Meet Your Onboarding Buddy

A Smart, Adaptive, and Conversational LLM Assistant to Smooth Your Software Onboarding Journey

Andrei Cristian Ionescu

Delft University of Technology

TU Delft

Meet Your Onboarding Buddy

A Smart, Adaptive, and Conversational LLM Assistant to Smooth Your Software Onboarding Journey

by

Andrei Cristian Ionescu

to obtain the degree of Master of Science

at the Delft University of Technology,

to be defended publicly on 6 November, 2024 at 3:00 PM.

Student number:	5002575
Project Duration:	February 19, 2024 – November 06, 2024
Thesis Committee:	Prof. Dr. A.E. Zaidman, TU Delft, chair
	Assistant Prof. Dr. M. Izadi, TU Delft, supervisor
	Assistant Prof. Dr. R. Hai, TU Delft

Cover:	AI-generated image, modified by Andrei-Cristian Ionescu
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

Summary

Effective onboarding in software engineering is critical yet challenging due to the rapid evolution of technologies, languages, frameworks, and tools. Traditional exploration, documentation and workshop-based onboarding methods tend to be expensive, time-consuming and can get outdated very fast in large, complex projects.

In this thesis, we introduce a novel solution: the Onboarding Buddy system which uses large language models (LLMs) and retrieval augmented generation (RAG), enhanced by an automated approach for chain-of-thought (CoT) that improves onboarding for new and existing developers. It integrates natural language explanations available in the development environment with relevant information, code explanations, and project-specific guidance. The system architecture is agent-centric, including contextualization, onboarding agents, instruction step processors and message enhancement agents that cooperate in delivering comprehensive, customized support with minimal reliance on human mentors.

While effective in supporting the completion of tasks and reducing stress related to onboarding, feedback also revealed some areas for improvement, like better context awareness, explicit instructions, improved technical stability, and UX adjustments. In general, Onboarding Buddy is an excellent promise to smoothen the onboarding process and therefore increase developer productivity and job satisfaction.

The experimental results demonstrated the system's effectiveness: participants spent an average of 175 minutes actively engaged in the IDE, completed tasks with nearly 100% accuracy, and gave high helpfulness ratings (3 out of 4). Task completion times averaged 50 minutes, with simpler tasks taking around 28 minutes and complex ones requiring 67 minutes. User feedback showed high satisfaction (3.35/4 for understanding, 3.15/4 for accuracy) and strong interest in such solutions (7.75/10). Interestingly, more experienced developers spent more time on tasks, suggesting a deeper exploration of the codebase. A strong positive correlation (0.70) between system usage frequency and perceived helpfulness indicated that increased engagement led to better outcomes.

In other words, while there are areas for improvement, such as context awareness and processing complex tasks, this research proves that LLM-based onboarding solutions are feasible and can have significant positive impacts on the software engineering onboarding process, thus laying the foundation for future progress in automated developer support and knowledge sharing for software development.

Contents

List of Figures

Nomenclature

Abbreviations

Abbreviation	Definition
API	Application Programming Interface
AWS	Amazon Web Services
CoT	Chain of Thought
HR	Human Resources
HTTP	Hypertext Transfer Protocol
IaC	Infrastructure as Code
IDE	Integrated Development Environment
JSON	JavaScript Object Notation
JWT	JSON Web Token
LLM	Large Language Model
ML	Machine Learning
NLP	Natural Language Processing
PDF	Portable Document Format
RAG	Retrieval Augmented Generation
SDK	Software Development Kit
SoC	Separation Of Concerns
UI	User Interface

1 Introduction

Software engineering is an intricate and varying field where engineers need to be highly dynamic and ready for frequent learning. The software developer has to understand and operate technologies, languages, frameworks, and tools, with which he most probably collaborates with other developers and stakeholders. This poses a significant challenge for new hires, who have to quickly familiarize themselves with the codebase, the project requirements, the development environment, and the team culture. In addition to this, software engineering is a moving discipline, and new trends, innovations, and best practices change very often. Therefore, a good developer is compelled to keep themselves updated, and their knowledge and skills refreshed [3, 14]. A viable solution for these challenges can be a well-established onboarding system with good training features for both new developers and the enrichment of existing developers' performance.

However, enabling such environments for even large and complex projects, which might include proprietary libraries or tools is costly in terms of time and resources needed to maintain such systems [2]. Furthermore, traditional methods, such as documentation and workshops, may no longer be useful, codebase exposure being a more important factor in understanding a project [9]. For instance, the documentation may be out-of-date, lacking, or hard to follow, videos may be too long, too general, or too specific, or the workshops too few, too inconvenient, or irrelevant. There is hence the need to bring innovation in orienting and training software developers. With recent developments in LLMs, intelligence systems can quickly forward relevant advice to each developer [21].

Natural language generation is an aspect of artificial intelligence that focuses on the production and synthesis of human language. These, when combined, can enable the creation of intelligent and interactive systems for communication with software developers in a natural language, assisting them with relevant and useful information, explanations, feedback, and guidance. In this paper, we present the design, implementation and empirical evaluation through an explorative user study of such a system, which we call Onboarding Buddy.

To guide our research and evaluation of the Onboarding Buddy system, we focus on four key research questions:

RQ1: What patterns of efficiency and task completion emerge when developers use Onboarding Buddy for software onboarding tasks, as measured by time spent in the IDE, number of file operations, and task completion rates?

RQ2: What relationships exist between developers' perceived usefulness of Onboarding Buddy and their task completion patterns?

RQ3: What is the relationship between user satisfaction with Onboarding Buddy and their engagement levels in effectively completing onboarding tasks?

RQ4: How feasible is the implementation of Onboarding Buddy as an LLM-based assistant for onboarding tasks?

Onboarding Buddy is a language description system that can provide natural language explanations for software projects integrated with a development environment, considering the onboarding of new employees and the creation of training materials through a chat-like interface. Buddy uses state-of-the-art language models like GPT-4 [23] or Gemini [1] to enable the provision of short but extensive statements to developers about what the code intends to do. Onboarding Buddy also uses Retrieval Augmented Generation (RAG), which combines the benefits of retrieval-based models and generative models. RAG can retrieve relevant documentation or code snippets when a new team member asks a question, and provide them as context to the language model, which generates a response. This can be particularly useful for answering project-specific questions that a generic language model might struggle with. On top of that, Buddy uses an automated chain-of-thought (CoT) paradigm to guide the user through the learning process. The onboarding buddy can adapt to various domains, across multiple languages and programming styles. It can also show appropriate examples with references, wherever required. The Onboarding Buddy solution is developed to bring together the language model, RAG, and MLOps components as a cohesive solution. New team members will be able to interact with Onboarding Buddy via a chat interface to ask questions on a software project and its documentation. In response, the Onboarding Buddy would carry out retrievals of relevant documents using advanced RAG and a CoT answering and prompting technique to respond to those questions using a language model.

The thesis is organized as follows: Section 2 introduces the topic of software engineering onboarding challenges and presents Onboarding Buddy as a potential solution. Section 3 covers background material including LLMs, current onboarding practices with case studies, and virtual assistants with a focus on RAG technology. Section 4 presents the experimental ”plug and play” platform architecture of Onboarding Buddy, detailing its backend, chat UI, and plugin components. Section 5 explains the agent-centric approach behind Onboarding Buddy’s functionality, covering its contextualization agent, onboarding agent, step processor, and message enhancer components. Section 6 details the study setup, describing the chosen software project and the three onboarding tasks that participants completed. Section 7 presents comprehensive results on user engagement, task performance, feedback, and research questions related to efficiency, usefulness, satisfaction and overall feasibility. Section 8 provides conclusions, and Section 9 outlines potential future work. Supporting appendices provide the experiment manual, a study of embedding performance in RAG, and additional task details.

2
Background

2.1. Introduction to Large Language Models (LLMs)

The fields of Natural Language Processing (NLP), Machine Learning (ML), and Artificial Intelligence (AI) have witnessed significant advancements over the decades. From the early conceptualization of languages as systems, we have now entered an era where Large Language Models (LLMs) are a reality, transforming how we interact with technology. This journey has been marked by milestones, such as the emergence of NLP and ML in the 1950s, the AI winter in the 1970s [26], a resurgence of ML research in the 1980s, and the shift towards data-centric approaches in the 1990s [13]. The 2000s saw the widespread adoption of unsupervised ML techniques, leading to the rise of deep learning and generative AI from the 2010s onwards.

In the past three years, LLMs have taken centre stage, particularly within big-tech companies. Models like BERT [7], LLaMA 1 and 2 [24, 25], OpenAI's GPT series [4, 23], and Google's Gemini and Bard [1] have revolutionized our interaction with AI-driven technologies. These models, built on AI principles and neural networks, possess the ability to understand and generate human language, making them invaluable across various domains.

The rapid development of LLMs is evident in the number of models released in recent years. Between 2019 and 2021, more than 200 LLMs were published, covering 109 languages and 28 domains [22, 29]. This surge reflects the growing interest and investment in LLMs, driven by their potential to perform a wide range of NLP tasks with remarkable accuracy. A key factor enabling this rapid development is the ability of LLMs to perform zero-shot and few-shot learning. Zero-shot learning allows models to handle tasks without any specific training data [15], while few-shot learning enables them to generalize from just a few examples or instructions [4]. These capabilities reduce the need for extensive dataset collection and annotation, making LLMs highly adaptable and versatile.

Natural language prompting, a technique that leverages the rich semantic and syntactic knowledge embedded in LLMs, has further expanded their applicability. By providing natural language instructions, users can elicit desired behaviours from these models without the need for code modifications or retraining. This has proven effective for a variety of tasks, including text classification, translation, and summarization [29].

2.2. Onboarding - Current Approach, Relevant Case Studies and Research

To understand what problems this thesis tries to tackle, we first have to analyze what onboarding is, and how is it used in practice by software companies.

In Human Resources (HR), onboarding is the process from the job offer, which sets a new employee into the company's affairs, culture, procedures, technical setup, and team dynamics. In software engineering, it also refers to getting familiar with the codebase and any proprietary technologies that the company may use. Onboarding starts from the job offer and continues until the new hire is fully integrated, to contribute effectively as an individual or within the team. Through this paper, when referring to onboarding at a software engineering project level:

Onboarding is generally understood as getting to know a new project, to which a new team member has not worked before, and its structure, business logic, architecture, deployment, cloud resources, and other important aspects are not known by the onboarded.

In the case study by Ju et al. [14], the scholars highlighted that the onboarding experience of a developer directly relates to productivity and job satisfaction, among other outcomes, in either the short or the long term but has a strong emphasis on **learning**, **confidence-building**, and **socialization**. The same conclusion was drawn from an interview with 32 new developers entering a team of software engineers and engineering managers at one big-tech company, namely Microsoft.

They noted three predominant strategies for onboarding based on the interviews:

- **Simple-Complex Strategy** It involves a progressive increase in task complexity with time. A new software engineer will be assigned simple tasks that will ease him or her into the tools, processes, and code base used by their team. As the developers become more comfortable and knowledgeable, they will be assigned tasks of increasing complexity over time. In this regard, it is important to note that the success of this strategy depends on the support accorded by the team, mentors, and managers in the provision of guidance and feedback. This assures easy passage through increasingly challenging assignments and further maintains assurance with a steady learning curve.
- **Priority-First Strategy** This assigns tasks based on their priority within the project, and hence, new developers can start contributing immediately to critical areas. This approach demands strong communication and support on the part of the team so that new developers do not become overwhelmed by the complexity or importance of these tasks. Frequent check-ins with supportive feedback from the experienced members of the group or mentors will help new developers work through high-priority tasks successfully and effectively integrate into the team dynamics.
- **Exploration-Based Strategy** Focus on the assignment of ill-defined tasks characterized by high levels of uncertainty and encourage new developers to explore and define the scope of their work. This strategy works best for senior developers or teams in which innovation and flexibility are key features. The team's support is crucial, and it requires a lot of discussion, processes of brainstorming, and a culture that welcomes inquiries and the sharing of knowledge. This open communication serves to mitigate the risks of working with uncertainty and facilitates a creative, shared and social working environment.

The authors conclude that onboarding is a long-term process because it is not the complexity of tasks, uncertainties, and contexts that shape the onboarding. Other tasks that have higher uncertainties might either facilitate the learning and improve the communication within your team or can delay the process of onboarding when the support is absent. Furthermore, the authors provide some guidelines on how the onboarding process should look like, together with the recommendations, they state that the productivity, stress, and happiness of a new developer can be managed through good onboarding strategies and possibly **automated support**.

In another study, Buchan et al. [5], authors conducted research into Agile Software Development teams using surveys on software practitioners from eight different organisations, Repertory Grid Technique to map the contributions of various onboarding techniques to specific onboarding goals and literature review. Similarly with the study conducted in Microsoft by Ju et al. [14], the authors observed that the main pillars of onboarding in the software engineering field are **mentoring**, **peer support**, **socializing activities**, **training and workshops** and **practical engagement**.

In a similar case study discussing the onboarding of software engineers, Britto, et al. [3] analyzed the process in three globally distributed legacy projects. Similarly to the cases above, the short-term outcomes associated with adaptation to new jobs by new employees were explored by the authors. Moreover, they analyzed self-efficacy, role clarity, and social integration, but also knowledge of the culture the employee is expected to have after finishing the onboarding. In this study, the scholars conclude that globally distributed projects are challenging and require long, **hands-on mentoring** periods with peer-assigned colleagues to assist the new employee. Coaching and mentoring are essential but they **set back** a mentor's productivity. The authors also concluded that the current onboarding techniques need enhancement since job orientation is usually neglected, especially in distributed teams.

In a subsequent study, Britto et al. [2] analyze, evaluate, and strategize the onboarding of software developers in large-scale globally distributed projects. The present paper resumes the study published in 2018 by the same authors [3], where the performance of the onboarding strategies was evaluated in a real-life scenario. For example, the authors conducted a case study of Ericsson with a 20-year project in production by analyzing the newcomers in remote working sites. The academics note again how mentor proximity, training approaches, task allocation, and team stability are the poles of successful onboarding in large-scale distributed projects. The study also identified that remote mentoring, training approaches not aligned with cultural expectations, complex tasks early in the onboarding process, and team instability impacted newcomer performance negatively.

The current onboarding method in software engineering is mentor-focused, which heavily burdens the mentor with additional responsibilities that often reduce the productivity of the mentor. Our proposed solution to tackle that is to transition from human peer-assigned mentors to an autonomous chatbot, the Onboarding Buddy. The chatbot should be made capable of handling routine queries and support functions so that the mentors keep busy with their productive roles while new hires get the necessary guidance and hand-holding to onboard seamlessly.

2.3. Large Language Models Virtual Assistants

Large language models like GPT-4, and Gemini, are trained over massive amounts of textual data and learn effective ways to understand and generate text in a human-like manner. In this regard, these models have been repurposed to understand the rate and analyse code, which has made them indispensable in the field of software development.

Virtual assistants powered by LLMs are generally implemented as plugins to IDEs, such as IDEA IntelliJ IDEs [1] or Visual Studio Code [2]. A plugin is a software add-on that increases the functionality of an application. In software development, plugins are commonly used in extending the features of the IDE, making it more powerful and appropriate to the developer's needs. These can vary from simple tools, such as linters and formatters, to AI-powered programming guides.

Some plugins that work like an AI virtual assistant in software development include:

GitHub Copilot[3]: An AI-driven plugin integrated into IDEs, such as Visual Studio Code, from GitHub. It provides context-aware code suggestions and autocompletion, and at times, even generates whole functions or code snippets based on the developer's input. Copilot enables developers to write code faster and more effectively with the enormous programming knowledge built into the model.

JetBrains AI[4]: An intelligent assistant provided within JetBrains IDEs like IntelliJ IDEA and PyCharm. Some of these include smart code completion, code explanation, and real-time code quality analysis. JetBrains AI is designed to understand the code context in a bid to provide actionable insights to developers, easing their job of debugging and improving the quality of their code over time.

These plugins are generally easy to install and configure within IDE, so that developers may instantly leverage all productivity tools with code generation and project navigation utilities.

