# Interactive & Adaptive LLMs

## Building an LLM plugin for JetBrains IDEs and evaluating a novel gray-text code completion style

Frank van der Heijden


Delft University of Technology

TU Delft

# Interactive & Adaptive LLMs

## Building an LLM plugin for JetBrains IDEs and evaluating a novel gray-text code completion style

by

Frank van der Heijden

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on Wednesday July 10, 2024 at 1:00 PM.


| | | |
|---|---|---|
| Student number: | 5102472 | |
| Project duration: | November 13, 2023 – July 10, 2024 | |
| Thesis committee: | Prof. dr. A. van Deursen, | TU Delft, chair |
| | Assistant Prof. dr. M. Izadi, | TU Delft, supervisor |
| | Assistant Prof. dr. U. Gadiraju, | TU Delft |



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

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.


TU Delft

# Preface

*A preface...*

*Frank van der Heijden*
*Delft, July 2024*

# Summary

In this thesis, we have explored the landscape of Large Language Model (LLM) plugins, focusing on their integration into JetBrains IDEs. We began by examining the current state of these plugins, from an early code completion tool Code4Me to the more sophisticated, interactive assistants of today. We then delved into the creation of a reusable LLM plugin and backend, detailing the design choices, architecture, and deployment strategies employed. The backend, built with Python and Django, serves as the backbone for the plugin, handling API requests, user management, and data storage. The plugin itself, developed using Kotlin and the JetBrains Plugin SDK, offers features such as an LLM chat, code completion, and customizable templates.

A key aspect of this thesis was a user study conducted at JetBrains, investigating a novel gray-text code completion style that leverages the IDE's static analysis. The study aimed to assess the usefulness and usability of this new approach, comparing it to the traditional gray-text completion. Results indicated that while the novel method showed promise in terms of accuracy and edit similarity, it scored lower on the System Usability Scale, suggesting a need for further refinement and user familiarization.

The challenges encountered during the development and deployment process, such as transitioning between LLM clusters and refining the code completion API, were also discussed. These challenges highlight the complexities involved in integrating LLMs into real-world development environments and underscore the importance of ongoing research and development in this field.

In conclusion, this thesis contributes to the growing body of knowledge on LLM plugins for IDEs. It provides a detailed account of building and deploying such a plugin, offers insights into a novel code completion approach, and presents the results using an A/B user study. The work presented here serves as a foundation for future research and development, creating a way for more interactive, adaptive, and user-friendly LLM tools for developers and researchers in the future.

# Nomenclature

## Abbreviations

| Abbreviation | Definition |
|---|---|
| ASGI | Asynchronous Server Gateway Interface |
| CD | Continuous Deployment |
| CI | Continuous Integrations |
| CLI | Command-Line Interface |
| CRUD | Create, Read, Update and Delete |
| DOM | Document Object Model |
| DRF | Django REST Framework |
| DSL | Domain-Specific Language |
| EM | Exact Match |
| ES | Edit Similarity |
| FLCC | Full-Line Code Completion |
| FP | Functional Programming |
| IDE | Integrated Development Environment |
| JSON | JavaScript Object Notation |
| JVM | Java Virtual Machine |
| LLM | Large Language Model |
| NDA | Non-Disclosure Agreement |
| NLP | Natural Language Processing |
| OOP | Object-Oriented Programming |
| OSS | Open Source Software |
| QA | Question and Answer |
| RAG | Retrieval-Augmented Generation |
| REST | REpresentational State Transfer |
| SAU | Specific Aspects of Usefulness |
| SDK | Software Development Kit |
| SES | Simple Email Service |
| SSE | Server-Sent Events |
| SUS | System Usability Scale |
| TCP | Transmission Control Protocol |
| TTFB | Time To First Byte |
| UML | Unified Modeling Language |
| VCS | Version Control System |
| WSGI | Web Server Gateway Interface |
| XML | Extensible Markup Language |

# Contents

# List of Figures

# List of Tables

# 1

# Introduction

Large Language Models (LLMs) have revolutionized the field of Natural Language Processing (NLP) with their ability to understand and generate human-like text. Recent advancements have led to the development of LLMs specifically designed for code generation and developer assistance. These models, such as CodeLLama, StarCoder, and InCoder, have shown promise in automating code completion, documentation generation, and error detection, thereby streamlining the coding process and improving code quality.

However, the practical application of these LLMs in real-world development environments, particularly within Integrated Development Environments (IDEs), presents unique challenges and opportunities. This thesis explores these challenges by developing a reusable LLM plugin and backend for JetBrains IDEs. The plugin integrates both code completion and chat functionalities, leveraging the capabilities of LLMs to enhance developer productivity.

Another key contribution of this thesis is the evaluation of a novel gray-text code completion style that aims to improve the alignment between LLM suggestions and developer intentions. This is achieved by incorporating the IDE's static analysis suggestions as prefixes to the LLM's code completions. The effectiveness and usability of this approach are assessed through a user study conducted at JetBrains, providing valuable insights into the practical implications of integrating LLMs into real-world development workflows.

The thesis is structured as follows: chapter 2 provides background information on LLM plugins, previous user studies, and a related experiment called Code4Me. Chapter 3 details the creation of a reusable LLM plugin and backend, discussing the architecture, design choices, and deployment strategies. Chapter 4 presents a user study conducted to evaluate the novel gray-text code completion style, outlining the research questions, methodology, and results. Chapter 5 discusses the challenges encountered during the development and deployment of the plugin. Finally, Chapter 6 concludes the thesis, summarizing the findings and contributions, and Chapter 7 proposes potential future work to further enhance the plugin and its capabilities.

# 2

# Background

During my bachelor project, I contributed to Code4Me[1], which was a plugin meant to test out pre-trained models in the wild. The models that were run in the backend were CodeGPT, UniXcoder, and InCoder. These were locally hosted at TU Delft, and run in parallel to generate multiple completions per triggerpoint. A triggerpoint could either be one of the pre-defined set of characters the model should be invoked upon (such as a period or opening parenthesis, which are common points in code where a code completion would be useful), or a manual invocation using a keybind. Finally, the plugin was set to collect data automatically, and it achieved this by tracking the line the code completion was inserted in. After 30 seconds, the same line was sent to the backend (starting from the offset the completion was performed), which would then be stored as a ground truth for the completion.

The plugin was built in the spring/summer of 2022, a time at which not many APIs were available for code completion plugins. It was built on top of the existing editor suggestions, which mixed the output of the models with the suggestions of the IDE from a static analysis of the project. Furthermore, the plugin did not offer any LLM chat functionalities, a feature which has been added to many code completion IDE plugins.

## 2.1. Large Language Model Plugins

Large Language Model (LLM) plugins are an emerging technology designed to integrate advanced language models into various software applications. These plugins leverage the capabilities of LLMs, either locally hosted or externally hosted, to provide enhanced functionalities, particularly in the realm of code completion and development assistance.

LLM plugins operate by embedding pre-trained language models into integrated development environments (IDEs) or other software tools. These models are capable of understanding and generating human-like output, which makes them particularly useful for tasks such as code completion, documentation generation, and error detection. The primary aim of these plugins is to streamline the coding process, reduce the time developers spend on routine tasks, and improve code quality.

### 2.1.1. Code Completion Plugins

Code completion plugins typically function by monitoring the developer's input in real-time. When a developer reaches a trigger point—such as typing a period, an opening parenthesis, or invoking a specific keybind—the plugin activates the LLM to generate code completions or suggestions. These trigger points can either be strategically chosen to coincide with moments when developers are most likely to need assistance, or invoked by the IDE automatically. With the new APIs JetBrains offers, the IDE is now in charge of these trigger points. Triggers for code generation are now invoked at the points the IDE deems it best to support the developer. IDEs are able to use language-specific information to determine the best trigger points, or because it noticed the developer stopped typing for a second and

[1] `https://code4me.me`

might need some asssistance.

The plugin sends the current context of the code to the model (either externally or locally), where the LLM processes it and returns a possible completion. The context it uses for this could range from the current method the user is working in, to the currently working file, or even an collection of open files that could be helpful for generating a code completion. Some models are only able to work with the left context (i.e. start generating from the input text), and some models are able to infill between two parts of the code, the part before the generation, and the code that might already exist after the completion (left and right context). These completions are then presented to the developer, who can choose to insert it, or ignore it and move on with their original idea. Having IDE completions could speed up developers in writing code.

Recently, JetBrains also introduced a local model which is now included in the IDEs by default, called "Full Line Code Completion" (FLCC). This model is smaller than the models that are hosted externally, but has been shown to be quite effective and internal metrics show it has a high acceptance rate, despite it being much smaller than externally hosted models. Unfortunately the exact numbers for the metrics concerning FLCC are under a JetBrains Non-Disclosure Agreement (NDA).

### 2.1.2. Chat Plugins

After the success of code completion plugins, a lot of plugins have been integrating a chat functionality in their feature suite as well. These plugins leverage LLMs trained for chat and QA, to provide a conversational interfaces directly within the development environment. This enables developers to interact with the model in a more intuitive and natural manner, similar to conversing with a coworker or student.

The primary function of chat plugins is to facilitate various coding tasks through natural language interaction. Developers can ask questions about code syntax, request explanations for specific pieces of code, ask suggestions for refactoring, or even discuss about best practices.

The chat functionality can be integrated with the IDE as well, depending on the context window or capabilities of the LLM. Context to a prompt could fully be provided by the initatior of the chat (developer or action that triggered a new chat), but could also be aided by an external database. One could index the currently worked on project in the IDE, and use Retrieval Augmented Generation (RAG) within the LLM models to query useful external information to best assist to a user's prompt.

Chat plugins can be designed to integrate seamlessly with other tools and features within the IDE. The chat could be started from a number of starting points within the IDE, such as for example by selecting a piece of code providing the baseline for the chat, or for example in a Version Control System (VCS) by asking for a fitting git commit message.

## 2.2. User Studies on LLM Plugins

The evaluation of LLM plugins is crucial to understanding their usefulness and usability in real-world programming environments. They typically aim to measure how effectively these plugins improve coding efficiency, reduce error rates, and enhance the overall user experience. This can for example be achieved using automated metrics collected in the background, and a user survey at the end after using the plugin.

Automated metrics typically include some form of an accuracy of the completions provided, and an edit similarity to the actual desired completion. Combining these two metrics typically give a good insight into how well the code LLM understands the context, and how well it has provided the developer with an accurate completion.

The survey is used for collecting nuanced information about the plugin, such as the usability. This type of information is a lot harder to collect through automated metrics – developers might accept code completions, but don't particularly like the plugin in itself and might have improvements for it, which can all be included in such a survey.

