., Gehring, J., Synnaeve, G., Leather, H.: Meta large language model compiler: Foundation models of compiler optimization (2024)
11. Fleiss, J.L.: Measuring nominal scale agreement among many raters. Psychological Bulletin **76**, 378–382 (1971)
12. Fowler, M.: Refactoring - Improving the Design of Existing Code. Addison Wesley object technology series, Addison-Wesley (1999)
13. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1 edn. (1994)
14. Gasparic, M., Ricci, F.: IDE interaction support with command recommender systems. IEEE Access **8**, 19256–19270 (2020)
15. Guerra, E.M., Cardoso, M., Silva, J.O., Fernandes, C.T.: Idioms for code annotations in the Java language. In: SugarLoafPLoP. pp. 7:1–7:14. ACM (2010)
16. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
17. Kelley, J.F.: An empirical methodology for writing user-friendly natural language computer applications. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems. p. 193–196. CHI '83, ACM, New York, NY, USA (1983)
18. Landis, J.R., Koch, G.G.: The measurement of observer agreement for categorical data. Biometrics **33**, 159–174 (1977)
19. Li, R., et al.: StarCoder: May the source be with you! CoRR **abs/2305.06161** (2023), see also https://huggingface.co/blog/starcoder
20. Lin, F., Kim, D.J., Chen, T.H.: When llm-based code generation meets the software development process (2024), https://arxiv.org/abs/2403.15852
21. Liu, K., Kim, D., Bissyandé, T.F., Kim, T., Kim, K., Koyuncu, A., Kim, S., Traon, Y.L.: Learning to spot and refactor inconsistent method names. In: ICSE. pp. 1–12. IEEE / ACM (2019)
22. Mazinanian, D., Ketkar, A., Tsantalis, N., Dig, D.: Understanding the use of lambda expressions in java. Proc. ACM Program. Lang. **1**(OOPSLA) (oct 2017)
23. Nazari, A., Swayamdipta, S., Chattopadhyay, S., Raghothaman, M.: Generating function names to improve comprehension of synthesized programs. In: Proc. VL/HCC. IEEE (2024)
24. Ozkaya, I.: Application of large language models to software engineering tasks: Opportunities, risks, and implications. IEEE Softw. **40**(3), 4–8 (2023)

25. Ozkaya, I.: The next frontier in software development: AI-augmented software development processes. IEEE Softw. **40**(4), 4–9 (2023)
26. Pérez-Soler, S., Guerra, E., de Lara, J., Jurado, F.: The rise of the (modelling) bots: Towards assisted modelling via social networks. In: ASE. pp. 723–728. IEEE Computer Society (2017)
27. Pérez-Soler, S., Juárez-Puerta, S., Guerra, E., de Lara, J.: Choosing a chatbot development tool. IEEE Softw. **38**(4), 94–103 (2021)
28. Recio Abad, J., Saborido, R., Chicano, F.: Naming methods with large language models after refactoring operations. In: Proc. JISBD (2024)
29. Rich, C., Waters, R.C.: The programmer's apprentice: A research overview. Computer **21**(11), 10–25 (1988)
30. Robe, P., Kuttal, S.K.: Designing PairBuddy – A conversational agent for pair programming. ACM Trans. Comput.-Hum. Interact. **29**(4) (may 2022)
31. Ross, S.I., Martinez, F., Houde, S., Muller, M., Weisz, J.D.: The programmer's assistant: Conversational interaction with a large language model for software development. In: IUI. pp. 491–514. ACM (2023)
32. Ross, S.I., Muller, M.J., Martinez, F., Houde, S., Weisz, J.D.: A case study in engineering a conversational programming assistant's persona. In: IUI. CEUR Workshop Proceedings, vol. 3359, pp. 121–129. CEUR-WS.org (2023)
33. Ruscio, D.D., Kolovos, D.S., de Lara, J., Pierantonio, A., Tisi, M., Wimmer, M.: Low-code development and model-driven engineering: Two sides of the same coin? Softw. Syst. Model. **21**(2), 437–446 (2022)
34. Savary-Leblanc, M., Burgueño, L., Cabot, J., Pallec, X.L., Gérard, S.: Software assistants in software engineering: A systematic mapping study. Softw. Pract. Exp. **53**(3), 856–892 (2023)
35. Sommerville, I.: Software engineering, 10th Edition. International computer science series, Addison-Wesley (2015)
36. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework, 2nd edition. Pearson Education (2008)
37. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. In: Guyon, I., von Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R. (eds.) Proc. NIPS. pp. 5998–6008 (2017)
38. Wasowski, A., Berger, T.: Domain-specific languages - Effective modeling, automation, and reuse. Springer (2023)
39. Wilcoxon, F.: Individual comparisons by ranking methods. Biometrics **1**, 196–202 (1945)
40. Xu, F.F., Alon, U., Neubig, G., Hellendoorn, V.J.: A systematic evaluation of large language models of code. In: MAPS@PLDI. pp. 1–10. ACM (2022)
41. Xu, F.F., Vasilescu, B., Neubig, G.: In-IDE code generation from natural language: Promise and challenges. ACM Trans. Softw. Eng. Methodol. **31**(2) (mar 2022)
42. Yang, Y., Xia, X., Lo, D., Grundy, J.C.: A survey on deep learning for software engineering. ACM Comput. Surv. **54**(10s), 206:1–206:73 (2022)
43. Zhang, J., Luo, J., Liang, J., Gong, L., Huang, Z.: An accurate identifier renaming prediction and suggestion approach. ACM Trans. Softw. Eng. Methodol. **32**(6), 148:1–148:51 (2023)
44. Zhao, W.X., et al.: A survey of large language models. https://arxiv.org/abs/2303.18223 (2023)
45. Zheng, Z., Ning, K., Wang, Y., Zhang, J., Zheng, D., Ye, M., Chen, J.: A survey of large language models for code: Evolution, benchmarking, and future trends (2024), https://arxiv.org/abs/2311.10372