In this manner, the virtual assistant extends the scope of functions for a developer's aid and can encompass functionalities like:

- **Autocompletion**: LLMs can predict and suggest completions of code within the context of the current file, further saving developers time typing and even reducing potential syntax errors.
- **Code Explanation**: The developers can ask for explanations regarding specific pieces of code. The LLM would produce an explanation regarding the goals and expected results of the code that is human-friendly.

[1] IDEA IntelliJ - `https://www.jetbrains.com/idea/`
[2] Visual Studio Code - `https://code.visualstudio.com/`
[3] GitHub Copilot - `https://github.com/features/copilot`
[4] JetBrains AI - `https://www.jetbrains.com/ai/`

- **Bug Detection and Fixes**: Some advanced LLM-based assistants are capable of detecting potential bugs or vulnerabilities in the code and proposing fixes, which can help in writing better quality code.
- **Documentation Generation**: LLMs can generate documentation in the codebase, including comments and README files, improving project maintainability.

Despite their performance, traditional LLM-based assistants may underperform in the case of domain-specific complex problems or when needed information is distributed across sources. This has resulted in many recent advances in more sophisticated systems that combine the strengths of LLMs with other AI techniques, for example, Retrieval-Augmented Generation (RAG).

2.3.1. Retrieval Augmented Generation

Retrieval-augmented generation (RAG) is an approach that empowers generative language models by interposing a retriever capable of scanning and adding relevant external information to the generative process to make it possible to produce both accurate and context-sensitive responses [10, 16]. RAG will retrieve relevant documents, code snippets, or another kind of resource which can provide more context to the LLM.

2.3.2. How RAG Usually Operates

In a more general sense, the RAG process goes through several critical processes that allow a system to extract and manipulate the information required. These steps are as follows: document chunking, embedding, and storing in a vector database:

- **Document and Code Chunking**: The first step in implementing RAG is to break down the available documents, codebases, and other relevant resources into smaller, manageable pieces known as "chunks." This chunking process is essential because it allows the system to retrieve specific parts of a document or code that are relevant to the query rather than retrieving entire documents that might contain a lot of irrelevant information. For example, one can chunk a long technical document into paragraphs or sections, or a codebase can be chunked into functions, classes, or even individual lines of code.
- **Embedding the Chunks**: After chunking, each chunk is embedded into a high-dimensional vector representation. Normally this can be done by using a pre-trained model like BERT [7] or using a fine-tuned embedding model trained on some related data of interest, for example, code-specific embeddings. Embedding is done in a process to represent the semantic content such that it allows the system to be easily compared with other chunks and, specifically, the query itself. These vectors capture the meaning of the text or code, therefore helping the system make efficient retrievals based on semantic similarity.
- **Storing in a Vector Database**: The embedded chunks are then stored in a vector database. A vector database is a specialized database optimized for the storage and retrieval of high-dimensional vectors (e.g. FAISS [8]). The database is indexed so that when a query comes, the database can quickly search for the most similar chunks with nearest neighbour searches. This step is important to guarantee that the system can quickly retrieve relevant information at need.
- **Query Understanding and Retrieval**: When the system receives a query from a developer, it first processes the question to have an understanding of the given intention and then creates an embedded vector for the representation of the question. This vector is compared to vectors in the vector database to identify the most relevant chunks. The system fetches such chunks, which are the most semantically similar to the query, thus ensuring a very high relevance of the information retrieved to the developer's need.
- **Contextual Response Generation**: The retrieved chunks feed a generative language model of the kind of GPT-4, which uses the context given by these chunks to come up with a response. Such a response will be tailored to the query's and project's context to a much greater degree than any standalone LLM could ever provide.

The final output is a natural language reply to the developer's question, rich with project-specific information and relevant details, which are extracted from the documents or code snippets.

2.3.3. Advantages of RAG in Software Development

RAG has been validated as an industry standard [10] in terms of LLMs because of the following reasons:

- **Project-Specific Responses**: The retrieval mechanisms of RAG enable it to generate highly project-specific responses, considering project-specific features of the codebase, documentation, and development environment.
- **Dynamic and Fresh Information**: Since RAG queries to retrieve the freshest and most relevant information available in the stored chunks, it ensures responses will be contextualized within the actual state of affairs for the project, so there will be little risk that they are outdated or on irrelevant advice.
- **Better Performance on Complex Queries**: Since more than one piece of information can be integrated into a single output representation, RAG is well fit for answering complex or multifaceted questions by generating holistic answers that cover multiple aspects of the query.

As mentioned in the section 2.2 the typical onboarding system in software development requires static documentation, video tutorials, and in-person training sessions. That is to say, while these can be highly effective ways to achieve onboarding, they get time-consuming and may not adapt to the unique needs of individual developers or projects.

Virtual assistants are more personalized, interactive, and dynamic combined with RAGs. Through such systems, personal guidance can be obtained, with instant explanations and relevant information that provides for a significant portion of the onboarding process to be automated. For this purpose, a new developer will be able to communicate with an assistant using the chat interface for inquiries into a project, codebase, or anything else that may be of concern. In case new information needs to be gathered from RAG, it can query it for what it needs to know and obtain a helpful, informative response in return.

Moreover, such aides may continuously adapt to the evolving nature of the software project and keep updating responses or suggestions as new features are added or the codebase changes. This continuous learning capability ensures that the information developers receive is relevant and up to date.

3

Plug And Play Experimental Set-Up - The Onboarding Buddy Platform Architecture

Along with the stated goal to create an Onboarding Buddy, we also wanted to provide the research community with a plug-and-play experimental set-up platform that can be adjusted, deployed and used in future LLM chat studies without any major effort. This is why we provide our experimental platform with features such as easy set-up and deployment for all the sub-components, scopes and purposes. For this, we divided the project into two entry points; the Onboarding Buddy Web platform and the Onboarding Buddy IDE platform. This allows us to use the platform from inside an IDE, but also from just a web browser, allowing the app to be accessed on any device with internet access. This chapter presents only the architecture and architecture choices of the system and does not dive into the details about how the system works, these details being presented in chapter 4. The overall system architecture can be observed in figure 3.1 and the details of each sub-component will be elaborated in this chapter. This plug-and-play set-up is composed of 3 different main components:

- **Experimental Backend** - responsible for data processing, LLM orchestration, tracing and data storage.
- **Experimental Chat UI** - responsible for user interaction, task presentation and data collection.
- **Experimental Plugin** - responsible for UI wrapping, and user interaction tracking (telemetry).

3.1. Experimental Backend

The experimental backend provides the core functionality to our experimental chat UI and experimental plugin. The backend infrastructure of the project is based on a Firebase[1] like architecture as seen in the project structure in Figure 3.2. This ensures a stable, scalable, and secure environment. On top of that, this allows the developer to use the Firebase Emulators to access a 1on1 replica of the deployed services locally, in the development environment. Each functionality is defined as a containerized serverless cloud container with its corresponding infrastructure as code definition (IaC).

Firebase offers a full suite of backend tools that easily integrate with Google Cloud services; therefore, the setup is very flexible and efficient at the same time. The indispensable components of the backend are the Firestore NoSQL database, Google Cloud Functions for serverless computation, Blob Storage for large data objects, and LLM tracing through LangSmith [2]. This infrastructure supports our application to auto-scale. Furthermore, this architecture allows us to facilitate the system architecture interaction with a custom orchestrated Large Language Model using the Langchain[3] library.

[1] Firebase - https://firebase.google.com/
[2] LangSmith - https://www.langchain.com/langsmith
[3] Langchain - https://www.langchain.com/

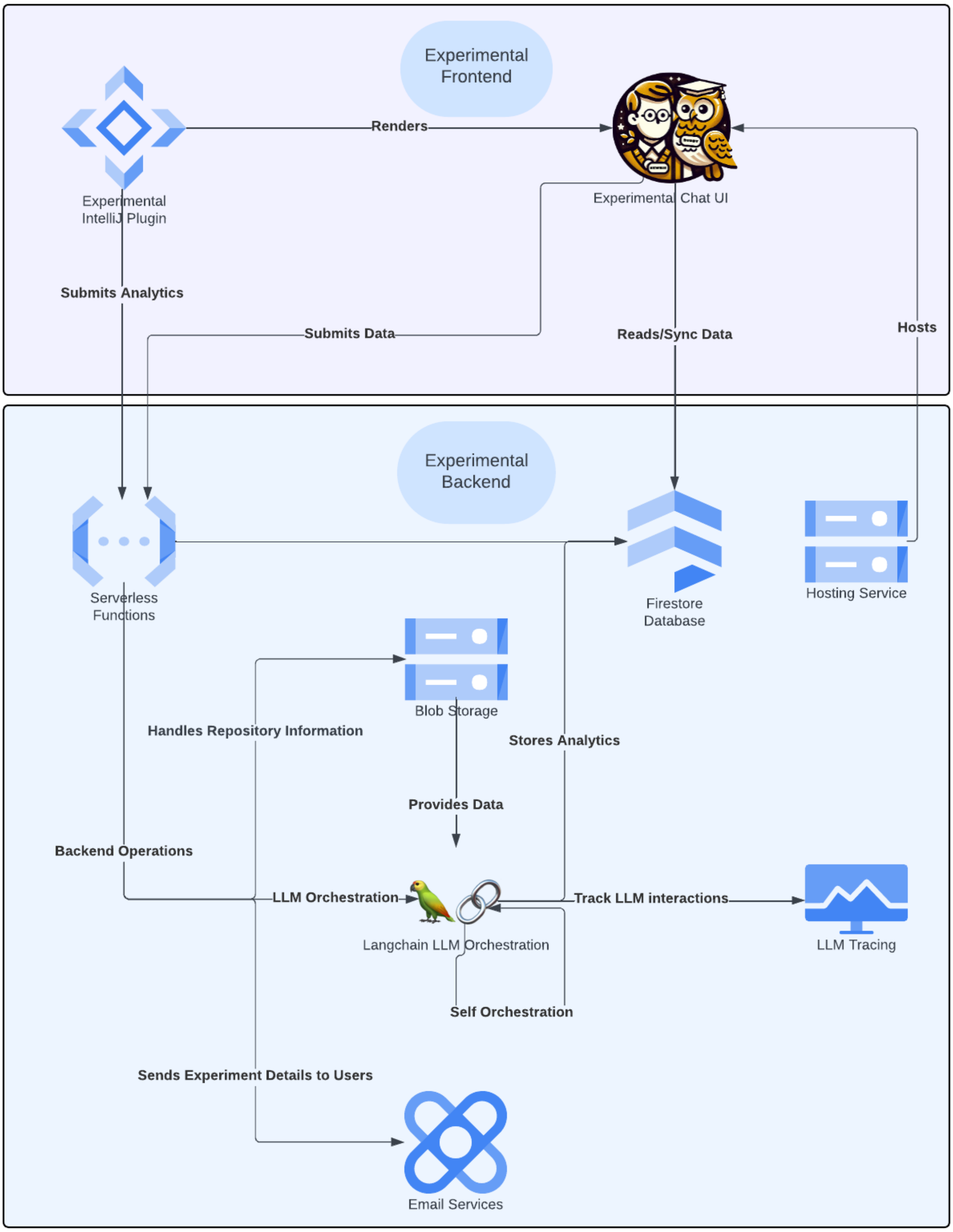


Figure 3.1: Onboarding Buddy System Architecture Diagram

```
OnboardingBuddy Backend
├── functions
│   ├── agents
│   │   ├── contextualize_agent.py
│   │   ├── onboading_agent_openai.py
│   │   ├── onboarding_agent.py
│   │   ├── tools
│   │   │   ├── missing_file_retriever.py
│   │   │   └── semantic_code_search.py
│   │   └── utils
│   │       ├── assistant_project_uploader.py
│   │       ├── firebase_project_uploader.py
│   │       ├── project_embeeding_creator.py
│   │       ├── retriever_loader.py
│   │       └── traces_export.py
│   ├── auth
│   │   └── user_checks.py
│   ├── data
│   │   ├── chats
│   │   │   └── firebase_chat_entity.py
│   │   ├── constants.py
│   │   ├── email.py
│   │   ├── prompts.py
│   │   └── utils.py
│   ├── functionsDef.py
│   ├── main.py
│   ├── models
│   │   └── gpt.py
│   ├── processors
│   │   ├── prompts
│   │   │   ├── constants.py
│   │   │   └── prompts.py
│   │   └── step_message_processor.py
│   └── util
│       └── outputs.py
├── firebase.json
├── firestore.indexes.json
├── firestore.rules
├── remoteconfig.template.json
└── storage.rules
```

Figure 3.2: File structure of the Experimental Backend project.

3.1.1. Firestore Database

The Firestore database is a NoSQL, document-oriented database provided by Firebase. Being a NoSQL in nature, Firestore allows flexibility in data structure; data are saved in documents that are grouped into collections. Each document holds some fields containing strings, numbers, arrays, or even nested objects.

Very useful with this setup is the real-time sync of Firestore: it would automatically update all connected clients instantly, a feature that is required for a live LLM chat. This becomes a critical feature in ensuring that the data remains consistent when multiple users or processes interact with the same data. In addition, Firestore, along with Firebase Authentication, ensures secure access to the data, allowing only authorized users to conduct read or write activities.

The database has three different purposes managing users, managing tasks, and managing study groups with the following structure:

- **users/{user_id}** - contains the user data along with their chat history.
  - chats (sub-collection) - contains the user chat interactions with Onboarding Buddy.
  - activities (Array[Object]) - contains the activities the user performed using the Onboarding Buddy Plugin (section 3.3).
  - demographics (Object) - contains the user demographics submitted when starting interacting with Onboarding Buddy.
  - githubID (string) - user GithubID used to give the user access to the experiment's repository.
  - group (integer) - user's study group.
  - os (string) - user's operating system that is used during the tasks.

One very important mention is the structure of the subcollection chats under each user:

- **users/{user_id}/chats/{chat_id}** - contains the user interaction with the chat and the feedback associated with the chat session.
  - **feedback (Object)** - contains feedback related to the chat interaction.
  - **started_at (timestamp)** - the date and time when the chat interaction started.
  - **finished (boolean)** - a flag indicating whether the chat interaction was completed.
  - **ended_at (timestamp)** - the date and time when the chat interaction ended.
    - **messages (sub-collection)** - contains individual messages exchanged during the chat session.
      - prompt (string) - the prompt or question asked by the system or user.
      - type (string) - the type of message (e.g., "steps", "question").
      - chat_id (string) - the ID of the chat session.
      - sub_messages (Array[Object]) - any sub-messages or follow-up messages related to the main message.
      - answer (string) - the response or answer provided during the chat.
      - id (string) - the unique identifier of the message.
      - dependsOn (string) - the ID of any preceding message that this message depends on.
      - user_track_id (string) - a tracking ID associated with the user's session or interaction.
      - forward_refs (Array[string]) - any forward references to the elaboration of the "steps" answer type (more details in the implementation section).
      - role (string) - the role of the sender (e.g., "AI", "User").
      - steps (Array[string]) - any specific steps outlined in the message.
      - timestamp (timestamp) - the date and time when the message was sent.

    - question (string) - the specific question asked in the message, if applicable.
    - processedMD (Markdown) - the processed markdown associated with the message.

- **tasks_collection/{task_id}**
  - title (string) - the title of the task.
  - description (string) - description of the task.
  - difficulty (string) - the difficulty of the task.
  - file_references (string) - the files that concern the task.
  - task_details (Markdown) - the description of the task.
- **groups/{groupCounts}** - contains data about the total number of users assigned to an experimental group.
  - gropu0 (integer)
  - group1: (integer)

The access to the Firestore can be configured into the .firestorerules file. This allows a high level of customization of the security rules. For example, our backend uses rules that allow only read access to the database from the users, up until a certain date, and write access only to the elevated environments such as the serverless functions presented in the subsection 3.1.3. These rules can be observed in figure 3.3.

```
rules_version = '2';

service cloud.firestore {
  match /databases/{database}/documents {
    match /{document=**} {
      allow read: if request.time < timestamp.date(2024, 10, 20);
    }
  }
}
```

Figure 3.3: Firestore Security Rules

3.1.2. Blob Storage

Blob storage is used to handle and store big objects like repository snapshots, vector database snapshots derived from these repositories, and user manuals. Firebase integrates directly with Google Cloud Storage, which provides a way of storing large data files in an easily scalable and secure manner. This system allows for efficient retrieval and management of data, something that is critically important, considering the size and complexity of the files at stake.