Findings from these studies generally highlight a positive impact on coding efficiency and a reduction in simple coding errors. From previous works, developers often report that LLM plugins help them write

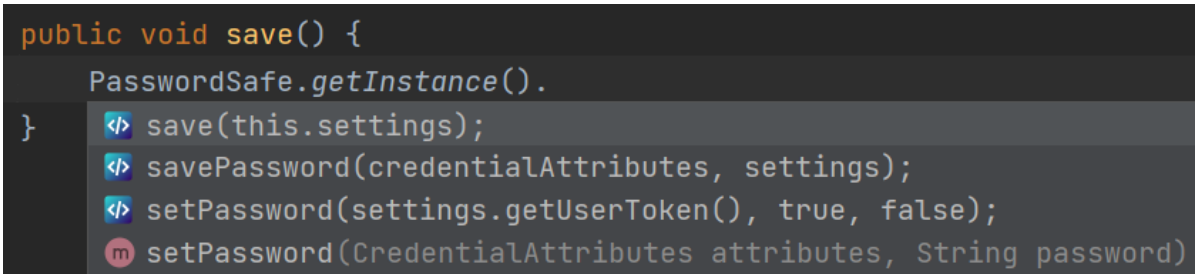



**Figure 2.1:** Code4Me's code completion style using the native JetBrains popover code recommendation system.[2]

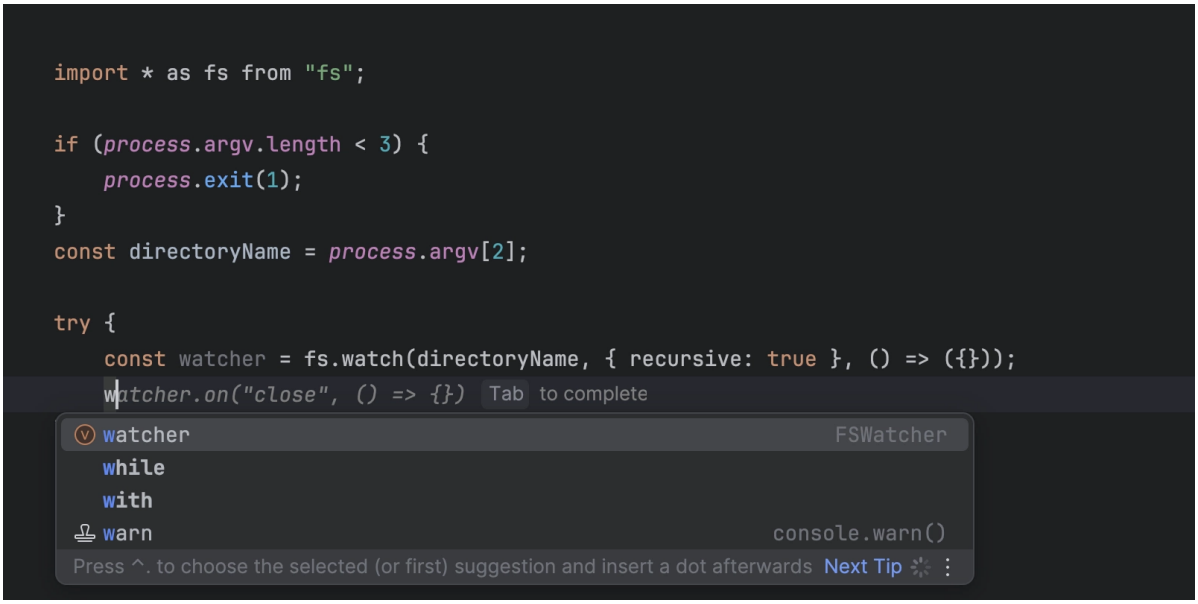



**Figure 2.2:** Gray text code completion style.[3]

code faster and with fewer interruptions, although some also report a learning curve for all these new tools. Insights often include suggestions for improving the context-awareness of the models and enhancing the integration of these tools into existing IDEs to make them more intuitive and less intrusive.

Despite the benefits, user studies also often reveal areas for improvement. Common challenges include the handling of complex coding tasks that require deeper contextual understanding (e.g. context in different files, or business logic), and the integration of LLMs into diverse coding environments.

## 2.3. A Previous Experiment: Code4Me

Two years ago, a code completion plugin for code was developed by Marc Otten, Jorit de Weerdt and me, as part of our bachelor thesises. The plugin featured code completion using the IDEs native code recommendation system at the time, whereas the common way of displaying code completions nowadays is using a so-called "gray-text code completion style". Figure 2.1 and Figure 2.2 display the style of Code4Me and the gray-text code completion, respectively.

The user study focused around evaluating the theoretical performance of code LLMs in a real world scenario, since most trained models are only evaluated using automated benchmarks and a test set very similar in tasks to the train set. Exposing these models to the real world would gain insights in how the code completions are used and perceived by developers.

The models that were used were CodeGPT, UniXcoder, and InCoder; all relatively "small" models in a sense that each could be run on a single GPU. The deployment of this backend was fairly simple,

[2] Image from "Language Models for Code Completion: A Practical Evaluation" by Izadi et al.
[3] Image from the blog post "Full Line Code Completion in JetBrains IDEs: All You Need to Know" by E. Ryabukha

a REST API was made around the HuggingFace inference library such that completions could be generated from the IDE plugin, and only a single deployment was used for all the API calls.

Code4Me supported multiple IDEs, VSCode and all of the JetBrains IDEs up until a few versions ago, which allowed for an extensive research across a large userbase with a lot of completion inferences in its dataset.

# 3

# Creating a Reusable LLM Plugin and Backend

Continueing on Code4Me with the goal in mind to create a better and reusable codebase for an LLM experiment in the IDE, a new plugin was written from the ground up using Kotlin and the latest APIs available from the JetBrains Plugin SDK. JetBrains has introduced new APIs in the past year for code completion plugins, including a full-fledged gray text code completion API. This code completion style seems to be the main style of code completion that combines well with the existing dropdown completions from the IDE, as this style is used by many commercial entities such as GitHub CoPilot, Supermaven, and JetBrains' own Full Line Code Completion (FLCC). The new plugin also features a chat functionality, which has been built natively into the plugin using the JetBrains UI APIs. The codebase has been split up into two main parts: the plugin that is actually embedded within the JetBrains IDE, and the plugin backend. The following sections will dive deeper into the backend and plugin's architecture, design choices and deployments.

## 3.1. Plugin Backend

The plugin backend's main purpose is to serve the APIs to be used by the embedded plugin and to host the signup website, where potential users for the user study can join the study. The design of this backend is made to be extended upon and focused on future iterations of the codebase, on both the maintainability and extensibility side. The code is split among logical components, using a standardised directory structure to find and work on components easily. Best practises are also applied for streaming LLM responses by implementing Server Sent Events (SSE).[1] The following sections will dive deeper into each component.

### 3.1.1. Language and Framework

Choosing the main language is directly tied with the extensibility and maintainability for future iterations of the project and codebase. If a language is chosen that's not widely adopted, this could pose a challenge in the future to find developers that are capable of working with such a codebase. On the other side, the language should also support all the intentions for future iterations of the project, and should not impose a limitation later down the road. Similarly for the framework that guides the main structure of the codebase, a well-documented, battle-tested, and popular framework should be chosen that can assist in all the needs for creating a reusable and purpose-built backend. In the following sections we will explain why the language Python, in combination with the framework Django and the frontend framework React were chosen. This frontend framework is not to be confused with the LLM Plugin: this type of frontend serves the signup website.

[1] `https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-events`

### The Python Programming Language

The chosen main language for the backend is Python. Python has great community support, an extensive standard library, and an extremely large amount of packages that can be imported. The latest iteration of the TIOBE Programming Community Index lists Python as the most popular programming language, and it has only been growing in popularity over the past years [2].[2]

Choosing Python for NLP and interacting with LLMs is also a very common choice, as a significant platform of pre-trained LLMs, HuggingFace,[3] creates libraries for Python. For future iterations of the backend, the transformers library can be easily imported and used in the codebase in order to use pre-trained models and datasets from HuggingFace.

To support best practises with the Python language and benefit from the IDEs smartness while developing, proper typings have been added where possible in the codebase to minimize errors that could easily be avoided by having an extra warning when using the wrong types. This is especially helpful when a newcomer deals with an unknown codebase: having no types leaves the developer guessing to what a method actually does, which types of input it has, and what it outputs.

### The Django Framework and Django Rest Framework

In the Python ecosystem a lot of libraries exist for creating APIs. For this project, Django was chosen as the main framework. Django is a very popular framework as well, and ranks highly in many charts on the web.[4][5] The Django framework comes with a vast amount of features baked into it, such as data models, database migrations, an admin section, authentication and authorization for users, and views and serializers for mapping data models to those views.

Data models are useful for creating Python-managed database objects, which can be queried, created and interacted upon from the source language (Python). It is built from a declarative model standpoint, which means that very little interaction is needed to the underlying database from the actual developers perspective. The Python model files are written, and Django automatically generates the underlying queries for creating the database tables (or documents in the case of a NoSQL database), creating the actual objects, quering, updating and deleting. Relations among objects are also directly communicated in this declarative style: Django offers relational constraints such as one-to-one, one-to-many, and many-to-many to efficiently communicate these.

A database naturally changes over time, this could for example be because of new requirements, enhancements, or bugfixes. Hence, it is necessary to have good support for schema changes in the chosen framework. Django offers database migrations right out of the box, which can almost always be automatically generated by the Command Line Interface (CLI), by looking at the previous state of the database, and comparing that to the newly written models in Python. A diff-check will result in the changes required to convert the old state, into the new state. Some changes may be hard to automatically detect, such as column value changes or the renaming from one column to another one with a similar datatype. Therefore, manual written migrations are always a possibility within the Django framework, as well as modifying an automatically generated migration.

Since the Django framework doesn't automatically include a REST framework to create an API for interacting with the LLMs, the Django Rest Framework (DRF) was chosen.[6] This framework neatly integrated with the Django framework, allowing for the creation of views and serializers. Views are essentially the classes that expose certain actions on objects, i.e., Create, Read, Update, and Delete (CRUD) operations. These operations can be defined as Mixins on the class. Each mixin is a parent class that defines a certain operation (i.e. one from CRUD, or a custom one). It also allows custom operations, with a custom REST url endpoint, which is used for generating custom actions. In this project's case, custom actions are used for the endpoints that expose an SSE-resource. Serializers are the classes that map a database object to a JSON response, it exposes the ability to only return the fields that are necessary for the endpoint. In other cases, some fields should never be returned,

[2] The TIOBE Programming Community Index is an automated popularity index, massively collected from search queries of the major search providers. The definition is published on the TIOBE website.

[3] `https://huggingface.co`

[4] `https://lp.jetbrains.com/django-developer-survey-2023/`

[5] `https://6sense.com/tech/web-framework`

[6] `https://www.django-rest-framework.org`

such as the user's hashed password or token. Furthermore, using the DRF library, we can generate an OpenAPI schema of all the API endpoints, which can then be used on the Plugin's side to generate endpoints.[7] This schema provides a standardised way of publishing the available endpoints for an API, as well as any inputs and outputs.

#### The React, TypeScript and Mantine frontend

It should be easy for users to join the user study, any additional step for joining the study at the start may result in a deduction of users willing to join the study. Hence, it is important to have an easy-to-use and quick sign-up process for the user study.

The framework to create this frontend in is React,[8] a very popular and known framework for writing websites. Creating the frontend in React has the benefit of being able to use JavaScript on the frontend, a language made for interacting with the Document Object Model (DOM). React creates a layer on top of the DOM, called the "Virtual DOM", which will be re-rendered into the actual DOM whenever a property has changed in the code. This allows for very minimal, but reactive code, which can help build pretty sites with minimal effort. React files are written in JSX, a language that allows combining HTML elements inside the JavaScript file.[9]. These JSX files (commonly ending with the `.jsx` extension), are then compiled by the React compiler into bundles of JavaScript chunks. These chunks, in combination with an empty HTML file where these chunks are loaded in using `<script>` tags, make sure the website is populated with components and styling.

In addition to React, the TypeScript language was chosen for the same reasons as using typings in Python.[10] TypeScript is a superset of the JavaScript language (in other words, all JavaScript is valid TypeScript, but not the other way around), and is a transpiled language that at runtime is transformed into JavaScript. TypeScript has a very expressive typing system, they can consist for example of expressions, conditionals, recursive definitions, and constants. The compiler can be made very strict, which disallows the `any` type (similar to defining a variable as `java.lang.Object` in Java). TypeScript works well with React too, and the syntax language is called TSX. It works the same way as JSX files, and it will also be transpiled into JSX before handing it over to the React compiler.

Finally, to make React development a little bit easier, the Mantine framework was chosen for common components.[11] HTML exposes a few standard components, but these are extremely basic and can lack a ton of features. They can also be extremely offsetting because each browser may implement their own version of the specification. This can result in some browsers having a very good implementation, which exceeds expectations according to the spec, while others are lacking [9]. In addition to fixing smelly HTML components, Mantine offers a bunch of common components that developers have found useful and implementing over and over for different projects. This ranges from many input components, to overlays, common ways of presenting content, or even a rich text editor. By using a framework for common components, it can greatly speed up development, and has the benefit that most of these components expose neat APIs for interacting with the inputs/outputs and customizations.

### 3.1.2. Database Models

The database that is used in the backend is the relational database PostgreSQL (Postgres).[12] Among professional developers, according to a survey taken by StackOverflow among 76634 developers, Postgres is the most popular database used for development and production environments [1]. The database contains a total of 17 tables, of which 10 are automatically created by the Django framework. These tables contain information about the basic functionality the framework provides, such as its admin features, users, user roles, and user permissions. The other 7 tables, are created by the 7 data models we use for the user study. These are:

- `Chat`: This model contains the basic fields required for a chat: The `name`, the `user` that created the chat, the `system_message` (or in other words: system prompt), and a `created_at` timestamp.