Having the repository snapshots stored within Blob Storage is important for recording the history of the state of the repository at any given time. These snapshots are used also to populate the vector database with data to perform several machine-learning tasks such as similarity search. Storing these snapshots in Blob Storage enables them to be accessed easily and allows for quick retrieval when needed.

Similarly to the Firestore security rules, the access to the blob storage can be configured into the storage .rules file. This allows a high level of customization of the security rules. For example, our backend uses rules that block all external access to the blob storage, allowing only access from inside the serverless functions as seen in figure 3.4.

```
rules_version = '2';

// You can craft rules based on data in your Firestore database e.g. for admin users
// defined in the Firestore
// allow write: if firestore.get(
//    /databases/(default)/documents/users/$(request.auth.uid)).data.isAdmin;


service firebase.storage {
  match /b/{bucket}/o {
    match /{allPaths=**} {
    // no access, only elevated environtments can access it (e.g. functions)
      allow read, write: if false == true
    }
  }
}
```

Figure 3.4: Blob Storage Security Rules

3.1.3. Business Logic - Serverless Functions

Google Cloud Functions implement the backend logic; they are serverless computing services provided via Firebase. This means functions can be triggered by an HTTP request, a potential change occurring in the database, or a state change.

All our serverless functions are written in Python and make use of the Flask framework for HTTP requests that are interpreted as containerized serverless functions. Flask is a very lightweight web framework that includes tools to easily create web applications and APIs. Flask allows us to set up RESTful endpoints, which help facilitate interactions between the frontend and backend, so communication and data interchange go smoothly.

The orchestration of the Large Language Model-based functions is enabled through Langchain's integration. Langchain provides tools to handle and interact with large language models, thereby enabling the backend to process complex language tasks. This orchestration capability is vital since the application largely depends on LLMs for its core capabilities. More details about the LLM-based functions are presented in the chapter 4.

Serverless functions are able to auto-scale, being represented by docker containers that are provisioned to the users when needed. In other words, a function can run with 0 to a predefined number of instances at the same time. When there is no traffic, the functions will scale to 0, with no billing occurring. This allows us to create a highly scalable and maintainable backend at very low costs.

This serverless architecture corresponds to a highly decoupled system corresponding to the principle of separation of concerns (SoC). Each function runs independently of the other function in terms of data access, code execution and environment. Furthermore, each function is defined as Infrastructure as Code (IaC) thus being able to pre-configure the size of each function instance. For example, the LLM-based functions in our backend are assigned 1GiB of memory and 0.583 vCPU with a 4 minutes timeout and the low computation functions such as database writes are assigned only 256 MiB of memory and 0.167vCPU with 30 seconds timeout. If the functions do not finish executing before the timeout is reached, the instance will automatically deprovision the resources. One such function definition can be seen defined throughout the **on_call** annotation in the code snippet presented in figure 3.5 that handles the user registration in our backend.

```python
@https_fn.on_call(
    memory=512,
    timeout_sec=60,
    cors=options.CorsOptions(
        cors_origins=CORS_ORIGINS,
        cors_methods=["GET", "POST", "OPTIONS"],
    ),
    secrets=["AWS_ACCESS_KEY_ID", "AWS_SECRET_ACCESS_KEY", "GITHUB_SECRET"],
)
def register_user(req: https_fn.Request):
    from functionsDef import registerUser

    params = req.data
    registerUser(params)
    return {"status": 200}
```

Figure 3.5: Python Serverless Function definition to register a user with specific settings and key secrets.

3.1.4. LLM Tracing

Langsmith is used for tracing interactions with our Language Model orchestration. This tracing serves a dual purpose: to analyze how users interact with the LLM and assistance in evaluating the orchestration.

Langsmith shows us how users interact with the system and with the LLM, thereby capturing important data at each touch point. In turn, we can trace user inputs with corresponding model outputs to find out what exactly a system was being used for, what kinds of queries are common, and if the model-generated responses meet user expectations. This data will enable us to have a better understanding of user behaviors, identify patterns of interaction, and enhance users' experiences by fine-tuning LLM responses or improving the user interface based on real-world usage.

The same kind of tracing can also prove helpful in tracking how relevant the outputs of the LLM are in terms of accuracy and can thereby help identify more clearly the areas in which it might be further trained or tuned. For example, if users quite often reformulate or specify their queries, this means a symptom that the initial answers of LLM can be unclear and too general, and this will give us specific feedback that can be implemented to orchestrate better precision.

In addition to all this, an orchestration layer managed by Langchain might offer extra insights such as how often the LLM used certain tools (e.g. RAG), time to first token or cost analysis. By monitoring each step of the orchestration process, its LLM invocation, and interaction among the different components, we can estimate efficiency and performance. This is an important test for ensuring that serverless functions, Firestore database, and Blob Storage all work together in the best possible way, and that orchestration logic efficiently uses the LLM functionalities.

This helps to pinpoint in orchestration any potential bottleneck, performance issues, or inefficiency. For example, we can monitor the latency of requests, the load on the system, and how resources are being allocated during usage. Based on such information, we optimize the orchestration pipeline in a way that reduces latencies further and lowers resource consumption while maintaining scalability with rising demands.

Langsmith then brings together these user interaction insights with orchestration performance data to provide a 360° view of how the LLM is doing in the broad backend landscape. This enables us to constantly tune, not only the user experience but also the technical architecture of our system.

3.1.5. Hosting

Our Firebase backend also takes care of hosting our UI presented in section 3.2. The main advantage of this hosting choice is the automatic provision of SSL/TLS certificates and the automatic integration with the aforementioned Firebase service (e.g. request authenticity verification).

3.1.6. Email Services

Our Firebase backend also uses an external email service, Amazon Simple Email Service[4] (SES) to send emails to the users that enrolled on the experiment. The SES packs and delivers to the users the ”Experiment Manual” as seen in Appendix A, their authentication token and some more information that is getting them ready to start their study submission. The email services are using Google Secret Manager[5] to access the private keys for the Amazon Web Services (AWS) and GitHub at deployment time.

For more details about code structure and architecture intricacies, you can access the backend repository[6].

3.2. Experimental Chat UI

The Experimental Chat UI is the component that directly interacts with the end user. The UI connects to the backend presented section 3.1 and allows the user to chat in a ChatGPT[7]-like style with our Onboarding Buddy. The UI is also part of the entry point of our study, by ensuring that the users can perform their tasks and give feedback. Since one of our goals for our experimental platform is to provide the research community with an easily maintainable codebase we chose that our fronted will use the Vue.js[8] framework in combination with the material design library Vuetify[9] and Vuefire[10] a Vue.js integration to communicate and synchronize with our backed presented in Section 3.1.

Vue.js is a progressive JavaScript framework for creating user interfaces. It has the reputation of being simple yet powerful, which makes it considerably scalable. By now, it has become very popular with developers and has more than 208,000 stars on GitHub[11]. Moreover, it is the third most starred project overall on GitHub for the Typescript category [17]. Vue is one of the most starred Javascript-based UI frameworks on GitHub, which represents widespread use, active support and active maintenance.

Vue was chosen for the current application based on a component-based architecture and reactive data binding that, in turn, eases application development on projects no matter their scale. It also effectively manages updating of the DOM by using virtual DOM, which actually computes the changes and then updates only the necessary parts of the actual DOM. In that way, user interface responsiveness and performance are guaranteed, hence critical for a real-time application like the Experimental Chat UI.

We further integrated Vuetify, which is an advanced full-featured component library for Material Design made explicitly for Vue.js, giving our UI enhanced design and experience. It helps us maintain the principles of Material Design from Google: a consistent and visually appealing interface that is also highly customizable. Material Design was introduced in 2014 by Google and was a design language focusing on simplicity, clarity, and responsive interaction based on real-world materials such as paper and ink [12]. It goes by these principles: Vuetify is following these principles for a consistent, user-friendly UI.

Our experimental platform is equipped with an intuitive, efficient, and maintainable user interface. By combining Vue.js with Vuetify and VueFire, we have crafted a strong UI for our experimental platform, which allows the user to interact smoothly with the system. This set of tools used in the front-end part also means that the project will be well prepared for future expansion and maintenance, which turns it into a valuable resource for the scientific community.

The project contains different Vue components and sub-components that handle user interactions. For example. the chat component is composed of 7 UI sub-components as seen in figure 3.6. A main chat session is composed of the user starting a task and being presented with the task prompt as seen in figure 3.7a. After this, the user can ask their question or access a curated topic or question from the

[4] Amazon SES - https://aws.amazon.com/ses/
[5] Google Secret Manager - https://cloud.google.com/security/products/secret-manager?hl=en
[6] Backend Repository (needs registration) - https://github.com/OnboardingBuddy/OnboardingBuddyBackend
[7] ChatGPT - https://chatgpt.com/
[8] Vue.js - https://vuejs.org/
[9] Vuetify - https://vuetifyjs.com/en/
[10] Vuefire - https://vuefire.vuejs.org/
[11] Vue.js Github - https://github.com/vuejs/vue
[12] Material Design - https://m2.material.io/design/introduction

”How can I help?” menu in figure 3.7b. After the user asks a question, the answer is processed and the user is returned an answer through our custom markdown rendering component as seen in figure 3.7c. More details about the internal works of the Onboarding Buddy are presented in chapter 4.

Furthermore, the business logic is located and structured as multiple Pinia State Managers[13] allowing us to structure the code by purpose (e.g. Chat Store, Task Store...etc.).

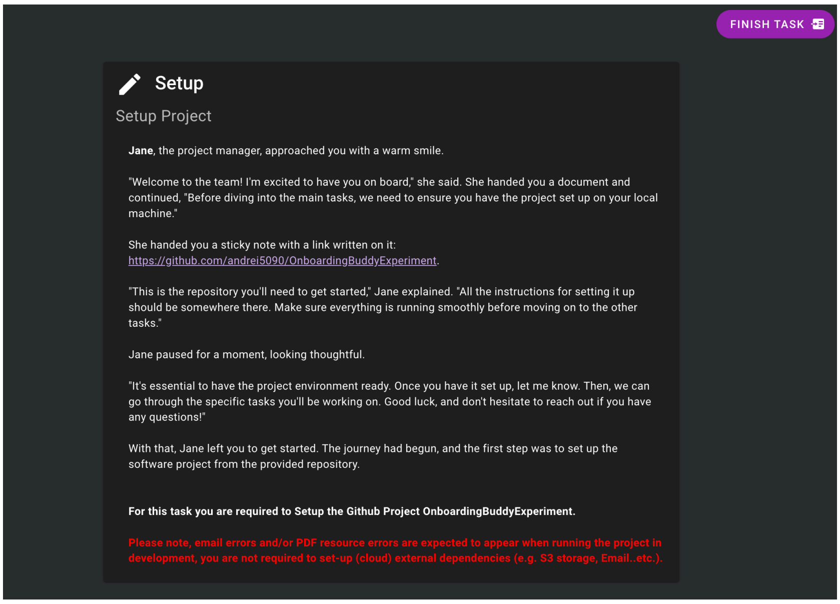
For more details about code structure and architecture intricacies, you can access the UI repository[14].

[13] Pinia State Manager `https://pinia.vuejs.org/`

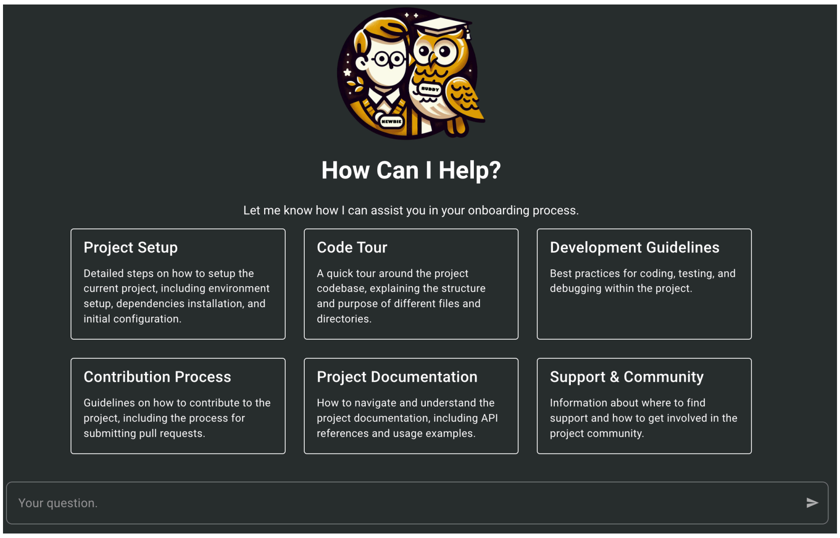
[14] UI Repository (needs registration) - `https://github.com/OnboardingBuddy/OnboardingBuddyFrontend/`

- Experimental Chat UI Root
  - src
    - App.vue
    - components - Contains the reusable UI Components
      - AppFooter.vue
      - AppNav.vue
      - ChatPage.vue
      - EndPage.vue
      - RegistrationPage.vue
      - chat
        - QuestionChat.vue
        - choices
          - ChatChoices.vue
        - constants
          - chat-buttons-constants.ts
        - feedback
          - TaskFeedback.vue
        - subcomponents
          - ChatDialog.vue
          - ChatInput.vue
          - FinishTaskPopUp.vue
          - MDRender.vue
          - TaskDisplay.vue
      - users
        - ChatSelector.vue
        - UserLogin.vue
        - consent
          - ConsentForm.vue
          - UserDataSelectors.vue
    - layouts - Contains the Layouts of the app such as the header or footer
    - main.js
    - pages - Contains the pages that are rendered inside of the Layout
      - index.vue
      - license.vue
      - register.vue
    - plugins - Contains the Plugins used by the app such as Firebase, Vuetify
    - router - Contains the Router configuration
    - stores - Contains the Business Logic and App State controls

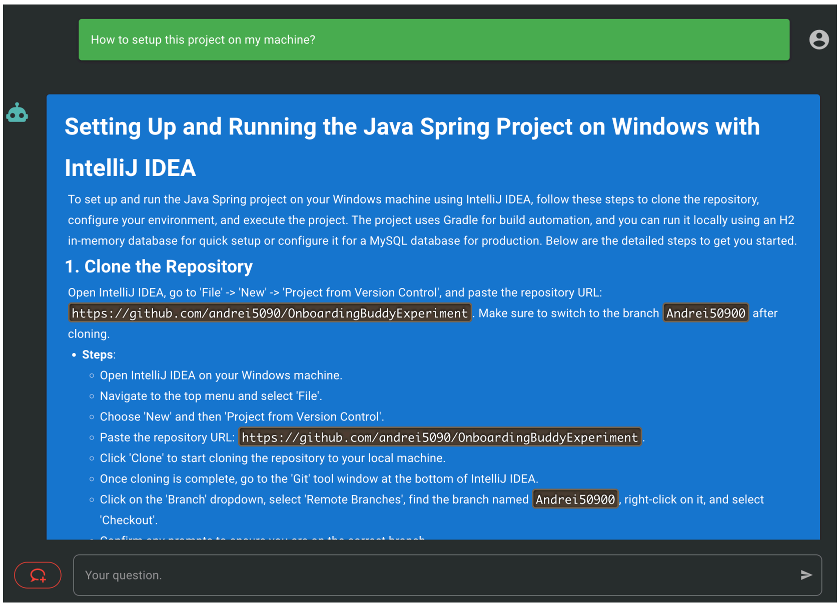
Figure 3.6: File structure of the UI project



(a) The task presented to the user



(b) The chat selector and the question-asking panel



(c) The answer returned by the Onboarding Buddy when requested to provide a project set-up guide.

Figure 3.7: The Main Chat-Related UI Components

3.3. Experimental Plugin

The plugin represents the main entry point of our users into the experimental platform. The main purpose of the plugin is to wrap the UI presented in the section 3.2 into the IntelliJ IDEA IDE and provide analytics to our backend presented in the section 3.1. The Plugin is based on Jetbrains's IDE Plugin SDK [15] and it is written in Kotlin, closely following JetBrain's architectural patterns for plugins. The sole purpose of this plugin is to pre-authenticate the user and provide telemetry such as user onboarding activity inside of the IDE. All the other Onboarding Buddy functionalities are performed through the UI application that is rendered inside of the plugin.

Experimental Plugin Root

- .github - Contains the CI/CD pipelines
- gradle - Contains the Gradle Configuration such as Plugin Configuration for the CI/CD pipelines
- src/main/kotlin/com/github/andrei5090/onboardingllms
  - actions
    - GetSystemDataAction.kt
  - extractors
    - LanguageVersionExtractor.kt
    - SystemInfoExtractor.kt
  - handlers
    - BrowserHandler.kt - Responsible rendering the UI presented in section 3.2
  - listeners
    - ApplicationActivationListener.kt
    - BulkFileListener.kt
    - DataAccumulator.kt
    - FileDocumentManagerListener.kt
    - FileEditorManagerListener.kt
  - toolWindow
    - WindowFactory.kt
- ui - The UI source code presented in Figure 3.6
- resources
  - META-INF - Contains the JetBrains Plugin SDK References
    - plugin.xml
    - pluginIcon.svg

Figure 3.8: File structure of the Experimental Plugin

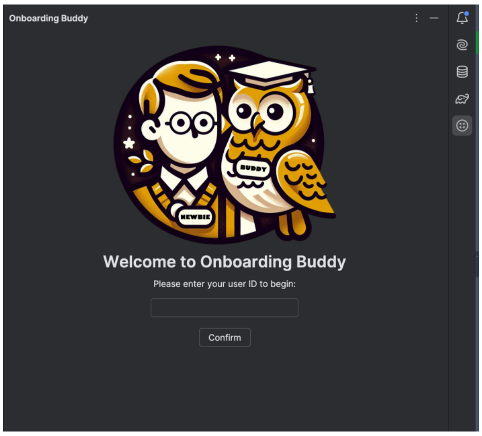
The architecture of the plugin is very simple, as seen in figure 3.8, the following components representing the core functionality of the plugin application:

- **Metrics and Analytics**: Files such as DataAccumulator.kt gather data about user activities (e.g.,

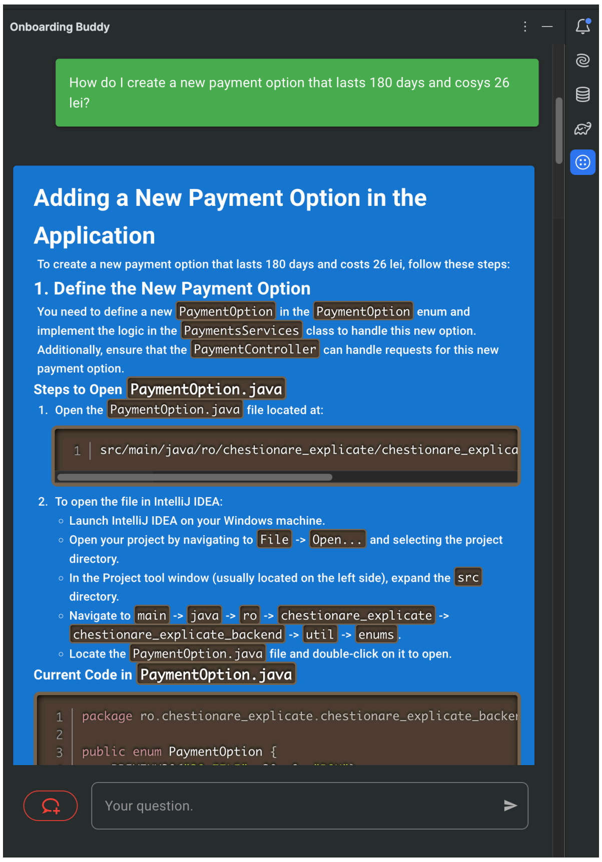
[15] Jetbrains Plugin SDK - `https://plugins.jetbrains.com/docs/intellij/welcome.html`

file edits, application activations) and send this information periodically to the backend.

- **User Authentication**: Pre-authentication occurs before users interact with the UI. The WindowFactory.ks renders an authentication page inside the IDE as seen in figure 3.9a ensuring secure access to the platform.
- **UI Rendering**: The BrowserHandler.kt class is responsible for rendering the experimental platform's UI within the IDE as seen in figure 3.9b. The actual UI code is located in the UI directory.
- **JetBrains SDK Integration**: The plugin.xml file in the resources/META-INF directory contains key plugin configurations, linking the plugin to the JetBrains Plugin SDK. This file defines how the plugin integrates with IntelliJ IDEA's ecosystem.



(a) The Plugin Authentication Page rendered inside of the IDE



(b) The Plugin Chat UI rendered inside of the IDE

Figure 3.9: User Interface Components Rendered Inside the IDE through the plugin

3.4. Development and Deployment

The key goals of the development and deployment strategy for the Onboarding Buddy platform were mutability and ease of deployment. It must be easy for future researchers or developers to deploy the platform for their experiments own LLM-Chat experiments. The architecture was designed with respect to making a production environment easily replicated out of development while at the same time leveraging CI/CD pipelines for automated, reliable deployments.

The system is designed so that the whole platform can be launched in a few commands as a replica of the production environment so that development and testing can be done in the fastest way. This will ensure the developers test changes locally in an environment that most resembles the setup of production with backend services and data flow.

3.4.1. Experimental Development Environtment

Frontend, or Experimental Chat UI, the development environment can be easily launched. To install and start your local environment, if you are a developer, you need to execute the following command to start the hot reload development server:

```
yarn
yarn dev
```

The configuration will automatically detect whether it is running with the Firebase Emulators, which run backend services on your computer replicating the production. The hot reload emulators can be started from the backend project by executing the following command:

```
firebase emulators --import=DATA_PLACEHOLDER --export-on-exit=DATA_PLACEHOLDER
```

This setup enables the front end to work seamlessly against the emulated backend and provides them with a full production-like experience for testing and developing independent of any external infrastructure. The data will be exported in the ’DATA_PLACEHOLDER’ folder when the emulators are stopped so the –import command can pick the folder at the next run and reuse the saved data.

Furthermore, the backend configuration can be adjusted in the .env file to choose the parameters that the LLM orchestration will use (more details in chapter 4). Such .env file will contain:

```
OPENAI_API_KEY=KEY
OPEN_AI_DEFAULT_MODEL="DEFAULT_MODEL"
OPEN_AI_ENHANCED_MODEL="ENHANCED_MODEL"
OPEN_AI_CONTEXTUALIZATION_MODEL="CONTEXTUALIZATION_MODEL"

#LANGCHAIN LLM TRACING
LANGCHAIN_TRACING_V2=true
LANGCHAIN_ENDPOINT="https://api.smith.langchain.com"
LANGCHAIN_API_KEY="API_KEY"
LANGCHAIN_PROJECT="PROJECT_NAME"

#SETTINGS
TEMPERATURE=TEMP(value between 0.0 and 1.0)
MAX_CHAIN_OF_THOUGHT_ITERATIONS=ITERATIONS(number)

#ABOUT PROJECT
LANGUAGE="DEFAULT_PROJECT_LANGUAGE"
OS="DEFAULT_OS"

#ORCHESTRATION VARIABLES
STEP_MESSAGES_MAXIMUM_DEPTH=2
MESSAGES_PAGINATION=5
HISTORY_DEPTH=3
K_RETRIEVED_MESSAGES=5

#USER_TRACK_ID
USER_TRACK_ID=""
CHAT_ID=""

#EMULATOR SETTINGS
FUNCTIONS_EMULATOR_TIMEOUT_SECONDS=140s

```

```
#GITHUB REPO USED FOR THE EXPERIMENT (USER INVITATIONS)
GITHUB_REPO_NAME = "OnboardingBuddyExperiment"
GITHUB_REPO_USER = "andrei5090"
GITHUB_REPO_LINK = "https://github.com/andrei5090/OnboardingBuddyExperiment"

#Code Source
GITHUB_REPO_BACKEND = "OnboardingBuddyBackend"
GITHUB_REPO_FRONTEND = "OnboardingBuddyFrontend"
```

3.4.2. Plugin Development

To try out the plugin, which integrates the UI into the IntelliJ IDEA environment plus analytics, on one's local machine, the Gradle build system can be used. We can start the IDE instance locally with the plugin loaded by using the following command in the plugin project:

```
./gradlew runIde
```

This will boot up the development of the IDE, and we can test the integration of the plugin with it, the functionality, and the interactions with the UI and the backend in this local development environment.

3.4.3. Deployment

In order to ensure that the platform is scalable, robust, and reliable, automated Continuous Integration/-Continuous Deployment pipelines were set up for both the frontend and the backend. The result of this process is smooth deployments with very limited manual intervention, and at the same time, quality control does not lose its high standards.

Experimental Chat UI CI/CD Pipeline

For the Experimental Chat UI, a CI/CD pipeline takes care of all the testing and deployment automatically whenever a new feature is pushed. The pipeline does it by creating a custom deployment with a unique URL each time for each branch when a PR is triggered, letting the developer verify changes in a production-like environment before merging them into the main branch. This kind of staging environment enables one to test new features or changes in isolation. Once verified, changes are merged into the main branch that triggers the final deployment to the production domain. The automated deployment ensures the latest version of the application is always live, with no downtime, and the unique URL feature provides a deployment validation mechanism before it goes live.

Deploying the Backend with Serverless Functions

Deploying the backend services consists of serverless functions hosted on Firebase. A developer needs to execute only one command in the backend folder:

```
firebase deploy --only functions
```

This will push all the serverless functions defined in the project to Firebase. Thus, the developer will have his backend services updated and live with a minimum effort. As the infrastructure of the system is already defined as code, there is no extra configuration for further environments, nor is there any need to manually update cloud services.

Regarding the back-end side, the CI/CD pipelines compile, test, and deploy a new function when changes in the code have taken place. Hence, it always keeps the back-end up to date, and problems will be detected very early.

3.5. Access to the deployed platform and source code

To gain access to the code source and access to the deployed Onboarding Buddy platform you can register at: `https://onboarding.software/register?colab=true`. Upon registration you will receive three invite links from Github. Moreover, in another email you will receive the access code and the study manual with extensive instructions to access the deployed platform.

- Study Repository - `https://github.com/andrei5090/OnboardingBuddyExperiment`
- Backend Repository - `https://github.com/OnboardingBuddy/OnboardingBuddyBackend`
- Frontend Repository - `https://github.com/OnboardingBuddy/OnboardingBuddyFrontend`

4

Our Approach - Inner Workings of Onboarding Buddy

Now that we have presented the system architecture in chapter 3, we have to present the inner workings of our agent-centric approach to the Onboarding Buddy. In this section we will elaborate on the the LLM orchestration, prompting techniques and the technologies we used to create the Onboarding Buddy Q&A capabilities.

As stated in the system architecture chapter 3, the LLM orchestration uses the Langchain library, this is why we first have to define two very important concepts that define our approach, namely chains and agents.

In LLM orchestration using chains, the activities to be executed are fixed and immutable in a predefined order within the code, which makes them quite inflexible. After the chain is implemented, it will act exactly according to the steps strictly defined and without consideration for probable changes in the environment or situation. For instance, when a chain has been developed to process a user question, it will always execute the very same sequence no matter whether the information is simple or complex and it cannot adapt if certain data happens to be different from what was anticipated.

Agents behave differently because, in real-time, they would look at the situation with the guidance of a language model and based upon that decide what and how to perform. An agent executes an action, then observes the result of that action and dynamically decides what it is going to do next until the desired result is achieved. That can make the agent much more flexible and responsive. For example, an agent that processes question queries could decide to check a knowledge base, then analyze the user's sentiment, and finally escalate the issue to a human representative based on evolving context about the interaction if needed. Wrapped up within an agent is the capability to utilize a language model for the dynamic determination of action sequences. Where traditional chains simply hardwire the order, an agent uses the language model as a sort of reasoning engine for determining not only what actions to perform but also in what order. Thus, agents have this capability for runtime adaptation, making it a powerful tool in building flexible systems that handle a wide variety of situations, in our case an onboarding buddy.

Our LLM orchestration implies a chain of agents:

- **Contextualization Agent** - Allows the user interaction to be conversational, taking into account past interactions with the system.
- **Onboarding Agent** - Allows the user to ask questions about a software project and provides tailored answers based on the project and the user's needs.
- **Step Processor** - A Chain of Thought processor [27] (CoT) that uses multiple Onboarding Agents in parallel to obtain a detailed answer.

- **Message Enhancer Agent** - Allows the system to provide a proper markdown formatted answer and fix certain inaccuracies in the answer by checking the answer against the codebase.

When a user asks a question, we use a chain of agents to get the desired answer concerning the target code base and documentation. As seen in figure 4.1, when a user asks a question, the question first encounters the 'Contextualization Agent' that puts the question into context based on past chat interactions. For example, if the user refers to past information in the chat, the agent will refer to that too in subsequent answers. After the question is put in the context, the user encounters the first layer of the 'Onboarding Agent'. This first layer analyzes the contextualized question and develops a step-by-step chain of thought (CoT) plan of action that is further used in the 'Step Processor'. The 'Step Processor' itself is just an abstraction of 'Onboarding Agents' that runs multiple 'Onboarding Agents' in parallel - this can be seen as an execution tree that starts with one root node that can span m children of type 'Onboarding Agent' and 'Step Processor' respectively. For each step in the planning new 'Step Processors' can be initiated, until the sub-steps are sufficiently granular (the step does no longer contain possible substeps to take) or if a hard-coded max depth of 'Step Processors' has been reached. When the 'Step Processor' execution ends, the root 'Step Processor' collects the data and passes it to the 'Message Enhancer Agent'. The 'Message Enhancer Agent' groups the answers from the root 'Step Processor' hierarchically in a post-order traversal of the tree concerning the steps and their answer and generates a markdown document based on the answers. This markdown document is now inputted into a 'Message Enhancer Agent' prompt that has the sole purpose of beautifying the markdown generated document and double-checking the formatting correctness but also fixing inaccuracies that the initial answer may have.

Now that we have defined the chain of agents and processors our system uses, we will elaborate on the internal workings of each agent.