[7] `https://www.openapis.org`
[8] `https://react.dev`
[9] `https://react.dev/learn/writing-markup-with-jsx`
[10] `https://www.typescriptlang.org`
[11] `https://mantine.dev`
[12] `https://www.postgresql.org`

- `ChatMessage`: Each chat message contains a `chat` object reference, a `role` (*System*, *Assistant*, *Function*, or *User*), the `message`, a `temperature`, and a `created_at` timestamp.
- `ChatMessageRating`: A rating is for each `chat_message`, with a `rating` (*Good*, *Neutral*, or *Bad*), and a `created_at` timestamp.
- `Completion`: Consists of a `prefix`, `suffix`, the actual `completion` from the LLM, a reference to the `user`, an optional `external_id`, and both a `created_at` and `updated_at` timestamp.
- `CompletionAcceptance`: When a completion is accepted, the `completion` reference is stored, as well as the `shown_period_millis` for how long it was shown, and the `created_at` timestamp.
- `CompletionChange`: For each `time_period` (*Ten Seconds*, *Thirty Seconds*, and *One Minute*), the `edited_completion` is stored, the `completion` reference, and the `created_at` timestamp.
- `Signup`: Each signup stores the `user` reference, `job_roles`, `job_role_other` in case other was specified, `employment_status`, `employment_status_other` in case their employment status was not present in the dropdown, `coding_experience`, whether they accept the `terms_of_service`, and a `created_at` timestamp.

Models are written using abstractions over the database. It combines the way of writing Python code effortlessly without conforming to a specific database implementation. One of the great features is that enums can be made easily as well, imposing constraints on character text fields. The `ChatMessageRating` Python class can be found in Listing 3.1, which showcases a Python enum for the possible ratings, and a relational constraint to the `ChatMessage` object. Furthermore, an overview of how these models relate to each other can be see in Figure 3.1. This graph was generated using the `django-extensions` package, in combination with Graphviz.[13][14] Most of the automatically generated tables by Django are omitted in this overview, but the `User` model is included as that's being referenced by multiple of our own tables.

**Listing 3.1:** The Django model for `ChatMessageRating`

```python
from django.db import models


class ChatMessageRating(models.Model):
    class Rating(models.TextChoices):
        """
        Rating enum
        """
        GOOD = "GOOD"
        NEUTRAL = "NEUTRAL"
        BAD = "BAD"

    chat_message = models.OneToOneField(
        "ChatMessage",
        on_delete=models.CASCADE,
        primary_key=True,
    )
    rating = models.CharField(choices=Rating)
    created_at = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.rating
```

Each model is then mapped at runtime by Django onto the Postgres database, and whenever a change is made to these models the following command can be ran in order to automatically generate a new set of migrations:

**Listing 3.2:** Command to generate Django migrations

```
$ python manage.py makemigrations api
```

And another command to apply the current set of migrations (one can modify the migrations after running the above command if changes are required):

[13] `https://django-extensions.readthedocs.io/en/latest/graph_models.html`
[14] `https://graphviz.org`

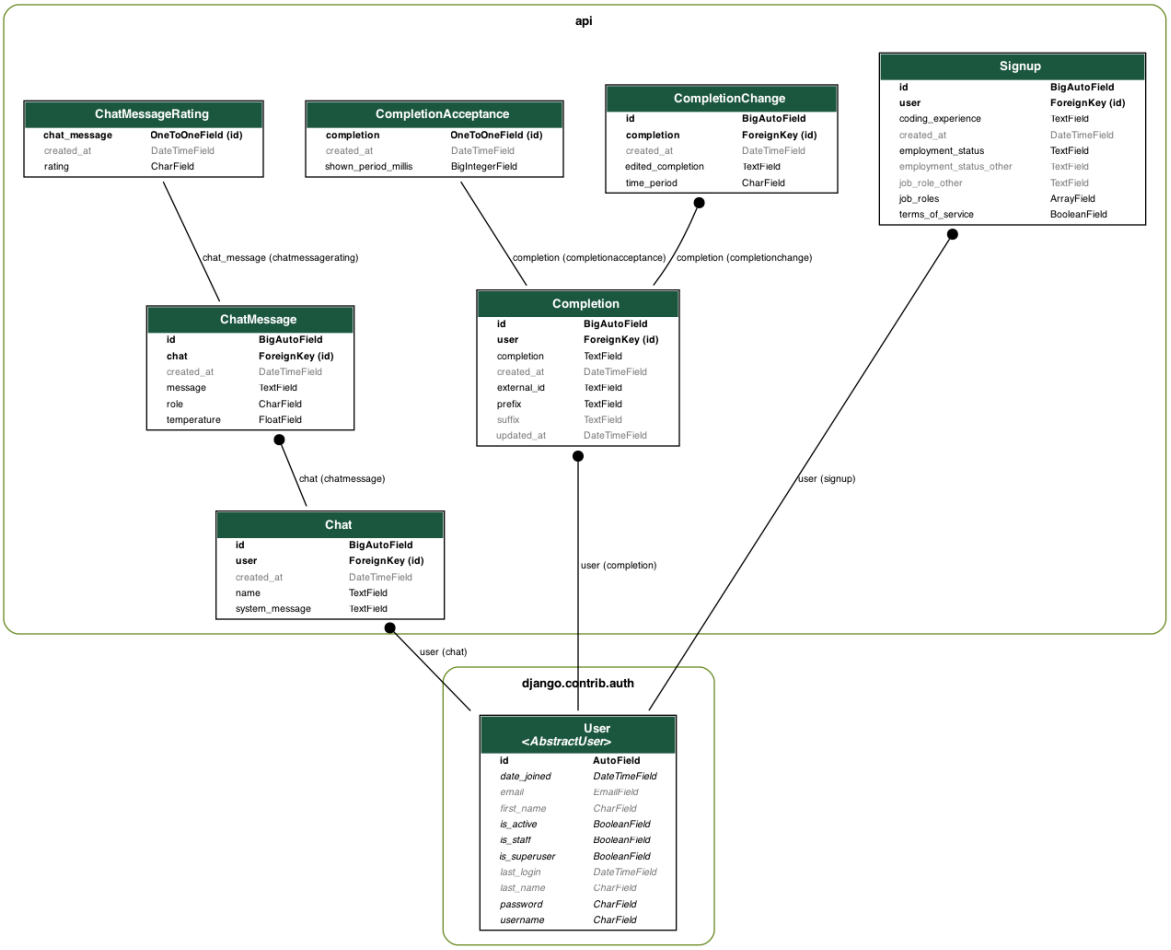



**Figure 3.1:** A graphical overview of the models used in Django.

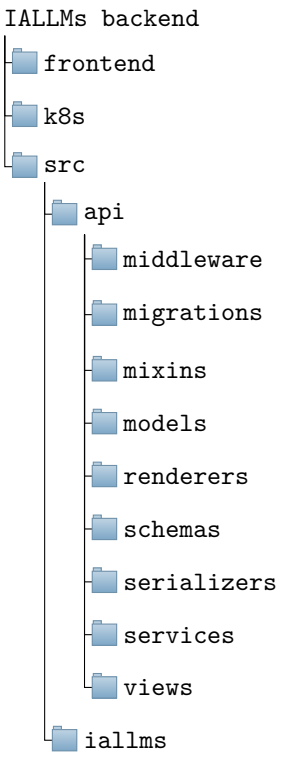



**Figure 3.2:** The file structure of the IALLMs backend

**Listing 3.3:** Command to apply Django migrations

```
$ python manage.py migrate
```

The last command executes each migration which hasn't been ran before, and inserts a row into the `django_migrations` table when it has been successfully executed.

### 3.1.3. Architecture Design

The backend has been split into logical components to ensure maintainability, readability and navigatability throughout the codebase. These components can be identified by the directory structure of the backend, which has been displayed in Figure 3.2.

The `frontend` folder contains the React project. This can be seen as a standalone project – it can be compiled and worked on independently. Later, the frontend is combined with the actual backend code, and served through the static files with the DRF. The `k8s` folder contains the files concerned with the Kubernetes deployment: it contains files for deploying the application on a Kubernetes cluster, and a "kube job" that will run the database migrations before the actual new version of the app boots. In subsection 3.1.4 the full CI/CD and combination of these projects will be explained.

Inside the `src` directory is where the Django codebase for the backend is present. The subdirectory `iallms` contains the Django root project, it is the coordinator for Django submodules. Django promotes to work with submodules: a feature that allows you to create any logical split among the components. For example, if you had a service specifically dealing with authentication, one could split that up in it's own submodule. Since this project is still relatively small compared to large enterprise code, it was chosen to use a single submodule `api`, to serve the backend APIs for the LLMs.

Inside this `api` folder, the DRF plugin is concerned with the `middleware`, `mixins`, `renderers`, `schemas`, `serializers`, and `views`. Middleware is being used for creating functions that should be executed on multiple endpoints. This could for example be code that should authenticate users for a set number of endpoints, the middleware could verify whether the user has permission for a resource, and if they do, allow the request to propagate to the next middleware (or router), or deny the request and break out of

the chain. The flow of a REST request through the code is essentially using the Chain of Responsibility design pattern. Mixins are classes that define logical operations (such as the ones from the CRUD abbreviation) / subroutes. A mixin could for example implement the *POST /<resource>* handler, which could define the “Create Resource” action. Another mixin could implement *DELETE /<resource>/<id>*, which could delete a particular resource.

These mixins together can be extended in a view located in the `views` directory, in which mixins are the parent classes of the view class. Custom actions can also be defined in views, similar to mixins. In our case, the endpoints for generating a LLM chat completion or code completion are custom actions, where we use SSE to efficiently transmit these long-taking requests in small chunks. The benefit of SSE is that the Time-To-First-Byte (TTFB) is low, and the user can immediately see that their request has succeeded. Instead of waiting for the entire response to be generated for a chat (this could take several seconds), the response is split into chunks which can contain a few tokens the LLM model generated, and are sent individually (but using the same TCP connection) to the client. This allows for an efficient and user-friendly way of generating lengthly sequential data.

The way SSE works is by opening a long standing TCP connection. This connection is kept open until a close request is initiated by the server. Data is sent through a standard text-based syntax, using data tags. These data tag typically contain a minified one-line JSON object, which can be parsed on the client. Data tags can be tagged with an event tag, which can indicate what kind of data is being transmitted. If only one datatype is used, typically the event tags are omitted. An example of a SSE request to the append chat message api endpoint can be seen in Listing 3.4. Each data tag is suffixed with two newline characters, this indicates a chunk has ended and the handler can begin processing this chunk while awaiting a new chunk. Furthermore, we explicitly indicate the end of the event stream by sending a *finish* event, which contains data of the chat message after it has been stored into the database, such as the message id. This id is only known after the message has been inserted into the database due to the autoincrement feature, so we can send this as a final message.

**Listing 3.4:** An SSE request made to the append chat message endpoint.

```
data: {"chunk": "", "function_call": null, "updated": null, "spent": null}

data: {"chunk": "1", "function_call": null, "updated": null, "spent": null}

data: {"chunk": "+", "function_call": null, "updated": null, "spent": null}

data: {"chunk": "1", "function_call": null, "updated": null, "spent": null}

data: {"chunk": "␣equals", "function_call": null, "updated": null, "spent": null}

data: {"chunk": "␣", "function_call": null, "updated": null, "spent": null}

data: {"chunk": "2", "function_call": null, "updated": null, "spent": null}

data: {"chunk": ".", "function_call": null, "updated": null, "spent": null}

data: {"chunk": "", "function_call": null, "updated": null, "spent": null}

event: finish
data: {"message_id": 53}
```

The folder `renderers` and `schemas`, contain a class that specifically deals with event streams, and is able to communicate that to the DRF plugin to generate the correct OpenAPI schema. Since SSE endpoints aren’t typically typed (arbitrary JSON data packets, and events), a good schema should clearly indicate the types of packets that are sent through SSE. As mentioned in a previous section, the `serializers` map the database objects to a JSON object which can be returned in the API. This serializer can also introduce relations that should be included in the JSON response, such as the full `User` object from a `ChatMessage` model. It is also possible to just hyperlink to the resource, or simply provide a numerical id for this relation to avoid overfetching and returning data that isn’t immediately necessary for the client. For example, on each `ChatMessageRating`, we do not need to include the full `Chat` or `ChatMessage` model, since in the case of giving a rating, the client should already know about which chat message they’re giving a rating for.