4.1. Contextualization Agent

The inner workings of this agent are quite simple, based on the chat ID and a history depth, it fetches the past messages in the chat and formats them into a chat history entity. After the data was fetched from the database, the question was inputted into an LLM with the following system prompt:

```
Given the chat history and the latest user question, your task is to formulate
a context question that incorporates all relevant context from the chat history
and the user question. Do NOT answer the question.
```

Instructions:

1. `If the latest question references specific details (like command names, variables, or explanations) from the chat history, ensure to include those specific details and answer snippets in the context question.`
2. `Make the context question clear and fully understandable on its own by including the message snippets the user refers to, preserving the specific context and meaning from the chat history.`
3. `If the chat history is not relevant, return the question as it was formulated by the user.`
4. `DO NOT generalize the context; focus on preserving the exact meaning and details from the previous messages.`
5. `DO NOT reformulate the relevant context. Provide it as it was given in the past messages.`
6. The contextualization format should be:
```
"For the context of the question: {context here, giving exact snippets from
the past answers and context, do not reformulate, keep the context as it
was from the chat history}
Answer the question: {question}"
```

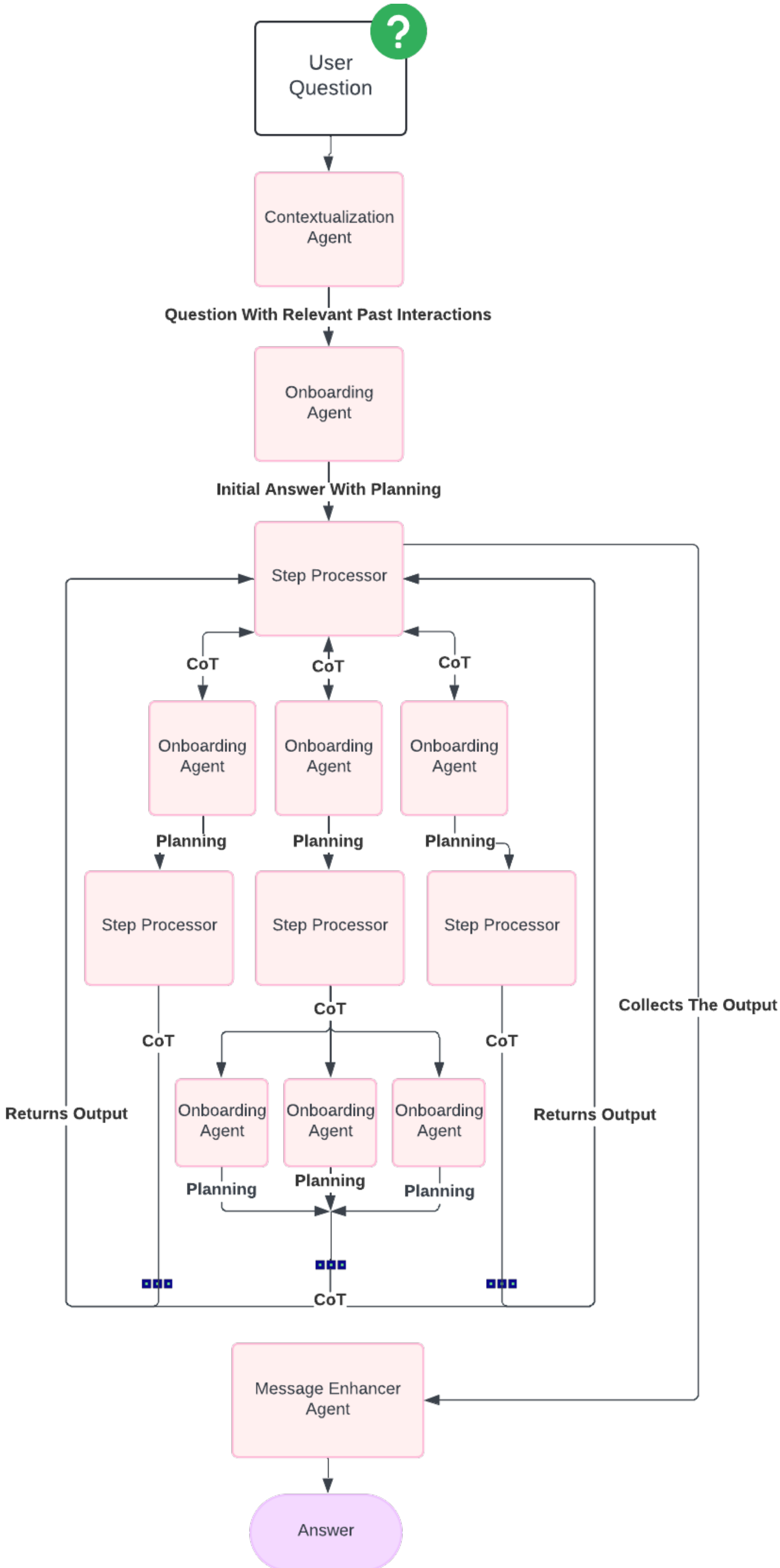


Figure 4.1: Onboarding Buddy Overall Architecture

An example of how this works in practice can be seen as a trace in our LangSmith project[1]. Such a contextualized question is then used in an 'Onboarding Agent'.

4.2. Onboarding Agent

This agent has a more complicated structure than the contextualization agent. The 'Onboarding Agent', as seen in figure 4.2 is composed of three sub-components, namely the memory storage, the planning scratchpad and the retrieval tools.

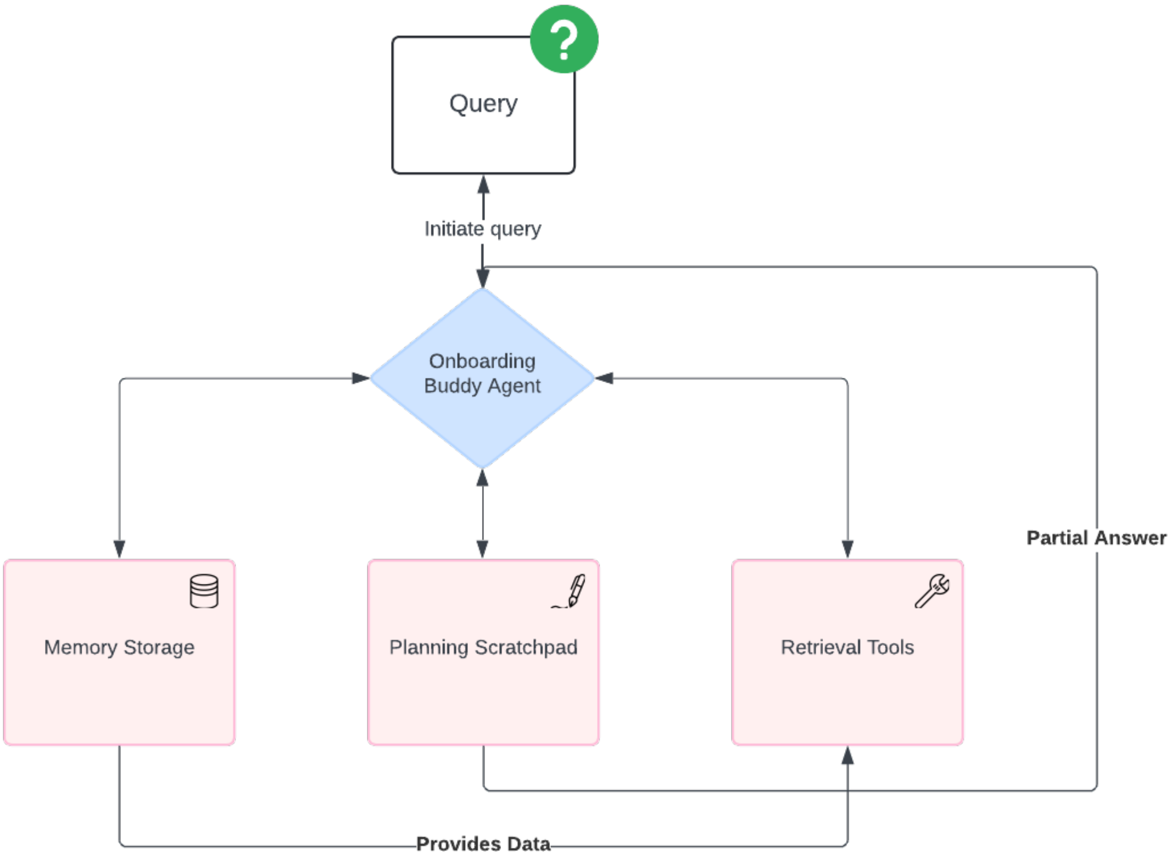


Figure 4.2: Onboarding Agent Architecture

This agent is an iterative type of agent that decides dynamically what actions to take to reach an answer using the planning scratchpad and the retrieval tools. For this, we have implemented a custom chain-of-thought approach (CoT) that will be used in the 'Step Processor'.

The approach we developed was inspired by Google's paper that introduced the concept of chain of thought prompting [27] where a problem is divided into intermediate reasoning steps, allowing the LLMs to solve more intricate tasks that require multi-step logic. Instead of getting an immediate answer, the CoT technique forces the model to "reason" into human-like steps. On top of that Amazon Research [28] researched the same idea from a different angle, and developed an automated chain of thought generation technique where the manual prompting abilities are minimised. In our approach we used the same ideas, but in an agent-centric approach, where the chain of thought can use external tools such as data retrieval from the codebase. For this purpose, each sub-component has its role in the CoT.

[1] Contextualization Agent Trace - `https://smith.langchain.com/public/4c7f9715-096b-46cd-a556-862e6efcce20/r`

4.2.1. Memory Storage

The memory storage is composed of two sub-components, a database and a blob storage. The database records the user and LLM past interactions and the blob storage provides information about the context of the question, in our case details about a software project and its documentation. This blob storage allows the retrieval tools to retrieve relevant information based on the planning scratchpad.

4.2.2. Retrieval Tools

The retrieval tools are represented by the agent being able to directly access the files of a software project inside the blob storage, but also the ability to do a semantic search based on a query against the contents of a software project. Using these functions, we were able to construct an advanced retrieval augmented generation (RAG) system. The planning scratchpad, presented in section 4.2.3, formulates queries for our retrieval tools and our retrieval tools provide feedback and results to the model. This way, the agent can decide to change the query, search for something else, or look more into certain issues.

The semantic search is using a FAISS[2], an in-memory dense-vector database that provides efficient similarity search. Each file, including the documentation, is chunked into documents. Each chunked document is represented by data chunks of 2000 characters with a 200-character overlap using a character splitter specialized in Java Language (more details on this choice will be presented in Chapter 5). Then each chunk is encoded as a vector using an embedding model, in our case OpenAI's text-embedding-3-large[3] due to its effectiveness in semantic understanding, especially for code understanding. For the aforementioned choices, we have conducted a mini-study on available open-source and closed-source embedding models where OpenAI's model performed the best as seen in Appendix B.

Each chunked document contains as metadata the GitHub URL that locates the source under the main branch and the path in the project along with a unique numerical identifier per file.

Based on this data, we make available two tools to the agent, that are "cooperating" to compose an advanced RAG approach, and have the following definitions:

- **retrieve_missing_files** - retrieve entire files of source code, the agent can decide to use it only based on the file name information, or if it is guided to do so by the 'retrieve_relevant_code_snippets' tool based on the location available in the metadata.
- **retrieve_relevant_code_snippets** - retrieves chunks of data based on similarity search using a query to the vector database.

At each iteration, the agent can decide to use any of these two tools, but we define the **retrieve_missing_files** as a more computationally expensive function since it can potentially retrieve blindly files from the storage. The function is defined as:

- **Input:** `Should be a list of string paths that represent the needed or required files to get a correct answer if the file is not already specified in the message. The path is formatted based on the provided project structure. The list can have at most 5 files, ordered based on file importance or probability of being important. The most important paths are first on the list.`
- **Definition:** `Retrieve the full contents of the files in a repository. Only answer if the file is not already specified in the message. The number of files retrieved should not be high, since more information about the files can be gathered in the future. Include only the most important files, and if required, files that are needed. This function (retrieve_missing_files) is more expensive than retrieve_relevant_code_snippets function so retrieve_relevant_code_snippets is preferred over this function. This function should be used only when the whole file returned by retrieve_relevant_code_snippets is needed for a correct response or when retrieve_relevant_code_snippets doesn't return an answer after some try.`

[2]Meta Faiss - https://ai.meta.com/tools/faiss/
[3]OpenAI embeddings - https://platform.openai.com/docs/guides/embeddings

- **Output:** A series of plain text full file contents wrapped in a region comment if successful:

```
//region {file}
//FILE CONTENT
//endregion {file}
```

Or a fallback message if the tool is failing:

```
//region {file} - File not found
//endregion {file}
```

This allows this function to be ”guided” by the output of the **retrieve_relevant_code_snippets** that is defined as:

- **Input:** `Query to look up in retriever`
- **Definition:** `Retrieve relevant project (code) snippets based on a query. This function is preferable over the retrieve_missing_files function and should be used over retrieve_missing_files. It takes as input a query. The search query should take into account the context of the project. IMPORTANT: If the query for this tool was already used before and it didn't provide a valid answer, do not try to call this tool again with the same parameter.`
- **Output:** A series of plain text full file contents wrapped in a region comment:

```
//region code snippet from file: {metadata['path']}
//DOCUMENT CONTENT
//endregion code snippet from file: {metadata['path']}
```

Or two possible fallback messages if the tool is failing. For the eventuality that the exact query did not retrieve any documents in past interactions, avoiding loops and ”writing” the choice in the planning scratchpad:

```
{
"status": "failure",
"message": f"The RAG query for the 'retrieve\_relevant\_code\_snippets' tool:
    '{query}' didn't succeed in finding the relevant documents. Try again with
     another query modify the current query or use your knowledge if the query
     cannot be rephrased. DO NOT USE THE SAME QUERY IN SUBSEQUENT CALLS. If
    this issue persists, you can use your knowledge and don't call the
    semantic search. You can use the retrieve\_missing\_files instead."
}
```

Or for the eventuality that the retriever does not retrieve any document, for a query that was not used in the past, it returns:

```
{
"status": "failure",
"message": f"The RAG query for the 'retrieve\_relevant\_code\_snippets' tool :
     '{query}' didn't succeed in finding the relevant documents, try again
    with another query or modify the current query or use your knowledge if
    the query cannot be rephrased. DO NOT USE THE SAME QUERY IN SUBSEQUENT
    CALLS."
}
```

Important to mention, is the fact that the relevant code snippets retriever returns the top five relevant documents above a similarity threshold of 0.1, this allows us to guide the query towards the right direction. The similarity threshold has values between 0 and 1, where 0 represents no similarity and 1 is an exact match. The threshold value of 0.1 allows us to exclude some extremely irrelevant documents concerning the query but also allows us to obtain 0 search results. A query with no results then falls under the fallback guiding prompts that are used in the agent planning to reformulate the query.

This in combination with the agent planning scratchpad presented in subsection 4.2.3 allows the agent to self-adjust the queries and files retrieved, storing this information into the agent planning scratchpad.

One such example is a trace[4] that first queries the vector database for a query about the 'QuestionnaireRepository.java', then it identifies the files that are used in the code present in the documents retrieved by the semantic search, and then fully retrieves only the needed ones using the missing files retrieval.

4.2.3. Planning Scratchpad

The agent planning scratchpad serves as a dynamic mechanism of the internal memory applied by the onboarding agent to track its reasoning process while interacting with different tools, queries and environments. It works just like a journal or log, where it writes down intermediate steps and thoughts. This way, the agent can maintain better control and much more transparently with complex and multiple-step tasks.

4.2.4. Tracking intermediate results

When the agent intends to solve any problem, it may have to invoke several tools presented in the subsection 4.2.2 or different actions sequentially (e.g. retrieve an answer from its knowledge). Of course, each subsequent action might produce new data or insights, and these would be stored temporarily in the agent planning scratchpad. For instance, if the agent is trying to find some information in the knowledge base (repository or documentation), gather information from the vector store, or reason out something over various facts, then at each step it writes it in the scratchpad so that it may refer to it later.

4.2.5. Record Reasoning Steps

The agent, after every action taken, writes its internal thought process to this scratchpad. This allows the agent to recall why it had chosen to perform some particular action or call some specific tool, making its decision process much more transparent and accurate. Some notes and memories help when the agent has to explain its reasoning or when some user wants to understand how it came up with a particular solution.

The task chaining can be efficiently done by utilizing the scratchpad. Each action informs the next step, as the agent avoids redundancy or repetition of steps. For example, an agent could run a tool and get results; these are stored in the scratchpad for easy reference next time it needs them, instead of having to rerun the tool. This also allows the guiding process of the retrieval tools, being able to guide the search in the correct direction.

If that agent receives new information or even corrects its path, the scratchpad aids the agent without displacing earlier context. It is a flexible buffer that clings to the decisions of the past but can change when the situation begins to evolve (e.g. after a series of unsuccessful searches in the codebase, it manages to find the required documentation or code snippet).

4.2.6. Debugging Tool

The planning scratchpad is handy during debugging and optimizing agents' workflows. With the contents of the scratchpad, we can identify where an agent might have gone wrong in either its reasoning or where an unnecessary call of a tool was made so, a means to improve efficiency.

Examples of multi-step tasks in an agent needing to answer a user's query, using more than one tool call would be something such as (hypothetical example used for simplicity):

User Query: "What's the weather in Delft, The Netherlands? Also, suggest a good restaurant for dinner?"

Step 1: First, the agent would call a tool giving the data about the weather in Delft.
Result: "It's 15°C and rainy as always in Delft."
Scratchpad Entry: "Weather retrieved for location Delft: 15°C and rainy as always in Delft."

Step 2: The agent suggests a restaurant. It takes the retrieved weather as input.
Result: "Indoor Dutch restaurant in Delft."
Scratchpad Entry: "Recommended restaurant: indoor Dutch restaurant in Delft."

[4] Example Guided Retrieval - `https://smith.langchain.com/public/b3b99a9b-5078-49be-85fe-88b3a85f4d0e/r`

Step 3: Taking as input the above recommendations and the menu for the recommended restaurant, the agent uses a food recommendation tool.
Result: "Suggesting bitterballen, frikandel and erwtensoep."
Scratchpad Entry: "Recommended food: bitterballen, frikandel, erwtensoep."

In the process, the ending scratchpad already contains the intermediate results, enabling the agent to assemble a final response that fully answers the user query while considering the reasoning for each step. A final answer might look as: "Since today it's rainy and cold in Delft, I would recommend you to go to a Dutch restaurant and try their warm traditional food such as bitterballen, frikandel and erwtensoep (might look nasty, but it's very tasty)."

4.2.7. Onboarding Agent Wrapping Up

Now that we have presented all the sub-components for the 'Onboarding Agent', and have defined how the planning scratchpad interacts with the memory storage and the retrieval tools, we have to establish how their orchestration works.

As seen in figure 4.2, the agent works iteratively through the planning scratchpad. We use a maximum iteration value of 15, as recommended by Langchain's agent default values, so our agent can take at most 15 steps to return an answer. Usually, based on our experiments, an agent takes at most 10 steps to return a stopping decision and assemble a complete answer. For each step, all the sub-components can add information to the planning scratchpad.

Each agent run is governed by the following system prompt:

```
You are a highly skilled software engineering mentor (Onboarding Buddy) tasked
with assisting new employees in onboarding software engineering projects written
in {PROGRAMMING_LANGUAGE}. Your role is to guide users through the codebase, provide
helpful
information, and assist with any questions or issues they may have. You should
act like a senior software engineer mentoring a new team member.

You are integrated inside IntelliJ IDEA (JetBrains IDE), and users ask you
questions from within the IDE. Provide detailed answers on how to perform specific
actions in the IDE, but keep the information practical and concise. When possible,
avoid overwhelming the user with unnecessary details.

The project you are helping with is a {PROJECT_DESCRIPTION}.

Your key priorities:
1. Understand the user's question and determine whether it can be answered directly
without external tools. Only use search tools if the question specifically requires
information from the codebase.
2. Provide practical examples where applicable, but do not share the full project
structure directly with the user. You may reference specific files or paths but
do not give the entire project structure in a single message.
3. Aim to resolve the user's question quickly and efficiently. Once you have
enough information to fully answer the question, conclude your response. Do not
continue searching if the question has been answered sufficiently.
4. If a question requires you to explore the codebase further, use the tools
available to retrieve the missing information. However, do not continue querying
tools once the necessary information has been found.
5. Clearly signal the completion of the task when you believe the user's query
has been fully answered.

This is the Java project structure the user will ask questions about:
{PROJECT_STRUCTURE_PLACEHOLDER}

Remember:
```

```
- The repository about the project in question can be found at:
{GITHUB_REPO_LINK_PLACEHOLDER}.
- The user needs to access the code in his branch called:
{GITHUB_USER_BRANCH_NAME_PLACEHOLDER}.
- Always be supportive and informative, but avoid overcomplicating the responses.
- You can answer based on your own knowledge when the tools are not necessary
 or useful.
```

And output system prompt that enables another chain of thought that we are going to discuss in the 'Step Processor' section:

```
You will output the result in a JSON format with one MANDATORY field, 'ANSWER'.
ANSWER will represent the answer, optionally may have an OPTIONAL field called
'STEPS' if there are any steps to follow by the users, represented by an array,
empty if there is background info that is still required. Everything should be
formatted in JSON format. Both Answer and STEPS have the contents only as strings
(possibly markdown strings) but ONLY strings. Include the contents of the files
you loaded using the tools.
```
[5]

This formatting rule is once more enforced in the User Prompts. If the formatting is not respected in the agent's answer, the error is added in the planning scratchpad to be solved, ensuring the proper formatting of the answer.

By orchestrating the user question with the contextualization agent, and the system prompts presented above, we can generate another chain of thought (planning) with the proper format as seen in figure 4.3.

```python
from typing import List, Optional
from pydantic import BaseModel

class Answer(BaseModel):
    answer: str
    steps: Optional[List[str]] = None

# Example answer generated by the agent when asked how to set up a Java Spring app
example = Answer(
    answer="Set up a Java Spring application",
    steps=[
        "1. Install Java Development Kit (JDK): Download and install the latest version of
            the JDK (at least JDK 8).",
        "2. Set up your Integrated Development Environment (IDE): Install an IDE like
            IntelliJ IDEA or Eclipse.",
        "3. Install Maven or Gradle: Ensure that you have a build automation tool such as
            Maven or Gradle installed.",
        "4. Create a new Spring Boot project: You can use Spring Initializr (https://start.
            spring.io/) to generate a new Spring Boot project.",
        "5. Import the project into your IDE: Once the Spring Boot project is downloaded,
            import it into your IDE.",
        "6. Configure the application: Open the `application.properties` or `application.yml`
             file to configure your Spring Boot app.",
        "7. Create your first Spring Controller: Define a basic REST controller to handle
            HTTP requests.",
        "8. Run the application: Use the IDE to run the Spring Boot application, or use the `
            $./gradle bootRun` command.",
        "9. Test the application: Access the REST API endpoint using Postman or a browser to
            ensure it's working.",
        "10. Package the application: Build the project into a JAR or WAR file using Maven or
             Gradle."
    ]
)
```

Figure 4.3: Example answer of the 'Onboarding Agent' generating a chain of thought planning to achieve a task.

[5] At the moment the system was implemented, structured output was not available on Open AI's models - `https://platform.openai.com/docs/guides/structured-outputs`

Such an answer, as seen in figure 4.3, is then processed by the following component in the answer processing chain presented in figure 4.1, namely the 'Step processor".

4.3. Step Processor

The 'Step Processor' is just a recursive chain of Onboarding Agents that run in parallel. As seen in the overall architecture figure 4.1 and the structure of the answers provided by the 'Onboarding Agent', figure 4.3, the 'Step Processor' executes new 'Onboarding Agents' for each step provided in the answer. If the newly instantiated agents provide steps to follow, new 'Step Processor' instances are instantiated until there are no more steps to follow, or a maximum threshold established in the configuration is reached. An example run of the 'Step Processor' can be seen on our tracing project [6] where the user asks where he can find existing endpoints in the project in the context of duplicating a database entry. In this case, the 'Step Processor' runs on the answer as seen in figure 4.4.

```
from typing import List, Optional
from pydantic import BaseModel

class Answer(BaseModel):
    answer: str
    steps: Optional[List[str]] = None

# Example answer generated by the agent when asked to duplicate a questionnaire
example = Answer(
    answer="To implement the `duplicateQuestionnaire` method, you need to duplicate the questionnaire identified by `idQuestionnaire`, including all its questions and metadata.",
    steps=[
        "1. Retrieve the original questionnaire using `QuestionnaireRepository.findById(idQuestionnaire)`.",
        "2. Check if the questionnaire exists. If not, return an appropriate error response.",
        "3. Create a new `Questionnaire` instance and copy the metadata from the original questionnaire (e.g., title, category, visibility).",
        "4. For each question in the original questionnaire, create a new `Question` instance, copy the question details, and add it to the new questionnaire's question list.",
        "5. Save the new questionnaire using `QuestionnaireRepository.save(newQuestionnaire)`.",
        "6. Return a `ResponseEntity` with the duplicated questionnaire details or a success message."
    ]
)
```

Figure 4.4: Example answer of the Onboarding Agent generating steps for duplicating a questionnaire that is used in the Step Processor.

Each step processing runs independently, in parallel among the other Onboarding Agents to provide a final answer that is at the end collected by the "root" Step Processor. After this, the 'Message Enhancer Agent' collects the answer from the root processor.

4.4. Message Enhancer Agent

When the root 'Step Processor' finishes running, the 'Message Enhancer Agent' collect the data hierarchically in a post-order traversal of the tree for the steps and their answer and generates a markdown document based on the answers.

The initial markdown document is then processed using an 'Onboarding Agent' with the following prompt, on top of the other tools, functionalities and prompts mentioned in section 4.2:

```
Adjust, Improve, Enhance, Repair, and Remove Redundancy from the following markdown
document. Follow these guidelines:
```

[6] Step Processor Example - `https://smith.langchain.com/public/4005e192-2ecd-4713-ac6b-9d17e75bbbc4/r`

```
• Adjust: Modify the structure and layout for clarity, readability, and proper
  formatting according to markdown standards.
• Improve: Refine the content by correcting punctuation or style issues to
  improve flow and coherence.
• Enhance: Add or refine details, headers, lists, or other markdown elements
  where necessary to enrich the document and make it more informative or aesthetically
  pleasing.
• Repair: Fix any broken links, references, or formatting errors.
• Remove Redundancy: Identify and eliminate repetitive content, ensuring that
  the document is concise and clear without losing important information.

 If additional context or information is required to perform these actions, feel
free to retrieve it from available sources using the tools provided to you. Output
only the enhanced markdown document without extra explanations or commentary
e.g. without statements like 'Here is the result' or 'Here's the enhanced markdown
document'...etc.

This is the document:
{MARKDOWN_DOCUMENT}
```

The output of this agent is then ready to be displayed to the user as a properly formatted markdown document. This ends the execution chain of our Onboarding Buddy solution from an LLM orchestration perspective.

5 The Onboarding Buddy Study

The Onboarding Buddy study centres around users onboarding on an unknown/unseen software project discussed in subsection 5.1. The initial study aimed for a comparative study. Our platform was specifically designed to support such comparative analysis through its group-based architecture, allowing for both AI-assisted and traditional onboarding approaches. However, due to time constraints and the scope of this thesis, we were unable to include a control group in our current evaluation. Therefore, in this chapter, we will present our exploratory study design and findings from participants who used our onboarding solution elaborated in the architecture chapter 3 and approach chapter 4.

5.1. Choice of software project

For our study, we needed to choose a software project on which the users will perform onboarding tasks. The selection of codebase is a proprietary Java Spring project that was independently developed and maintained for freelance engagements in the past by this thesis' author. This decision took into account several factors combined in line with the objective of our onboarding solution. First, the project is fully owned and copyrighted by the author, with the assurance of full retention of their intellectual property rights to ensure that the LLMs are not "contaminated" with our codebase.

The private code base ensured that the code was kept non-exposed to existing LLMs throughout all its life cycles and during their training processes. Moreover, this ensured precautionary measures to get rid of any bias or direct memorization by the LLMs themselves. Thus, our study set-up tries to preserve the integrity and validity of the results.

The project in question is a backend application managing theoretical driving lessons oriented at the 'Romanian Theoretical Driving Exam'. This includes but is not limited to, theoretical test management, road legislation materials, questionnaire management, and user authentication. Such broadness of the application provides a perfect environment for testing the capabilities of our onboarding solution in handling real-world software projects.

The code base comprises about 10,000 lines of code and is supported by 65 integration tests. This size and level of complexity allow users to be presented with something substantial they can work with to perform a wide range of onboarding tasks that realistically can be used to model actual instances of in-practice software development. Interaction with this project provides the user with a realistic onboarding process, making it much easier for our solution to achieve its objective and efficiently integrate into new software environments.

In this line of work, participants in our study have to be asked to perform certain onboarding tasks in light of the performance and usability assessment of our LLM-based onboarding solution. These should include how well the system facilitated understanding the codebase, navigating its structure and performing changes within the application.

The project is made available to interested researchers and practitioners upon registration for the study in our platform[1]. This open access allows readers to thoroughly investigate and further validate our experimental set-up. This way, our experiments will not be directly exposed to external influences that may affect the reliability and significance of the result.

5.2. Choice of Onboarding Tasks

With the choice of codebase in mind, as presented in section 5.1, we analysed the possibilities of onboarding tasks. For this, we created a list of 12 onboarding tasks that the participant users can perform on the codebase as seen in appendix C. From this list, we chose only two tasks, namely task C.1 and task C.11 plus a project set-up task that will require the users to set up the development environment for the codebase on their computers. Each task was then wrapped in an onboarding story presented in the subsections 5.2.1, 5.2.2 and 5.2.3.

5.2.1. Set-up Task

This task requires the user to set up their project on their local machines and it is formulated as:

```
Jane, the project manager, approached you with a warm smile.

"Welcome to the team! I'm excited to have you on board," she said. She handed
you a document and continued, "Before diving into the main tasks, we need to
ensure you have the project set up on your local machine."

She handed you a sticky note with a link written on it:
https://github.com/andrei5090/OnboardingBuddyExperiment.

"This is the repository you'll need to get started," Jane explained. "All the
instructions for setting it up should be somewhere there. Make sure everything
is running smoothly before moving on to the other tasks."

Jane paused for a moment, looking thoughtful.

"It's essential to have the project environment ready. Once you have it set up,
let me know. Then, we can go through the specific tasks you'll be working on.
Good luck, and don't hesitate to reach out if you have any questions!"

With that, Jane left you to get started. The journey had begun, and the first
step was to set up the software project from the provided repository.

For this task you are required to setup the Github Project
OnboardingBuddyExperiment.

Please note, email errors and/or PDF resource errors are expected to appear when
running the project in development, you are not required to set-up (cloud) external
dependencies (e.g. S3 storage, Email..etc.).
```

At first glance, this task is very simple and from a technical point of view, to set up the project for a development environment, the user has just to uncomment 3 lines of code in the configuration file and the project is ready to run. What makes it a bit tricky is how to verify that the set-up was actually correct. For this, we have left some hints in the README file in the repository that require the user to register into the backend by performing a registration request and then perform some endpoint actions. We have left more ”breadcrumbs” into the files that define the user registration endpoints so it’s easier for the users to test their set-up. One such example can be seen in figure 5.1.

[1] Onboarding Buddy Registration Page - `https://onboarding.software/register?colab=true`

```java
/**
 * Local login Example (POST request): localhost:8080/auth/signup
 * With Body: {"email":"test@email.com","password":"TestPassword1!"}
 * @param loginRequest login request
 * @return JWT Bearer Token to be used in future requests
 */
@PostMapping("/login")
public ResponseEntity<?> authenticateUser(@Valid @RequestBody LoginRequest loginRequest)
    {
    try {
        return authServices.loginUser(loginRequest);
    } catch (Exception e) {
        logger.info("Something went wrong with the auth/login endpoint: {} for user {}",
            e, loginRequest.getEmail());
        throw e;
    }
}

/**
 * Local Signup Example (POST request): localhost:8080/auth/signup
 * With Body: {"name":"test","email":"test@email.com","password":"TestPassword1!"}
 * @param signUpRequest signup request
 * @return status
 */
@PostMapping("/signup")
public ResponseEntity<?> registerUser(@Valid @RequestBody SignUpRequest signUpRequest) {
    try {
        return authServices.registerUser(signUpRequest);
    } catch (Exception e) {
        logger.info("Something went wrong with the auth/signup endpoint: {} for user {}",
            e, signUpRequest.getEmail());
        throw e;
    }
}
```

Figure 5.1: Example of a breadcrumb hint left to the participating users inside the repository.

This task allows us to understand what steps the user take to understand how the project is running and how open-ended tasks are perceived by the user. For example, in the set-up stage, we left some console warnings masked as errors (non-breaking runtime warnings - they do not stop the execution of the software). We mention during the task elicitation that these errors are expected, but we want to see if the users fully understand the requirements of the task and whether they expect and understand the ”expected” errors.

5.2.2. New Payment Option Task

The second task introduces the user to the codebase by requesting them to add a new payment option into the system:

```
After some time of setting up the project, you notified Jane that everything
was ready. She seemed pleased with your progress.

"Great job on setting up the project," Jane said with an encouraging smile.

"Now that you have the environment ready, let's move on to your next task."

Jane handed you another document with detailed requirements:

"We need to add a new payment option to our application. Specifically, we want
to allow users to purchase subscriptions for 180 days at a cost of 25 Lei.
Your task is to integrate this new payment option and update the application
accordingly."

She pointed out a few key areas in the codebase where you might need to make
```

```
changes.
"Make sure the new payment option is seamlessly integrated and that the subscription
duration and cost are correctly implemented."

Jane gave you a reassuring nod. "If you run into any issues or have questions,
don't hesitate to ask. We're here to help you succeed."

With the new task in hand, you set out to add the new payment option and update
the application, ready to make a significant contribution to the project.
```

For this task you are NOT required to add tests.