Another directory is the `migrations` directory. This directory contains a sequential order of files, each prefixed with an automatically generated sequential number with zero padding. These files should transition an empty, just initialized database, into the schema state the project expects.

The final directory contain the `services`. These final set of classes are interacting with (external) services, or provide a logical service component to the rest of the codebase. The first service is handling the LLM requests, called the `GrazieService`. This service has ownership of any interaction with the Grazie LLM Cluster, on which the models are hosted. The inputs from our application are transformed into the appropiate JetBrains cluster request, using the correct authentication without exposing this to the outside. Requests are essentially proxied by the backend, but with an added layer such that we can track the responses back to the requesting user, track ratings, and provide the edits that the user performed after generating a completion.

Another external service is the `Simple Emailing Service (SES)`, which is being used to send out emails to the users who signed up for the user study. This email contains a warm welcome, as well as their API credentials which they will need to use in the JetBrains' IDE Plugin's settings. The emailing service handles authentication to the AWS SES service, and deals with the API requests such that it is easy for other methods in the codebase to send out such emails.[15]

The final service is a utility service that can deal with SSE endpoints. It contains a class for creating SSE packets, serializing them, creating streaming responses and deserializing an SSE response into SSE packets.

### 3.1.4. Development & Deployment Environment and CI/CD

In this section the development environment is explained, and how to boot the development server. This section will also explain the combination of how the React frontend is bundled with the Django framework, into a single Docker image.[16] This docker image can then be used to deploy the full backend. The IALLMs application is deployed using Skaffold, a CLI component that can easily deploy applications onto a Kubernetes cluster.[17] The following sections will explore these components into more detail.

#### Development Environment

The first step to get the application running locally is to setup a database. Since one might work with a lot of databases at the same time, it is better to create a sort of virtual environment for each database instance, similarly to how Python projects are recommended to be setup. This can be achieved using Docker: for each application that requires a particular development environment, a docker compose file could indicate each dependency and allow for an easy to start with development environment.[18] In this case, the docker compose file contains the Postgres database dependency, which when started in the terminal, will contain an empty Postgres database mapped to a local folder on the host system (in the IALLMs backend root project), for persistence across restarts.

Developing with the application can then be done in realtime, as Django features it's own development server which supports live reloading. After a change has been made to the codebase, Django detects this change (through a simple directory watch), and will attempt to restart the server.

There are two techniques to have the signup site appear in the Django application. The first one is less interesting, but can be used to test the final docker image. It requires the build command to be ran in the `frontend` directory, which will then be served by the same application as the iallms app. The other approach is to use the React live-reload development server in addition to the Django development server, where both are ran simultaneously. This has the benefit of working with the frontend in realtime as well, such that stylings, new components are immediately visible, which can greatly speed up development without the need for constant restarts to check out how it visually looks (a practise that is quite common when dealing with visual applications).

The command to start the docker compose services (executed in the root), as well as the command to

[15] `https://aws.amazon.com/ses/`
[16] `https://www.docker.com`
[17] `https://skaffold.dev`
[18] `https://docs.docker.com/compose/`

start the Django development server (executed in the `src` directory), and the React development server (executed in the `frontend` directory), are displayed in Listing 3.5. After running these commands, each development server will print their localhost url which can be visited to inspect the current version of the application.

**Listing 3.5:** Commands to start the development services

```
./$ docker compose up
./src$ python manage.py runserver
./frontend$ npm run start
```

### Building the Docker Image

The application uses Docker to build a stable deployment environment. By using Docker, issues such as “works on my machine”, and other nasty environment issues can be mitigated. Docker is similar to running software on a virtual machine, but Docker instead runs directly on the host OS by translating CPU instructions, and having a small mini-OS on the host’s machine. Such an OS can be Ubuntu, Debian, or based on an Alpine distro. This is essentially the first layer in the Docker Image: its base. Any additional layers, are steps to convert this base layer into the desired deployment environment. This can be done through bash commands, installing additional packaged into this OS (it behaves almost exactly like a similar OS installed on the host machine), or executing docker specific commands to move files during the image building step from the host onto the image. After the image is built, this image can be published to a specific kind of blob storage, called a docker registry. This registry takes in built docker images, and allows other users to download these pre-built applications. They usually have some form of authentication such that it is limited who has access to these files.

For the IALLMs application, a multi-stage dockerfile is used. This means that we start from multiple base images, but later apply a set of merge rules to end up with a single docker image (and inheriting only from one base image). The first base image that’s used is `node:20`. This image comes with the latest LTS node version 20 preinstalled. Similar to how we extend upon a base image, the `node:20` image builts on top of the Debian OS. This node image is used for building the frontend directory, by letting the React and TypeScript compilers compile the TSX into plain HTML, CSS and JS files. The other base image, `python:3.11` contains a fresh install of Python 3.11, and is built on top of the same OS, Debian. In this image, we set the correct linux user permissions and transfer the backend’s source code files onto the image. In addition to the backend’s source code files, the compiled files from the other base image are copied onto this image, such that the backend is able to serve these static files. Finally, the final layer in the Dockerfile tells docker how to start the application when ran. This is done through the `CMD` dockerfile command. The command specified here runs the Python application in the Asynchronous Server Gateway Interface (ASGI) mode. The benefit of using ASGI over the Web Server Gateway Interface (WSGI) is that Python can better handle asynchronous code. Running Python on WSGI means that every web request is run synchronously using web workers. Whenever an IO request is done on the backend, such as a database request because we need to store the users, or a reuqets is made to the Grazie LLM Cluster, the response would be awaited on the same thread executing requests. This is quite wasteful: when Python is waiting, we’d rather spend that CPU time on handling another request. Fortunately ASGI shines on this aspect: all requests are handled asynchronously and multiple requests can be handled at the same time by a single web worker. The ASGI implementation that is used for serving the Django application is Daphne, an ASGI web server made by the Django team themselves.[19]

### Deploying on Kubernetes with Skaffold

The deployment of the application is done using Kubernetes (k8s). The first part in making the app ready for a Kubernetes environment is containerizing the application, which was done using Docker. For deploying the actual application on a Kubernetes cluster, Skaffold is used. This CLI tool applies kubernetes files onto a cluster, and takes away some of the boilerplate when dealing with building, pushing and deploying the docker image onto the cluster. In the Continuous Integration (CD) environment, skaffold takes care of building the docker image and publishing it to an internal docker registry. The is configured in the *skaffold.yaml* file located in the root of the IALLMs backend. This file specifies

[19] `https://github.com/django/daphne`

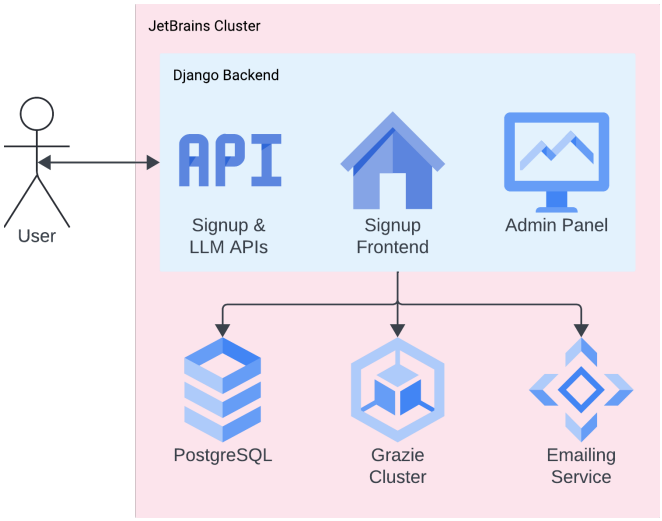



**Figure 3.3:** Overview of the backend's deployment on the JetBrains Kubernetes cluster.

the deployment artifact (the built docker image in the registry), build arguments for building the docker image, and hooks that can be executed on the host machine that's building the docker image before or after the image has been built. This is handy to communicate variables to the next step after building and publishing: running the fresh set of database migrations. For that, a simple bash script has been created, *migration.sh*, which applies the *k8s/migration.yaml* file onto the cluster, and waits for it to complete successfully. In case it errors, the deployment fails, and this is immediately recognized by the CI environment, and thus the pipeline will fail. The developer can then fix the Continuous Deployment (CD) by pushing a fixing commit, such that the CI/CD environment can pick this up and restart with a new deployment. Fortunately, this failing deployment doesn't affect the currently running application, as applications are only rotated when a new deployment is successful.

The *skaffold.yaml* file also references the Kubernetes files for the app image: *k8s/app.yaml*. This file contains the Kubernetes Service, Deployment, and Ingress objects for the application. Environment variables are referenced in the Deployment object from k8s Secret objects, such that they can be used in the application. The Service object exposes the ports from the Deployment, such that they become available for an Ingress. The Ingress object then exposes this port to the public network, using a regular hostname, and can take care of TLS and SSL certificates. The application is currently deployed on the JetBrains Kubernetes cluster, under the hostname `https://iallms.labs.jb.gg`.

An overview of the deployment can be seen in Figure 3.3. The app has been configured with 3 pod replicas, such that requests can be balanced over these three instances. Each pod is assigned a single CPU, with 2 GBs of RAM. Bursts may allow these resource requests to exceed with a limit of 1.5 CPU units, and 3 GBs of RAM.

## 3.2. JetBrains Plugin

Now that the backend is set into place, the actual plugin can communicate with it to retrieve data. The plugin is made in Kotlin, using the latest available JetBrains IDE Plugin SDK. The latest features have been used to create a gray text code completion, as well as a novel way of presenting gray text code completions. Furthermore, the plugin is set up for A/B user testing, by limiting certain features only to a particular group. The plugin implements a number of features: LLM Chat, LLM Code completions, chat templates, a quick extract code into a new chat action. In the following sections, the architecture design of the plugin will be explained, as well as the other implemented features. The code completion feature, as well as the novel gray-text code completion feature, will be explained in more detail in chapter 4.

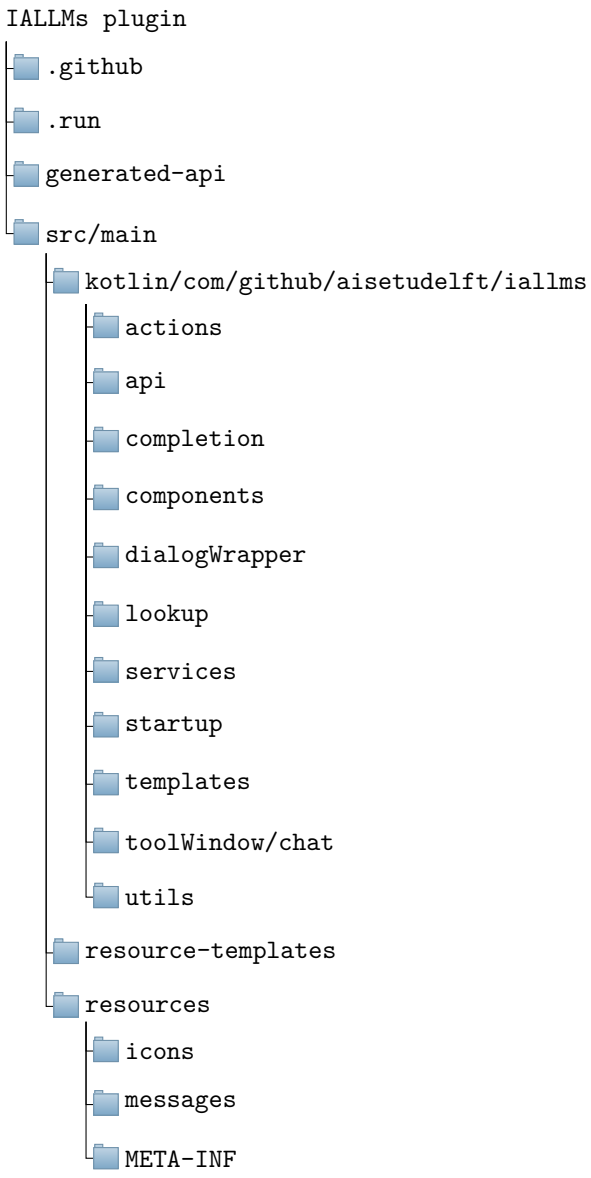



**Figure 3.4:** The file structure of the IALLMs plugin

### 3.2.1. Architecture Design

The main plugin tries to closely follow the standards defined from the starting template[20] JetBrains provides for plugins which target the JetBrains IDEs. The template is written in Kotlin, a JVM language similar to Java and Scala, but with a combined Functional Programming (FP) syntax and Object Oriented Programming (OOP) concepts.[21] Kotlin is a great language to build fast and readable applications in, as compared to Java it has less boilerplate in its syntax. The plugin has also been split into logical components, such that it becomes clear where to locate each specific feature, and to extend the plugin in case other features are needed. This structure has been displayed in Figure 3.4.

The initial starting point of the application is in the *META-INF/plugin.xml* file. This file defines the libraries from the JetBrains Plugin SDK that are required in the plugin, as well as the extension points, event listeners, and actions. This metadata file is used by the JetBrains IDE when loading the plugin to register these actions, event listeners and detect compatibility. The libraries that IALLMs depend on are the common Plugin SDK platform, and the markdown plugin for rendering the LLM chat messages.

[20] https://github.com/JetBrains/intellij-platform-plugin-template
[21] https://kotlinlang.org

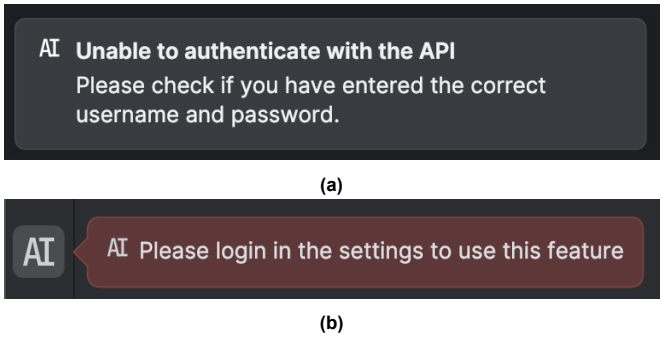



**Figure 3.5:** (a) The balloon error style, displayed on the bottom right of the IDE. (b) The toolwindow style error, displayed as a popover balloon on the IALLMs toolwindow.

Since the backend employs the OpenAPI specification, a generator can be used for generating a full blown Kotlin API client for this particular API. Data classes for each REST resource are generated, as well as methods to make a request to these endpoints using Kotlin methods. Data has the correct Kotlin types, and are automatically serialized to the format the API expects. The generated files are then committed into the same repository, and are located under the `generated-api` folder. The generator is called OpenAPI Generator, and is a CLI utility that will create these Kotlin classes.[22] The command that's used to generate these is displayed in Listing 3.6. This command can be executed from the root project, while the backend development server is running.

**Listing 3.6:** Command to generate the Kotlin API Client from the OpenAPI specification

```
$ openapi-generator generate \
  -i http://localhost:8000/openapi \
  -g kotlin \
  -o generated-api \
  --package-name com.github.aisetudelft.iallms.grazie
```

In addition to the automatically generated API interfaces, a nice and easy abstraction on top of SSE has been built to support custom deserializers per event/data type. Error logging has been added everywhere, this is especially important in visual applications as it might not always be clear to the user what's currently happening. Errors can be communicated through the notification balloons, or through a tool window notification. These types can be seen in Figure 3.5.

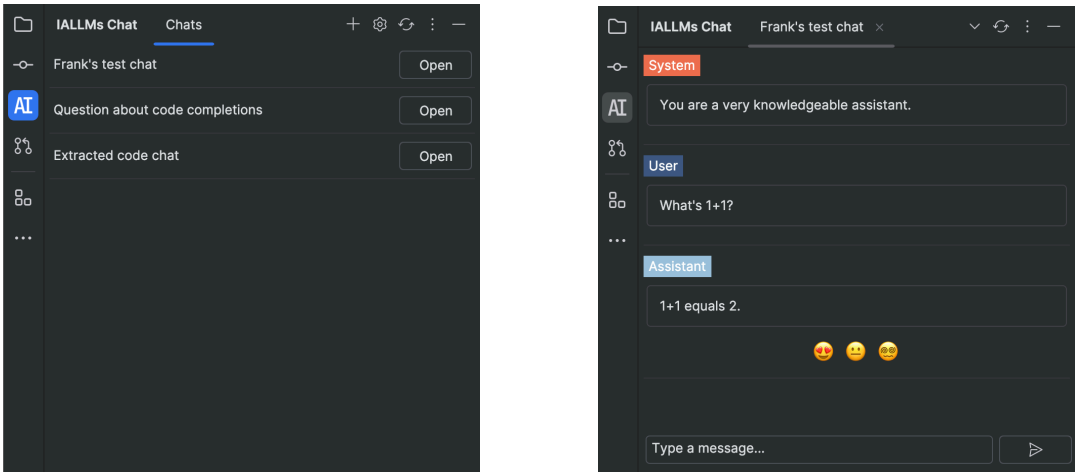
When working with the IDE Plugin SDK (or any software which uses the Swing or JavaFX libraries), it is important to know the difference between the main thread and the asynchronous IO threads. The purpose of the main thread is to be the only thread handling visual changes, such as displaying new windows, elements, tabs, lists, texts, etc. This thread should be kept from performing operations which are IO-bound, such as network requests, or interacting with the filesystem. Performing complex operations on this thread will cause the UI to freeze, and not respond to user input anymore, resulting in a laggy application. Therefore, any complex operation should be performed on an asynchronous thread, which when completed, will schedule the UI-change onto the synchronous UI thread. In the plugin's code, all network requests are done asynchronously, to not block the main thread. JetBrains even has a helpful warning when it detects particular plugins take a long time on the main thread, and will print the stacktrace of the currently being execute code to easily fix this.

### 3.2.2. Plugin Features

The main plugin features are the chat functionality and the code completions. The code completions are presented in the same fashion as in Figure 2.2. For the chat functionality, a custom toolwindow is made to facilitate this. There are two main views in the chat feature: the chats overview page, and the chat detail page. These two views can be seen in Figure 3.6.

There are two ways currently to initiate a chat, either by pressing the (+) button on top of the chats overview tab to create a fresh, blank chat, or by selecting a piece of code in the editor and using the

[22] `https://openapi-generator.tech`




(a) (b)

**Figure 3.6:** (a) The chats overview tool window tab. (b) The chat detail page tab.

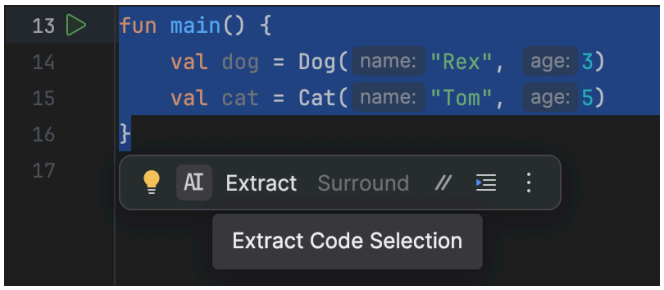



**Figure 3.7:** The floating context bar that appears when selecting code in the IDE.

quick action on the floating bar that appears to extract that code piece into a new chat. The floating context bar that appears is shown in Figure 3.7.

Chats are initiated using templates, configured in the settings of the plugin. This is also the place to add the credentials received by email from the website after signing up. Templates contain variables, which are replaced at the time of the creation of the chat, using the context. For example, the extract code action has access to the `{language}` and `{selectedCode}`, which is not a variable that can be used in the default template. All available variables to a template can easily be retrieved by hovering over the (i) icon in the settings. The settings page is displayed in Figure 3.8, and the popover that displays the list of available variables is displayed in Figure 3.9.

The codebase has been made in a very extendable way for templates, by having a sealed class of the possible templates. Each template extends interfaces that define their context, and whenever a chat is created using a specific context, a new anonymous class is instantiated of the abstract template class. The interfaces are then implemented using this code context. An example implementation of such a template is shown in Listing 3.7, and the definition of the template is shown in Listing 3.8.

**Listing 3.7:** Instantiating an extract code action template.

```
val template = object : ExtractCodeActionTemplate() {
    override fun getProjectName(): String = project.name
    override fun getSelectedCode(): String = selectedCode
    override fun getPsiFileName(): String = psiFile.name
    override fun getPsiFileType(): String = psiFile.fileType.name
}
```

**Listing 3.8:** Instantiating an extract code action template.

```
sealed interface Context
interface DefaultContext : Context {
```

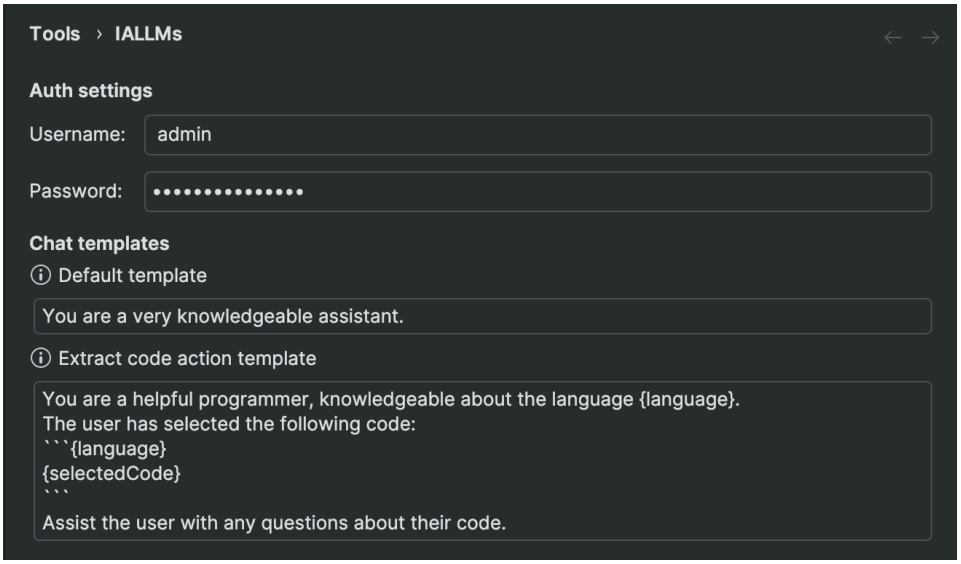



**Figure 3.8:** The settings page of the IALLMs plugin.

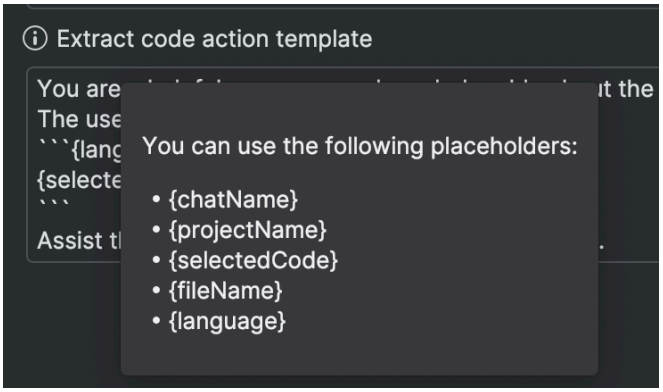



**Figure 3.9:** The available context variables for the extract code action.

```
    fun getProjectName(): String
}
interface PsiFileContext : Context {
    fun getPsiFileName(): String
    fun getPsiFileType(): String
}
interface SelectedCode : Context {
    fun getSelectedCode(): String
}

sealed class Template : DefaultContext {
    fun getRawTemplateText(): String = ...
    fun getPlaceholders(): List<Pair<PlaceholderKey, String>> = ...
}
...
abstract class ExtractCodeActionTemplate : Template(), SelectedCode, PsiFileContext
```

Many reusable components have also been created for modifying the chat panel, as well as the settings page. An abstraction has been made over the settings page, such that state management should be a breeze. JetBrains settings have a “reset” button, to revert to the previously configured state, and a “confirm” button to save the current settings in the view to a persistent storage. Both of these features have been implemented by wrapping fields into a `FieldState` object. Since all the settings can be configured with two types of components, implementations have been made for `PasswordFieldState` and `TextComponentState`. Adding in any additional fields should be very similar to these two field states.