The system allows the users to buy in-app subscriptions, and the participant is required to add a new payment option based on a subscription duration and price.

As seen in the task definition available in appendix C.1, the user has to add a new enum value option to finish this task, but this is very tricky without knowing the codebase. To reach this option file, the user has to study the entry point (controller), and the services that define it, and understand that the payment encoding is represented and signed cryptographically based on an enum that contains the payment options. The solution of this task is a one-liner but to understand that, the user needs first to understand the overall logic of the payment pipeline. This task allows us to understand user's code comprehension and their ability to understand complex code, but without the hassle of actually write a complex functionality.

5.2.3. Questionnaire Duplication Task

The last task is more demanding in terms of how much code the participants have to write, but the task itself is easier to understand, but more demanding to implement:

```
After successfully integrating the new payment option, you reported back to Jane.
She was impressed with your efficiency and thoroughness.

"Fantastic work on the payment integration," Jane said, clearly satisfied.
"I have another important task for you."

She handed you a document with a new set of instructions:
Add functionality to duplicate an existing questionnaire.
```

Tasks:

1. **Add a new endpoint POST /questionnaires/duplicate/{id} to duplicate a questionnaire. (ADMIN-ONLY)**
2. **Ensure that your code handles the duplication logic, including questions and metadata.**
3. **Create integration tests to ensure the duplication works correctly.**

```
Jane explained, "This functionality is crucial for our users who need to reuse
and adapt existing questionnaires quickly.
Make sure the duplication process is smooth and retains all the necessary details,
including the questions and metadata."

She gave you an encouraging look. "As always, if you need any help or run into
issues, don't hesitate to ask. You're doing great, and we appreciate your hard
work."
```

```
With the new task outlined, you felt ready to tackle the challenge, eager to
enhance the application's capabilities and contribute further to the project's
success.

For this task you are required to add the duplication API and integration tests.
```

As seen in the task definition available in appendix C.11, the user has to create a new endpoint, duplicate an existing questionnaire into a service and test the implementation writing integration tests into an already existing test suite. This task allows us to analyze what steps the users take, if their code-style understanding is aligned with the project and whether a much more complex task is achievable in a more complex setting with or without the Onboarding Buddy.

5.3. Task Feedback

In this chapter, we provide the outline of the feedback questions that the participants were asked when interacting with the system during their assigned tasks. The feedback is used to evaluate several aspects of the users' interaction experience with the onboarding assistant in terms of experience and ease of onboarding, usefulness, accuracy, and satisfaction. The participants were asked to assess their experience by responding to a set of questions that made use of both quantitative and qualitative methods.

Quantitative feedback was gathered using a 5-point Likert [18] scale, where questions were asked about various aspects of user interaction with the Onboarding Buddy. Qualitative feedback was elicited through open-ended questions, allowing participants to express whatever came to mind, and thus provide suggestions and any further comments related to the experience.

Alongside the subjective feedback, we also collected detailed interaction data to analyze how users engaged with their tasks. This includes information such as files opened, created, edited, or deleted. Furthermore, we collect the user activity within their IDE, including instances where they switched to another application during the task (the IDE was unfocused). Moreover, we tracked how users completed the tasks by analyzing the code they wrote or edited by making sure they committed and pushed their changes.

Tasks Feedback Questions for the Onboarding Buddy Group

- **Overall Experience:**
  - *Question:* What was the overall experience with Onboarding Buddy for the current task?
  - *Likert scale:* 0 (Very Bad) to 4 (Excellent)
  - *Data:* Measures general satisfaction with the onboarding process.
- **Ease of Onboarding:**
  - *Question:* How was the ease of onboarding with Onboarding Buddy for the current task?
  - *Likert scale:* 0 (Very Difficult) to 4 (Very Easy)
  - *Data:* Captures user perception of how easy or difficult the onboarding process was.
- **Ease of Use:**
  - *Question:* How easy was it to use Onboarding Buddy for the current task?
  - *Likert scale:* 0 (Very Hard) to 4 (Very Easy)
  - *Data:* Evaluates how user-friendly the onboarding tool was.
- **Helpfulness:**
  - *Question:* How helpful was Onboarding Buddy for the current task?
  - *Likert scale:* 0 (Not Helpful) to 4 (Very Helpful)
  - *Data:* Assesses how much the tool-assisted in completing the task.
- **Frequency of Use:**

- *Question:* How frequently did you use Onboarding Buddy for the current task?
- *Likert scale:* 0 (Never) to 4 (Always)
- *Data:* Tracks the frequency of engagement with the tool.

- **Accuracy:**
  - *Question:* How accurate was the information provided by Onboarding Buddy for the current task?
  - *Likert scale:* 0 (Very Inaccurate) to 4 (Very Accurate)
  - *Data:* Measures how reliable or correct the provided information was.
- **Understanding:**
  - *Question:* How well did you understand the information provided by Onboarding Buddy for the current task?
  - *Likert scale:* 0 (Very Poor) to 4 (Excellent)
  - *Data:* Determines how easily users comprehended the information.
- **Suggestions:**
  - *Question:* What suggestions do you have for improving Onboarding Buddy for future tasks?
  - *Data:* Gathers qualitative feedback for tool improvement.
- **Additional Comments:**
  - *Question:* Do you have any additional comments about Onboarding Buddy for the current task?
  - *Data:* Collects any extra feedback or insights from users.

Tasks Feedback Questions for the Control Group - Not included in the results

- **Tool Effectiveness:**
  - *Question:* How effective was the traditional onboarding method for the current task? (exploration method, you decide how to perform the task)
  - *Likert scale:* 0 (Very Ineffective) to 4 (Very Effective)
  - *Data:* Measures the perceived effectiveness of the traditional onboarding method for the task.
- **Problem Solving Steps:**
  - *Question:* What steps did you take to solve problems for the current task?
  - *Data:* Gathers qualitative information on the problem-solving strategies and approaches taken by participants.
- **Tools Used:**
  - *Question:* What tools did you use to finish the task? (e.g. other Assistants, Google Search, Stack Overflow)
  - *Data:* Collects information on external tools and resources used during task completion.
- **Unresolved Issues:**
  - *Question:* What issues remained unresolved for the current task, and why? Please mention any insights in your problem-solving process.
  - *Data:* Gathers details on unresolved problems, obstacles encountered, and reasoning behind them.

Both groups send analytics periodically to the backend through the IDE plugin, representing the files that are created, opened, and edited, and when the IDE is focused or unfocused, with timestamps for each activity.

6 Results

In this section, we would like to present the results of our evaluation of the Onboarding Buddy system. The results are organized to highlight the impact on critical aspects of analysis: user engagement, task performance, user feedback, and factors influencing the coding experience and familiarity with development tools. This section emphasizes overarching trends and patterns, grounded in data gathered throughout the study.

6.1. User Engagement and Activity Patterns

Participants exhibited varying levels of engagement with the Onboarding Buddy system. On average, users performed approximately 368 activities during the study. An activity is defined as unfocusing the IDE, focusing the IDE, editing, creating or deleting files. Figure 6.1 illustrates the distribution of the number of activities per user, indicating that while some users interacted extensively with the system, others had more moderate engagement.

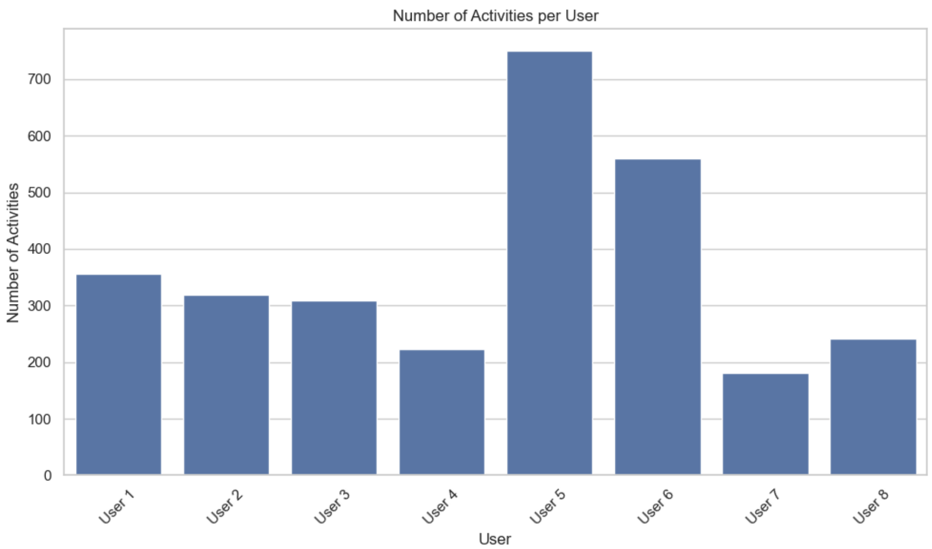


Figure 6.1: Number of activities per user

The temporal distribution of activities indicated that the peak times were mid-morning and late afternoon. Activity counts surged at approximately 10:00 and 16:00 hours, suggesting during these hours the users were quite active; this might be indicative of typical workday rhythms and/or optimal times for onboarding activities.

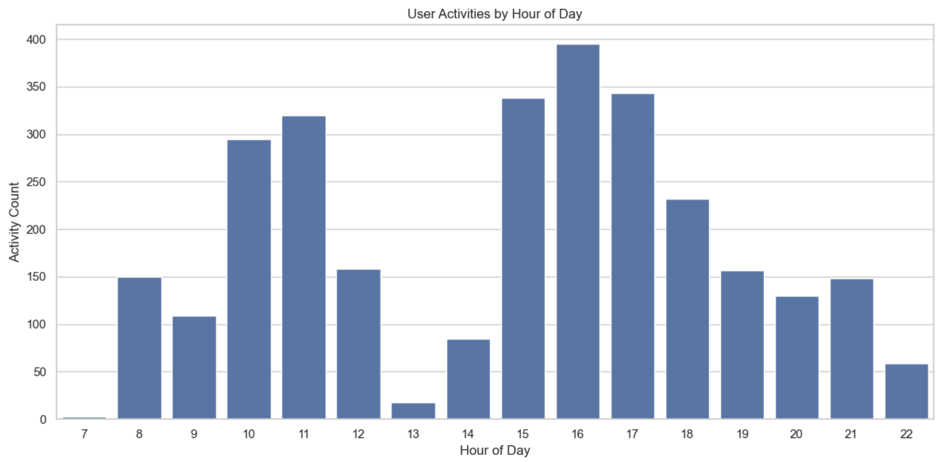


Figure 6.2: Distribution of user activities by hour of the day

6.2. Time Allocation in the Development Environment

Participants predominantly spent their time within the Integrated Development Environment (IDE), averaging 175 minutes, compared to 37 minutes spent on external resources. This preference indicates that the Onboarding Buddy effectively kept users engaged within the development environment, potentially streamlining the onboarding process. Figure 6.3 shows the average time spent inside and outside the IDE.

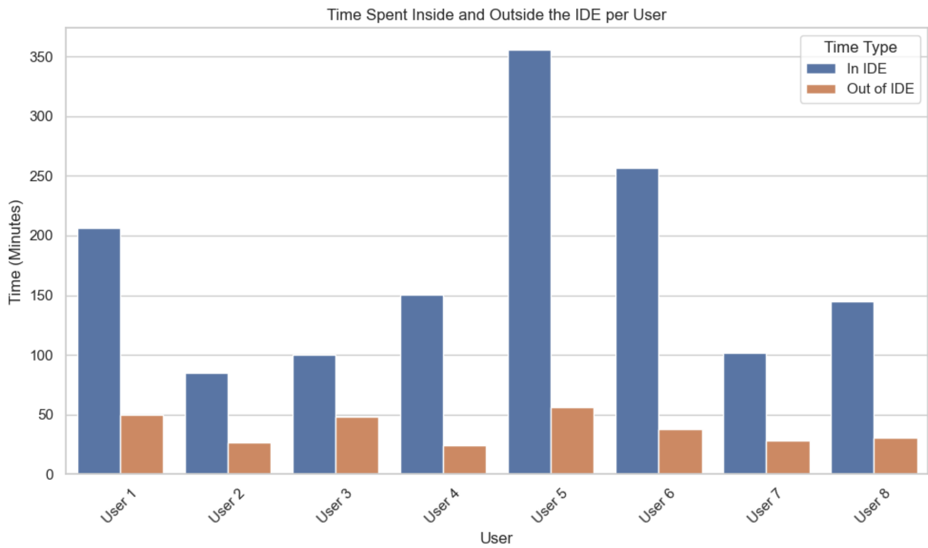


Figure 6.3: Time spent inside and outside the IDE

6.3. Task Performance Analysis

We analyzed the adjusted time taken by participants to complete each task, excluding inactivity periods longer than 10 minutes. The overall average task duration was 50 minutes. Task 2, which involved adding a new payment option, had the shortest average duration of 28 minutes, indicating that participants found it relatively straightforward. In contrast, Task 3, which focused on implementing questionnaire duplication and writing integration tests, had the longest average duration of 67 minutes, suggesting a higher level of complexity. Figure 6.5 and 6.4 presents the average adjusted task duration per task.

Additionally, a manual analysis of the participants' solutions for the onboarding tasks, focusing on those

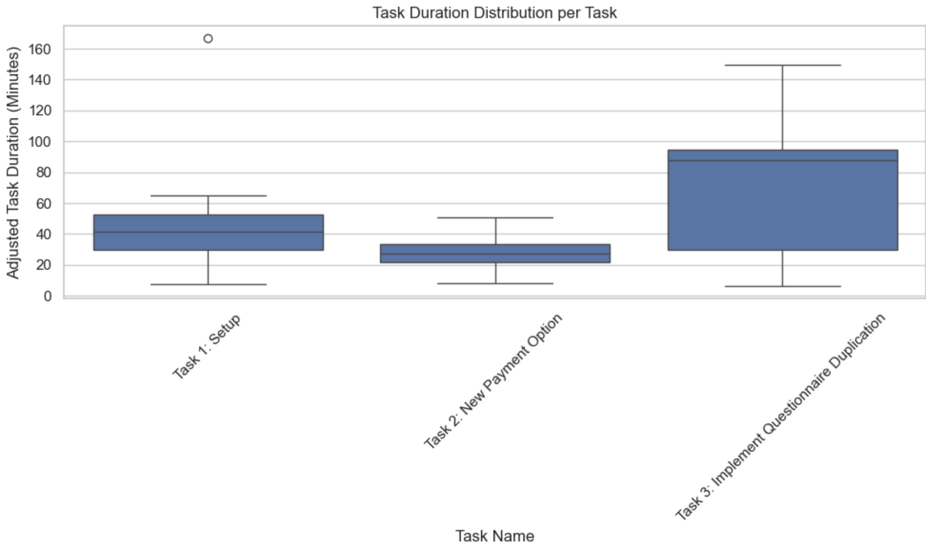


Figure 6.4: Average adjusted task duration per task

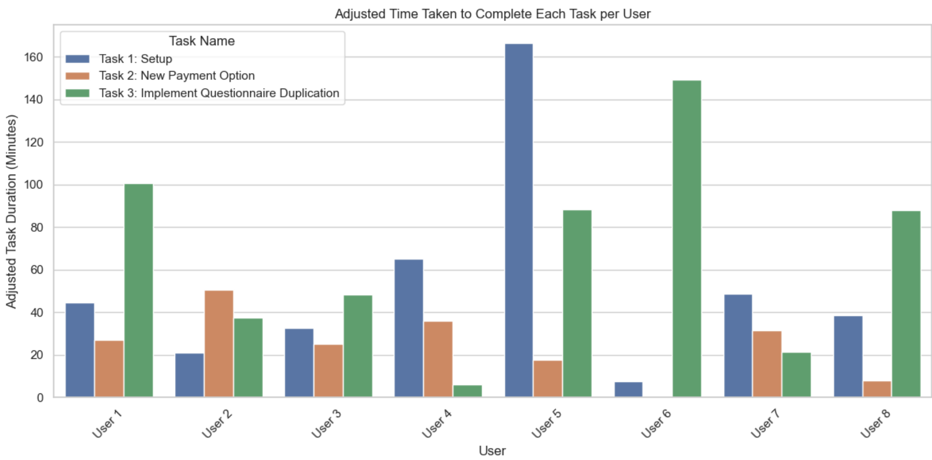


Figure 6.5: Average adjusted task duration per task per user

who marked their submissions as completed, revealed that all but one participant completed all tasks correctly. The one exception made only minor error on a single task. Moreover, 4 participants did not managed to identify that the expected errors in the Task 1 were actually expected (mentioned in the task description), so they conducted extra steps to validate their task correctness. Overall, the completion and accuracy rate was nearly 100%.