Each chat message that’s generated from the LLM API is presented in realtime. Each time a new SSE chunk is received, the chat message is rerendered using a markdown parser. This is very user-friendly, as the user immediately receives an acknowledgement their action has done something, and they’re not waiting for the backend to do all of the work, and return the response at once. After a chat message is generated from the LLM API, the user can be asked to rate the chat message. This is done by showing three emojis in the chat window, immediately under the assistant’s response. This data collection metric should be very minimally invasive, since it can be completed by simply clicking on an emoji without any further interaction.

Code completions are displayed using the novel gray-text code completion API introduced in the 2024 update of the JetBrains IDEs. The gray-text elements are neatly integrated with the IDE, and multiple providers may suggest completions at the same time. When a provider chooses to provide a code completion for a certain code point, the gray text element is rendered into view. Since these requests for a code completion is triggered quite often by the IDE (almost on every keystroke), developers should filter out requests that are deemed unlikely to be accepted by the developer. This can be done by using a local model that can be inferenced using contextual data whether or not a code completion would be useful at this point, or by using simple heuristics. In the current state of the plugin, a simple heuristic was chosen. If a subsequent code completion request is triggered in a time window of 250 milliseconds, the previous request would be skipped.

Code completions allow for the collection of data about each completion, in order to calculate metrics after the fact. When a completion is generated, the left and right context are both sent to the backend. The left context is essentially everything before the to be infilled code completion, and the right context is everything that comes after. Both of these contexts are stored, as well as the generated completion. Furthermore, when a completion was accepted by the developer, the line where the completion was inserted is tracked for three intervals. After 10 seconds. 30 seconds, and 1 minute, the same line offsetted from where the completion was requested from is sent to the backend. This allows for calculating metrics that show the change of the completion over time, and whether the accepted completion actually ended up in the codebase, and wasn’t immediately deleted after a wrongful insertion.

### 3.2.3. Plugin build tools and CI/CD

For local development, an IDE run configuration is present in the codebase, `.run/Run Plugin.run.xml`. This spawns a local instance of a JetBrains IDE which is preinstalled with the IALLMs Plugin. In the XML configuration, the backend can be configured using env variables. By default, the local instance is set to the default port of a locally ran IALLMs backend, on port 8000. This can be changed to for

example the production instance, for debugging against the production backend.

The project is built using the Gradle Build Tool.[23] A possible alternative build tool is Maven, but Gradle is beneficial over Maven since it can use Kotlin as the Data Syntax Language (DSL) to define the build steps in. The gradle build file (*gradle.build.kts*) defines all the build tasks, the JVM version to compile to, and information for the Gradle IntelliJ Plugin to build, package and sign the codebase for publishing on the JetBrains marketplace. The plugin is currently shown as unlisted on the JetBrains marketplace, as we really want to let users signup through our website, before installing the plugin and being confused what to do. The plugin can be found at `https://plugins.jetbrains.com/plugin/24439-iallms`.

For each commit, a github actions task is summoned to check whether the current plugin can be published or not. It checks against all JetBrains targets the plugin is configured to be compatible with, and checks whether all the APIs exists it uses in the codebase. If all checks succeed, a draft release is created on GitHub that shows the changes compared to the latest released version. When a new version should be published, this draft release can simply be promoted to a published status, and a new github actions job is spawned. This task then builds, signs, and publishes the plugin using Gradle onto the JetBrains marketplace. For this, environment variables are replaced in the source code with production values configurd in GitHub Actions variables, such as the production IALLMs backend url.[24] This ensures no variables are hardcoded in the codebase, and can easily be swapped in the repositories settings if it needs to change.

[23] `https://gradle.org`
[24] `https://docs.github.com/en/actions/learn-github-actions/variables`

# 4

# A User Study on a Novel Gray-Text Code Completion Style

Two types of gray-text code completion have been implemented. One is the “regular” code completion style as displayed in Figure 2.2. The other is a novel gray-text approach, where the suggestions by the statical analysis from the IDE are used as prefixes to the LLM. For example, if the line contained $val\ cat\ =\ C$, the IDE would suggest some possible objects / classes to use starting with the letter $C$. The original gray text completion would not take into account these options selected by the Developer, and thus the LLM might be missing out on better completions.

The novel gray-text code completion will detect which item the user is currently selecting (the developer can navigate between the options presented by the IDE using their arrow keys). This currently selected piece is then suffixes to the left context of the code completion context, which the model can then use as additional context to provide a better completion. This feature is shown in Figure 4.1. On the top of the dropdown, a row with “AI Original completion” can be seen. This row is initially selected from the dropdown, when it appears, and contains the original suggested gray text completion from the LLM API. The developer is then always able to navigate back to the original completion given by the model, whenever they want to insert this completion instead of the novel gray text completion.

The following sections will explain the chosen research questions, and the setup of the user study at the JetBrains Open Source Software (OSS) department. The models used, experimental setup and survey questions will be discussed as well as the results of this small-scale user study.


**Figure 4.1:** The novel gray text completion, using the dropdown completions from the IDE.

## 4.1. Research Questions

The research questions are focused on the interactivity and alignability of the code completion system with the IDE. Having an interactive and aligned system means its well usable, and having correct and better completions means the system is useful as well. Therefore, the following research questions were chosen: *RQ1: How useful is the code completion function?*, and *RQ2: How usable is the code completion function?*. These questions will be answered through a combination of automated metrics as well as a survey.

### 4.1.1. RQ1: How useful is the novel code completion function?

This research question focuses on the data that's automatically collected by the plugin. The data that is collected can be used to collect metrics how useful the code completions actually are. The metrics that will be used are Exact Match (EM) and the Levenshtein Edit Similarity (ES). There are three timestamps where metrics can be calculated. These are the datapoints of the tracked line after 10 seconds, 30 seconds, and 1 minute.

In addition to the automated metrics, seven questions are present in the questionnaire to also gauge the usefulness of the completions, as perceived by the developers themselves. There are seven questions:

1. I am satisfied with the speed of code completions provided by the system.
2. The system's overall performance met my expectations.
3. The code completion suggestions helped me write code more quickly than I would have otherwise.
4. The suggestions provided by the code completion system were relevant to my coding tasks.
5. The code suggestions were accurate enough that I could use them without significant modifications.
6. I would recommend this code completion system to other developers.
7. I would prefer to continue using this system for future coding projects.

These questions are based on the specific code completion plugin that is created, and ask about the performance, accuracy, and whether this system is useful for future projects.

### 4.1.2. RQ2: How usable is the novel code completion function?

This question aims to answer the nuanced aspects of the plugin which might be hard to answer through automated metrics. For example, how easy was the system to use? A system could be hard to use but still give good metric results, therefore it is of essence to gauge these observations through a survey. To answer this research question, the System Usability Scale was chosen as the choice of questions for the survey [4].

The questions are as follows:

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

These questions aim to answer the effectiveness of the solution, the efficiency of the user when using the plugin, and the satisfactory level of the user when using it. They are answered based on a Likert scale, to be rated from "Strongly Disagree" (1 points) to "Strongly Agree" (5 points). In the end, the

$$SUS = 2.5 \times \left( 20 + \sum_{i=1,3,5,7,9} X_i - \sum_{j=2,4,6,8,10} X_j \right)$$

**Figure 4.2:** The formula for calculating the System Usability Scale Score.

$$X = \frac{4}{5} \times S_i + 1$$

**Figure 4.3:** A linear equation to map scores 0-5 evenly onto scores of 1-5.

SUS-score can be calculated for a given set of answers to all these questions. This score is in the range of 0-100, and can be calculated using formula shown in Figure 4.2.

The variables $X_i$ and $X_j$ are the scores for the odd and even question numbers respectively. In our case, we formulated the questions on a scale of 0-5, resulting in a 6-point scaling system. However, the SUS score requires values to be in a range from 1-5, so to transform 0-5 into 1-5 the linear equation in Figure 4.3 is used to map the values.

In addition to the SUS score, three open ended questions have been included in the questionnaire that ask the developer about even more nuanced details of their experience. They are able to share their experiences, and suggest some feedback if they have any. The following three questions are present:

1. What did you like about the system, what potential value does it bring for you as a user?
2. What challenges, if any, did you encounter while using the code completion system? How did you address these challenges?
3. What improvements or additional features would you suggest for this code completion system?

Together with the SUS questions, this should provide an accurate insight into whether the code completion system was experienced usable or not.

## 4.2. User Study at JetBrains OSS

The research questions are tested in an A/B user study, where one group (A) will get the “normal” gray-text code completion without alignment with the statical analysis of the IDE (as control), and the other group (B) will have the novel code completion style. The user study was internally performed at JetBrains, via a select amount of users working on opensource projects within the company to not expose and collect company code under a Non-Disclosure Agreement (NDA).

The code completion model used in the backend is a model trained by JetBrains, called `code-one-line-complete:jet-all-default`. This is a code completion model capable of using both the left and right context of the user’s opened document. Furthermore, it is capable of generating code completions for a wide variety of languages, such as Python, Kotlin, Java, and JavaScript. For the chat feature, OpenAI’s GPT-4 was used.

The users were recruited through Slack, the company’s communication tool for coworkers. They were asked to go to the IALLMs website, signup, and install the plugin through the unlisted JetBrains marketplace link. After installing, the credentials should be entered into the plugin’s settings, and we instructed the employees to code “as usual”, while exploring the plugin’s code completions. They could also make use of the chat functionality, and see if it helps them with programming as well. The main focus was on the code completion feature: in total there were 20 users who signed up for the study, but only 10 actually installed the plugin and were participating in the user study.

Assigning a user to a group was done using a simple modulo operation on the automatically generated autoincrement id from the database. Of the 10 users that were left over, 3 were in the control (A) group, and 7 were in the feature group (B). Both groups participated in a similar amount of completions in the end. After filtering, the groups had 5 647 and 6 203 triggers, respectfully.

Filtering on the completions was done since it was noted that completions take longer than usual to

**Table 4.1:** Results over time for the EM and ES metrics, after 10 seconds (10S), 30 seconds (30S), and 1 minute (1M).

| User Group | Metric | 10S | 30S | 1M |
|---|---|---|---|---|
| A (control) | EM | 41.67 | 39.76 | 32.10 |
| | ES | 78.24 | 75.32 | 70.77 |
| B | EM | **72.24** | **63.19** | **58.03** |
| | ES | **90.23** | **86.59** | **84.38** |

**Table 4.2:** Results of all the completions for each group.

| User Group | Accepted | Total | Ratio |
|---|---|---|---|
| A (control) | 86 | 5 647 | 1.52% |
| B | 614 | 6 203 | 9.90% |

appear on the client. Completions are triggered too often, and often result in the completion being skipped before it was even presented to the user. Therefore, a filtering technique based on code completion sessions was implemented. A single code completion session should either result in a completion being shown, but implicitly rejected by continueing typing, or accepted by tab-completing it. The following definition defines code completion sessions for the set $C$ of all code completions, for each consecutive pair $c_x$ and $c_y$:

1. $c_x$ and $c_y$ are no further apart than $t$ ms, or
2. $c_x$ is within the accepted timestamp of a previous completion ($c_1 ... c_{x-1}$)

The value chosen for $t$ is 2 000ms, as from empirical testing, it took around 2 seconds before a completion actually appeared in the IDE when a code completion was triggered. It is important to note that this technique does not filter away accepted code completions, and only filters away completions which were implicitly rejected too fast before it could be shown. This reduced the total number of code completions of both groups combined from 35 140 code completions to 11 850 code completion sessions.

### 4.2.1. RQ1 Results

The results to the first research question are displayed in Table 4.1. In table Table 4.2 the results for both groups' total amount of accepted completions can be seen. In Figure A.3 and Figure A.4, graphs for the SAU questions for both user groups combined can be seen.

The data displays a strong increase in accuracy as well as edit similarity over time for the novel code completions. This can further be explained by the fact that part of the context is better: by using the statical analysis of the IDE, the model can better predict what the user actually wants. Since the user already showed intent of using a particular IDE dropdown completion, they are more likely to accept a code completion which uses this extra context.

The results of the total number of accepted completions for user group A are quite low. This could be explained by the fact that completions take quite a while to appear, and the user might not like that when they're in the acceleration phase of programming, contrary to group B who are more in the exploration phase, as they are exploring the options presented by the IDE at the same time. The overhead of having a completion take a little bit longer might then be accepted. This can also be seen by the results of the first and third SAU question, there are two peaks, one user group found it too slow and slowed them down, and the other found it okay, and wrote code quicker for them. The interesting result here is that the users in group B actually found the code completions too slow, while the users in group A were quite happy with the code completion speed. This is contrary to what is expected, but is consistent for group B. They actually found that it slowed them down, while it sped things up for group A. Both groups did however think the results were accurate, when accepted.

**Table 4.3:** Results per SUS question for each group.

| Question | User Group | Score |
|---|---|---|
| SUS1 | A | 4.5 |
| | B | 2.25 |
| SUS2 | A | 0 |
| | B | 1 |
| SUS3 | A | 5 |
| | B | 2 |
| SUS4 | A | 0 |
| | B | 0 |
| SUS5 | A | 3.5 |
| | B | 2 |
| SUS6 | A | 1 |
| | B | 1.5 |
| SUS7 | A | 5 |
| | B | 4.25 |
| SUS8 | A | 0 |
| | B | 1.75 |
| SUS9 | A | 3.5 |
| | B | 3 |
| SUS10 | A | 0 |
| | B | 0.25 |

**Table 4.4:** The SUS Scores for each user group

| User Group | Score |
|---|---|
| A | 93.5 |
| B | 70.5 |

### 4.2.2. RQ2 Results

The results for the SUS scores per user group are displayed in Table 4.3. It can be seen that the positive questions (higher score is better), which are represented by odd numbers, are consistently higher in user group A. Similarly, for the negative questions (lower score is better), represented by even numbers, are slightly lower in user group A. In Figure A.1 and Figure A.2, graphs for the SUS questions for both user groups combined can be seen. In the end, of the 10 users that ended up using the plugin, 6 participated in the survey.

The results for the SUS-score, calculated using the formula in Figure 4.2, in combination with the linear transformation in Figure 4.3, are displayed in Table 4.4. This can be explained by the fact that this system is quite new, and the usability of this system needs to become more battle tested.

In the open questions it was noted that the completions did take a while, longer than expected, to appear. But, especially for group B, the accuracy was noted. Furthermore, a common requested feature is the deduplication of parenthesis and closing brackets when an inline completion is accepted.

# 5

# Challenges

There were quite some challenges when making the plugin and backend. Making a reusable plugin for testing interactivity and alignability was a good project on itself, but in the thesis we also wanted to perform an experiment with the plugin. This took a bit longer to figure out, since we needed a good small scale user study where we could use the plugin in the deployment. Fortunately, in the final 2.5 months of the thesis, S. Titov came with the excellent idea to perform an experiment on a novel way of generating code completions.

Another challenge was moving from the staging Grazie LLM cluster to the production version. The documentation unfortunately isn't particularly present for this internal service, which caused an issue as a staging model wasn't available on the production cluster. The staging cluster was using an older version of the code completion inference API as well, so that had to be switched to the novel "Task API", when moving to the production cluster. Luckily, the JetBrains team was very helpful in providing alternative models, and we even ended up using a model which was trained from scratch by the JetBrains team. This also provided a new benefit: whereas the model we intended to use, CodeLLama-7B, only supported the left context, the JetBrains model `code-one-line-complete:jet-all-default` supports both the left and right context. This results in a better accuracy, as the model has more quality surrounding context.

In the early phases of the plugin, figuring out which code completion API to use was quite cumbersome. Documentation was lacking a bit on this side as well, and since this API was very experimental in previous IDE versions (all 2023 versions), we had to limit compatibility to the recent JetBrains IDE to use the latest available version of the gray-text code completion. The earlier versions of the API should be supported by later versions of the IDEs, but when testing the plugin manually a few issues were present that led us to bumping the minimum supported version to the latest IDE release.

For the chat feature, it was quite a challenge to get the live-rendering of the assistant's response working. The markdown parsers convert markdown into HTML, and displaying this HTML in the IDE had some issues with setting the proper height for the container that contained the HTML element. This was eventually fixed by creating a workaround by patching the markdown parser. Another JetBrains coworker had the exact same issue, and we gave them the same workaround we used, which helped them as well.

# 6

# Conclusion

In conclusion, the task of creating a plugin and backend has been successful.

The backend has been written using a logical structure, using common, battle-tested languages and frameworks. Python was used since its the most popular programming language, and it can easily be used with LLM providers such as HuggingFace. The signup page was made using React, TypeScript and Mantine, to create a maintainable and extendable signup website. The database models are not fixed in Postgres, migrations can easily be made and executed in the codebase. Server Sent Events has been implemented for the Grazie LLM Service, as well as an easy-to-use emailing service using Amazon SES. The application has also been deployed using a qualitative enterprise deployment using Kubernetes. CI/CD has been setup to automatically roll out new deployments with Skaffold, while handling Django database migrations and containerizing the application properly using Docker.

The plugin is written with a similar logical structure, while integrating neatly with the backend. OpenAPI generators have been used for automatically generating a Kotlin API client, which simplified using the backend API. The plugin also features two major features: a chat toolwindow as well as code completions. Two versions of code completion are present, the “regular” gray-text code completions, and a novel approach. Templates have been implemented for the chats as well, and can be easily extended upon. Chat messages can be rated, and this data is collected in the backend. Code completions are tracked after 10s, 30s, and 1 minute, which allows for a nice way of tracking the ground truth of code completions over time. The plugin uses Gradle as build tool, which builds, verifies, signs, and publishes the plugin to the JetBrains marketplace. GitHub Actions was used as CI/CD for creating automatic deployments.

In the user-study, interesting results were presented. Although the accuracy and edit similarity was higher from the calculated metrics for group B, and thus had a higher usefulness score than the gray-text code completions, the users found the system less usable according to the SUS score. This could be explained by the fact that the system is novel, and/or might need some getting used to.

# 7

# Future Work

The main contribution of this thesis are the reusable plugin, the backend, and the work done on a novel gray-text code completion. Future work could reuse the plugin for other user studies that might involve Chat LLMs and Code completions. The work has been made in an extendable and maintainable style, so adding in features and modifying the codebase should be attractive for future studies.

Furthermore, future work could work on making adapters into HuggingFace directly, which can be done because the backend's code is made in Python. HuggingFace could allow for thousands of other models to be tested in the plugin, and could allow for a medium to test models trained by researchers out in the wild using the plugin. Most LLMs, when trained, aren't directly tested in the real world, and would only be run against benchmarks and the test dataset. Including a small real-world experiment as part of a new model's release could be interesting to include.

Another feature that future work could focus on is the implementation of a local filtering model on the plugin's side. In the current iteration, we employed a simple heuristic to ignore keystrokes happening within 250ms, but this still posed some challenges when we got the actual metrics. Hence, by implementing a local filtering model, the session filtering would need to filter less obsolete completion requests. Furthermore, both approaches could be tested against each other, to see the different sets of filtered out completions by the session filtering versus a local filtering model.

Future work could also improve upon the distribution of the A/B-user study user distribution, as it was noted during the study a lot of people sign up, but never actually install the plugin. This could mess with the sizes of the groups, they should preferably roughly equal. Although this might be resolved already by just having more users join the user study, it could also be prevented by only assigning users to a group when they actually install the plugin. Logic could be ran in the backend on the first API request made by a user, and then assign them to a user group immediately.

Finally, future work could investigate the usage of this novel gray text completion even more by including a larger population of test users in the wild, from differing companies and backgrounds. The next iteration of the plugin should likely also investigate into using a faster code completion model, as it was noted during this user study this was a let down of the plugin.

# 8

# Related Works

The integration of Large Language Models (LLMs) into software development workflows, particularly in the context of code generation and assistance, has seen a significant evolution over recent years. This evolution is marked by the development of tools that not only generate code but also interact with developers in an increasingly sophisticated manner. This work aims to further this evolution by exploring the usability and effectiveness of code completion and chat-based LLMs directly within Integrated Development Environments (IDEs). To contextualize this study, a review was done on related works that have laid the groundwork in understanding the interaction between developers and AI programming assistants, the challenges faced, and the opportunities for improvement.

Liang et al. [7] conducted a large-scale survey on the usability of AI programming assistants, uncovering both successes and challenges. They identified issues such as the generated code not meeting developers' intentions (categorized with the code S22), developers giving up on incorporating code (S2), and the cognitive load imposed by balancing interaction modes with user tasks. Their findings suggest the need for improved developer-tool alignment and the incorporation of non-functional requirements like readability and performance into code generation.

Sarsa et al. [13] highlighted the potential of LLMs in generating tailored programming exercises and explanations, suggesting further investigation into their capability to handle more complex and advanced computing concepts. This aligns with our focus on enhancing the usability of LLMs through features like customizable prompt templates and automatic code insertion.

The study by Rodriguez-Cardenas et al. [11] emphasizes the importance of benchmarking LLMs for source code interpretation, suggesting future research to identify unmeasured confounders that could affect LLM predictions.

Yetiştiren et al. [15] evaluated the code quality generated by various AI-assisted tools, identifying common errors such as the use of functions from unimported libraries and syntax errors. The former this project aims to investigate as well by utilizing a novel OpenAI API.

Vaithilingam et al. [14] and MacNeil et al. [8] both stress the need for better ways to understand, edit, and validate generated code, with MacNeil et al. particularly noting the significance of prompt engineering and the benefits of line-by-line explanations. They also found that few-shot learning might not be particularly helpful for generating helpful explanations and that the model tends to overfit on the structure of the response.

Studies also explored a dual role of GitHub Copilot as both an asset and a potential liability, depending on the user's expertise level, suggesting that expert developers tend to get more use out of LLM developer tools, while novice developers tend to get confused [6, 12]. Barke et al. [3] recommended giving developers more control over input to code-generating models, which could reduce confusion for novice developers.

Another study confirmed this finding by evaluating students using Copilot [10], and observed two ad-

ditional phases in addition to the acceleration and exploration phase: shepherding and drifting. The former explains students mindlessly trying to get a completion from copilot, and the latter students hesitantly accepting the output but later on ended up deleting that part of the code. Aligning the LLM with a chat functionality in addition to regularly styled code completions can mitigate both of these issues.

Finally, Chaves and Gerosa [5] investigated the social characteristics in human-chatbot interaction, identifying the management of user expectations as a critical factor. In this work, we aim to address this issue by having clearly defined features that give the developer control over the input of the prompt.

These studies collectively underscore the complex interplay between developers and AI-assisted tools, highlighting the need for continued exploration into how these tools can be designed to better meet developers' needs, reduce cognitive load, and improve code quality. This study builds on this foundation, aiming to address some of the identified gaps through a user evaluation of customized, interactive LLMs within easily accessible IDE settings.

# References


[1] 2023. URL: https://survey.stackoverflow.co/2023/.

[2] 2024. URL: https://www.tiobe.com/tiobe-index/.

[3] Shraddha Barke, Michael B. James, and Nadia Polikarpova. *Grounded Copilot: How Programmers Interact with Code-Generating Models*. 2022. arXiv: 2206.15000 [cs.HC].

[4] John Brooke. "SUS: A quick and dirty usability scale". In: *Usability Eval. Ind.* 189 (Nov. 1995).

[5] Ana Paula Chaves and Marco Aurelio Gerosa. "How Should My Chatbot Interact? A Survey on Social Characteristics in Human–Chatbot Interaction Design". In: *International Journal of Human–Computer Interaction* 37.8 (2021), pp. 729–758. DOI: 10.1080/10447318.2020.1841438. eprint: https://doi.org/10.1080/10447318.2020.1841438. URL: https://doi.org/10.1080/10447318.2020.1841438.

[6] Arghavan Moradi Dakhel et al. *GitHub Copilot AI pair programmer: Asset or Liability?* 2023. arXiv: 2206.15331 [cs.SE].

[7] Jenny T. Liang, Chenyang Yang, and Brad A. Myers. *A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges*. 2023. arXiv: 2303.17125 [cs.SE].

[8] Stephen MacNeil et al. *Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book*. 2022. arXiv: 2211.02265 [cs.SE].

[9] Mayank. 2023. URL: https://www.htmhell.dev/adventcalendar/2023/13/.

[10] James Prather et al. ""It's Weird That it Knows What I Want": Usability and Interactions with Copilot for Novice Programmers". In: *ACM Transactions on Computer-Human Interaction* 31.1 (Nov. 2023), pp. 1–31. ISSN: 1557-7325. DOI: 10.1145/3617367. URL: http://dx.doi.org/10.1145/3617367.

[11] Daniel Rodriguez-Cardenas et al. *Benchmarking Causal Study to Interpret Large Language Models for Source Code*. 2023. arXiv: 2308.12415 [cs.SE].

[12] Daniel Russo. *Navigating the Complexity of Generative AI Adoption in Software Engineering*. 2024. arXiv: 2307.06081 [cs.SE].

[13] Sami Sarsa et al. "Automatic Generation of Programming Exercises and Code Explanations Using Large Language Models". In: *Proceedings of the 2022 ACM Conference on International Computing Education Research - Volume 1*. ICER 2022. ACM, Aug. 2022. DOI: 10.1145/3501385.3543957. URL: http://dx.doi.org/10.1145/3501385.3543957.

[14] Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. "Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models". In: *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*. CHI EA '22. New Orleans, LA, USA: Association for Computing Machinery, 2022. ISBN: 9781450391566. DOI: 10.1145/3491101.3519665. URL: https://doi.org/10.1145/3491101.3519665.

[15] Burak Yetiştiren et al. *Evaluating the Code Quality of AI-Assisted Code Generation Tools: An Empirical Study on GitHub Copilot, Amazon CodeWhisperer, and ChatGPT*. 2023. arXiv: 2304.10778 [cs.SE].

# A

# Survey Results

In this appendix the raw results from the user study are shown. Figure A.1 displays the questionnaire responses for the ten System Usability Scale questions, and Figure A.3 show the answers for the Specific Aspects of Usefulness survey questions.


**Figure A.1:** The results of the user study on the System Usability Scale (SUS) questions 1-5. Group A is presented on the left, and group B on the right.

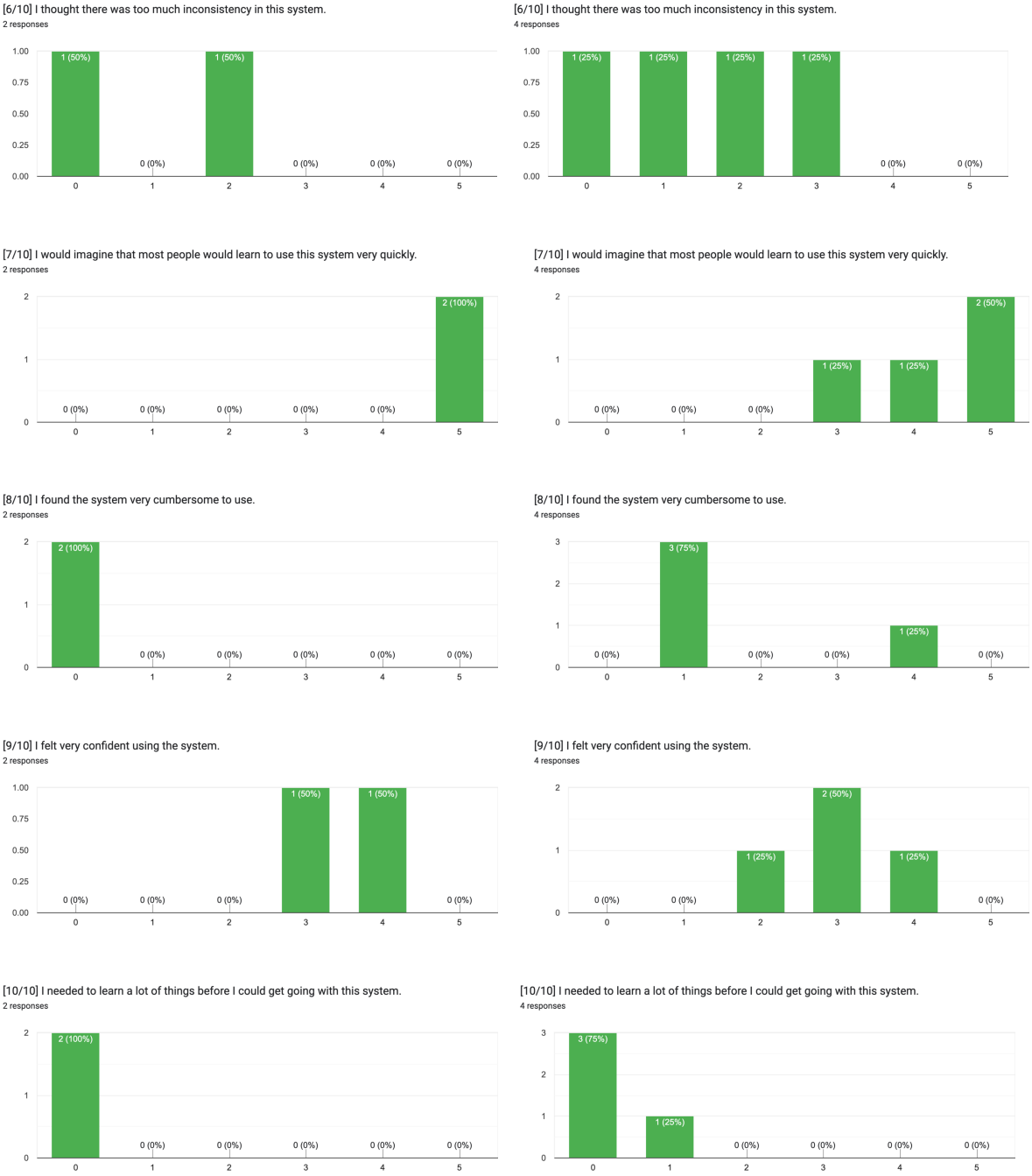



**Figure A.2:** The results of the user study on the System Usability Scale (SUS) questions 6-10. Group A is presented on the left, and group B on the right.

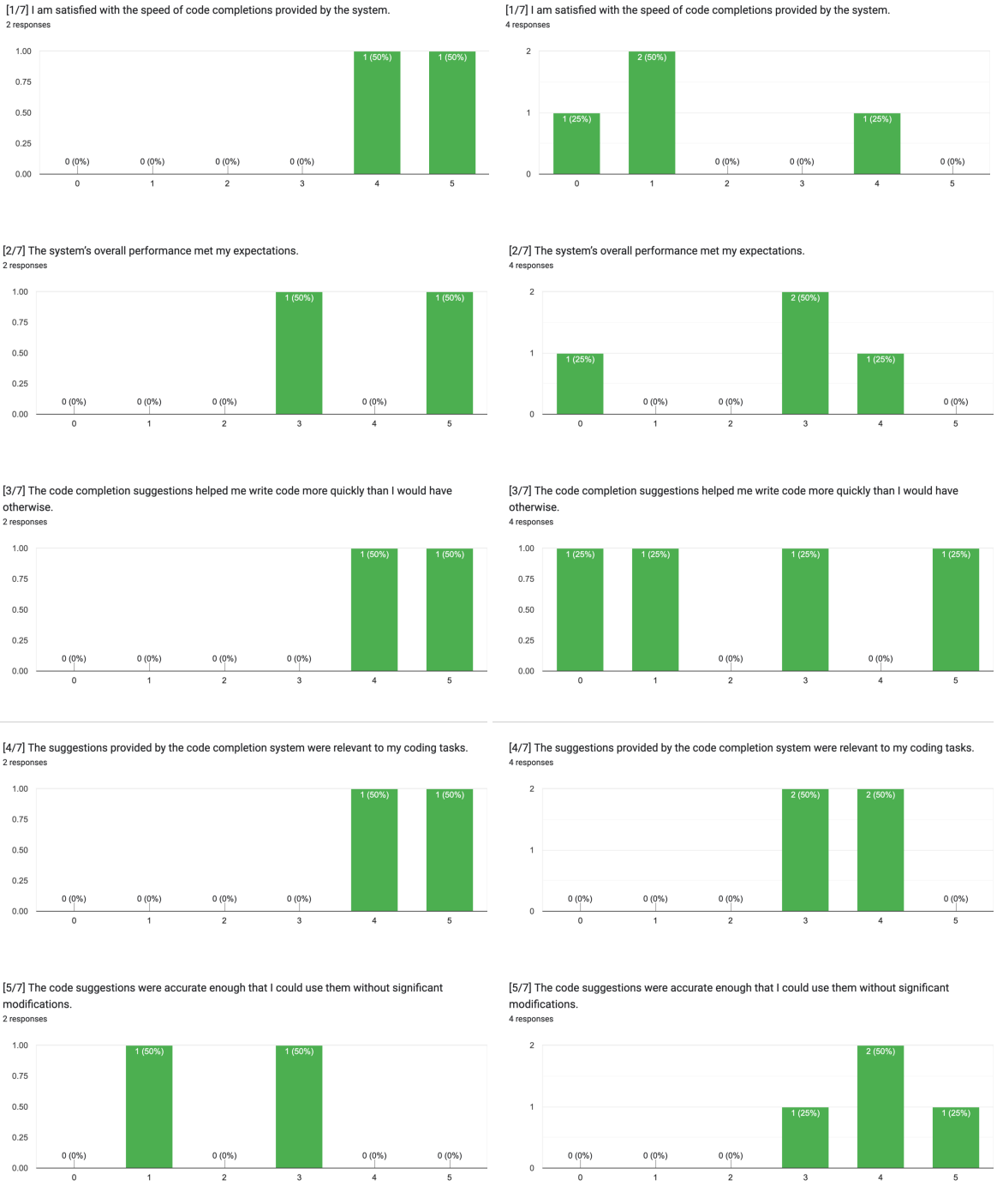



**Figure A.3:** The results of the user study on the Specific Aspects of Usefulness questions 1-5. Group A is presented on the left, and group B on the right.

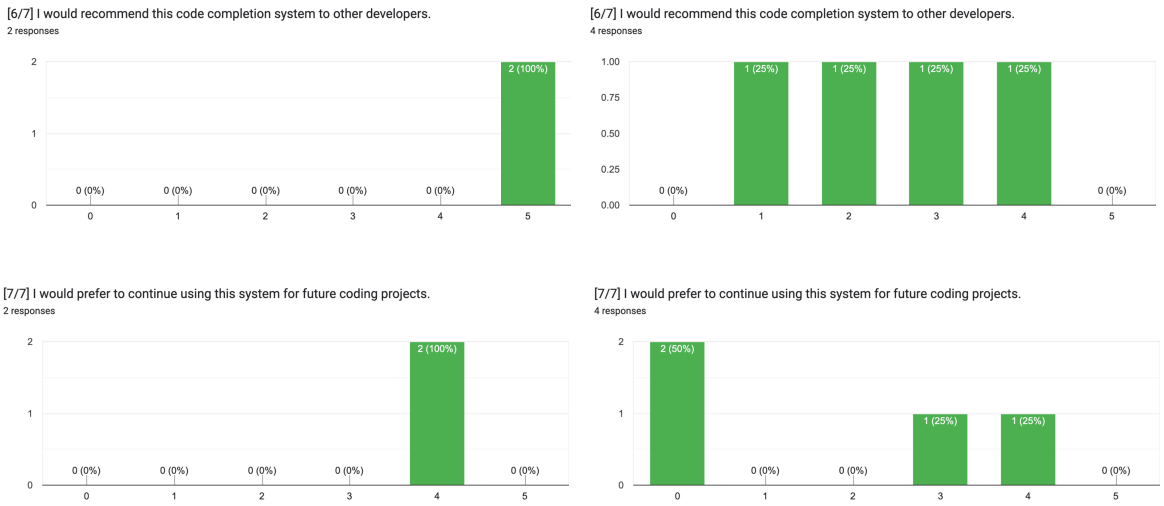



**Figure A.4:** The results of the user study on the Specific Aspects of Usefulness questions 6 and 7. Group A is presented on the left, and group B on the right.