6.4. User Feedback and System Helpfulness

Participants provided valuable feedback on their experience with the Onboarding Buddy system. The overall helpfulness rating was 3 out of 4, indicating a high level of satisfaction. Task 2 received the highest helpfulness rating, aligning with its shorter completion time and suggesting that users found the system particularly effective for this task. Figure 6.6 shows the average helpfulness rating per task.

Sentiment analysis of the qualitative feedback revealed a slightly positive overall sentiment, with an average polarity score of 0.10 (slightly positive) according to theTextBlob library [20]. Common themes in the feedback included the usefulness of the Onboarding Buddy in facilitating task completion and areas where the system could offer more explicit guidance or context awareness.

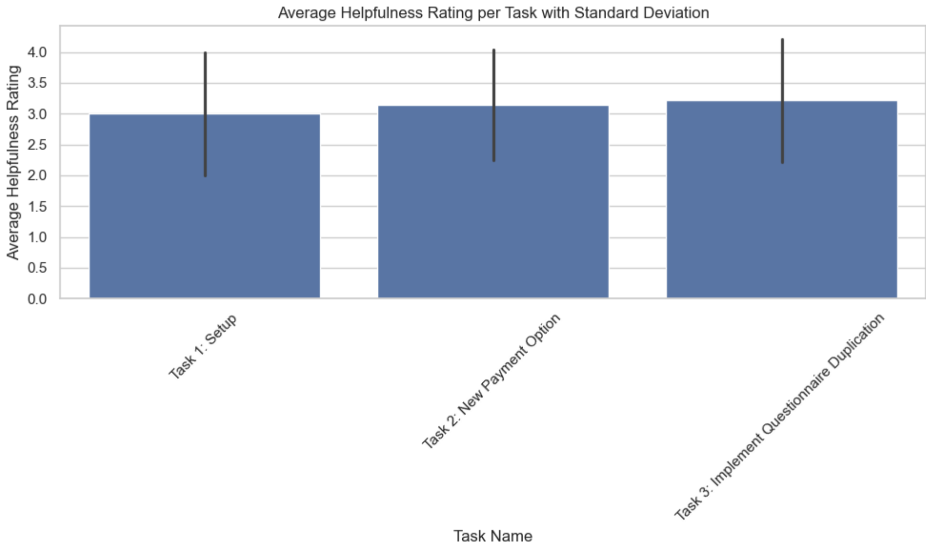


Figure 6.6: Average helpfulness rating per task

6.5. Impact of Coding Experience and Tool Familiarity

We examined how participants' backgrounds influenced their onboarding experience, focusing on coding experience and familiarity with the IntelliJ IDE.

6.5.1. Coding Experience

Contrary to expectations, participants with more coding experience had longer average task durations. Users with 6–10 years of experience took an average of 65 minutes per task, compared to 40 minutes for those with less than 1 year of experience, as illustrated in Figure 6.7. This may be due to experienced users delving deeper into the tasks or exploring the codebase more thoroughly.

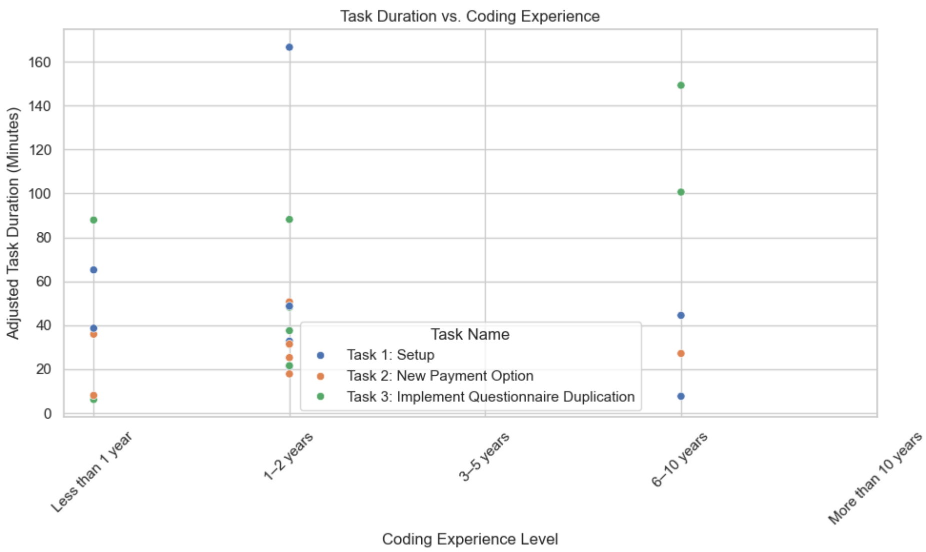


Figure 6.7: Average task duration by coding experience

6.5.2. Familiarity with IntelliJ IDE

Participants familiar with IntelliJ (4 participants) had longer average task duration (64.35 minutes) compared to those unfamiliar (4 participants) with it (37.75 minutes) as seen in figure 6.8. Additionally, IntelliJ users reported slightly lower helpfulness ratings. This suggests that users new to the IDE may have relied more on the Onboarding Buddy for assistance, while experienced users possibly expected more advanced functionalities.

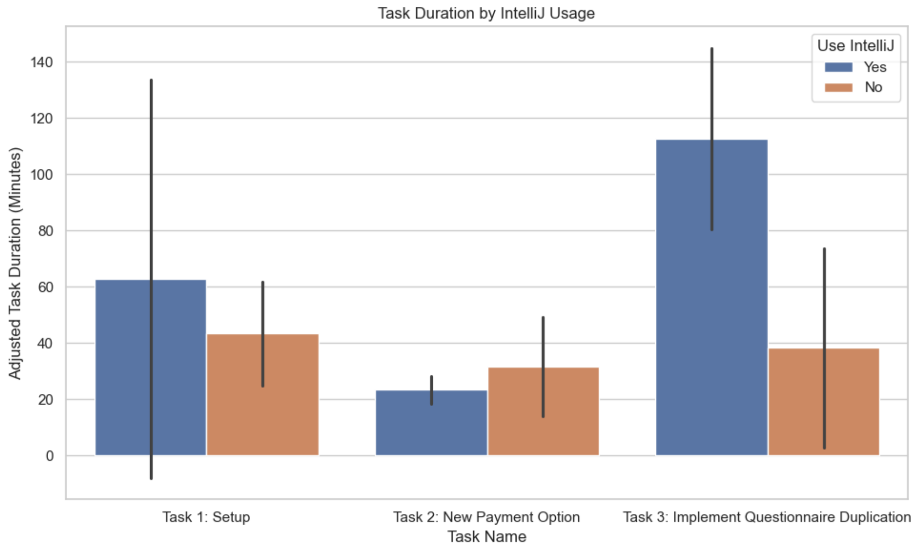


Figure 6.8: Task Duration by IntelliJ Usage - the black lines represent the standard deviation

6.6. Correlation Between System Usage and Perceived Helpfulness

A strong positive correlation (0.70) was observed between the frequency of Onboarding Buddy usage and the helpfulness ratings, as depicted in Figure 6.10. This indicates that increased engagement with the system led to higher perceived helpfulness, highlighting the importance of encouraging regular interaction with onboarding tools to enhance the onboarding experience.

6.7. Perspectives on Accuracy, Understanding, and Ease of Onboarding

Assessing the accuracy, clarity, and overall ease of onboarding provided by the Onboarding Buddy is crucial for evaluating its effectiveness. Participants were asked to rate these aspects during their tasks, and the results offer valuable insights into the user experience.

The average accuracy rating was 3.15 out of 4, and the average understanding rating was 3.35 out of 4, indicating a generally positive perception but also highlighting areas for enhancement. Similarly, the ease of onboarding, as seen in figure 6.9, received an average score of 3 out of 4, with a standard deviation of 0.78. This suggests that most participants found the onboarding process relatively smooth but experienced varying levels of ease.

Notably, Task 2 received the highest ratings in all three categories, which aligns with its shorter completion time and higher helpfulness rating. This pattern suggests that the Onboarding Buddy was particularly effective in facilitating tasks of moderate complexity.

The standard deviation of 0.78 in the ease of onboarding scores indicates some variability in participants' experiences. Factors contributing to this variation may include individual differences in coding experience, familiarity with the development environment, and personal learning preferences.

6.8. Activity Types and File Interaction

We analyzed the distribution of activity types to understand how participants interacted with the development environment. The most frequent activities were the focusing and unfocusing of the IDE (moving to other applications), indicating active engagement within the IDE but also outside of it during the com-

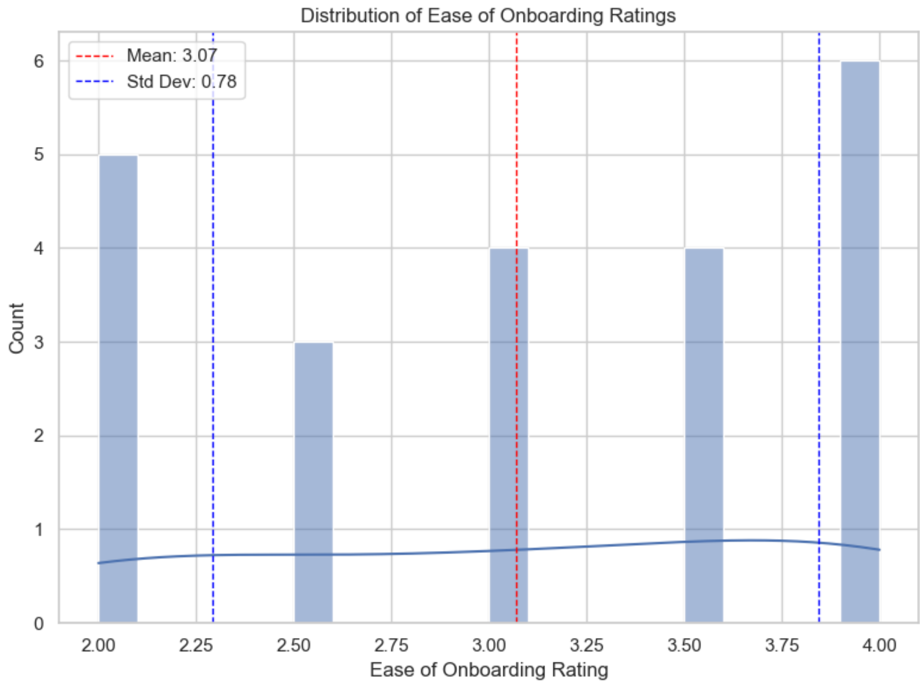


Figure 6.9: Self-Perceived Ease Of Onboarding

pletion of the tasks. This is further confirmed by the number of breaks the participants took during the study. Figure 6.11 illustrates the distribution of different activity types recorded during the study.

Participants most frequently opened configuration files such as `application.yaml`, suggesting a focus on setup and configuration tasks. The most edited files were related to integration tests and service implementations, reflecting the nature of the tasks assigned.

6.9. Breaks and Productivity

The number of breaks taken during tasks provided insights into user engagement and potential friction points as seen in figure 6.12. On average, participants took 1.35 breaks per task. Task 3 had a notably higher number of breaks, totalling 20 across all users, which could reflect its complexity or areas where the Onboarding Buddy could offer improved support.

6.10. Interest in Onboarding Solutions

All participants expressed a high level of interest in onboarding solutions, with an average interest score of 7.75 out of 10. This demonstrates a strong demand for tools like the Onboarding Buddy in the software development industry.

6.11. Common Challenges

A word frequency analysis of participant feedback revealed common themes. Participants frequently mentioned the need for additional help, more explicit information, and improved context awareness. Suggestions mostly focused on technical scalability in terms of question-answer time.

Participants' qualitative feedback shed further light on their experiences. While many users appreciated the guidance provided, some noted that the Onboarding Buddy could improve in offering more detailed explanations, especially for complex tasks like Task 3, which involved duplicating questionnaires and handling nested data structures. This occasionally led to confusion and required participants to invest additional effort to overcome these challenges.

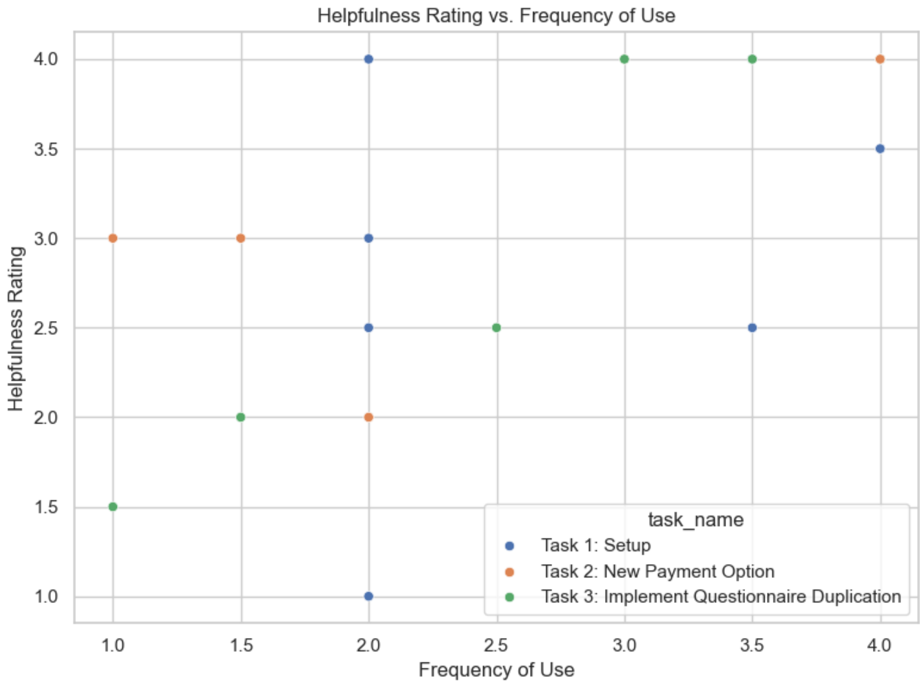


Figure 6.10: Correlation between frequency of use and helpfulness rating

6.12. Research Questions

This section details the key research questions guiding our research into the effectiveness of the *Onboarding Buddy* tool in facilitating software onboarding for developers. We want to explore how influences in using the tool relate to efficiency and completion of tasks, how perceived usefulness relates to the performance of the developers, if user satisfaction is positively related to user engagement levels and task completion rates, and lastly, the overall feasibility of the implementation of the *Onboarding Buddy* tool as a large language model-based solution. We hope that answers to these questions will provide a complete picture of how this tool contributes to a better onboarding experience.

6.12.1. RQ1: What patterns of efficiency and task completion emerge when developers use Onboarding Buddy for software onboarding tasks, as measured by time spent in the IDE, number of file operations, and task completion rates?

The Onboarding Buddy influences onboarding efficiency and the completion of tasks measured with self-assessed metrics and the data collected. An average of 175 minutes was spent inside the IDE, showing high engagement with the development environment via the Onboarding Buddy. Average activities per user were at about 368, describing an active interaction with the codebase concerning file edits, creations, and openings. Their performance yields almost 100% accuracy, which indicates that all the required tasks were performed correctly. The most considerable average time taken toward the completion of these tasks was 50 minutes. Among these, Task 2, adding a new payment option, was the one resolved in under 30 minutes being at 28 minutes on average. Thus, it may be inferred that Onboarding Buddy does support faster performance of tasks, especially for medium-complexity tasks.

Although we cannot draw any comparative data from the participants for other onboarding practices due to the limitation of the study, from a self-assessment metric viewpoint, it seems that the users felt the Onboarding Buddy was effective in enhancing their onboarding efficiency. Moreover, large periods of IDE engagement and successful completion of tasks support the conclusion that the Onboarding Buddy positively impacts onboarding efficiency and task completion rates.

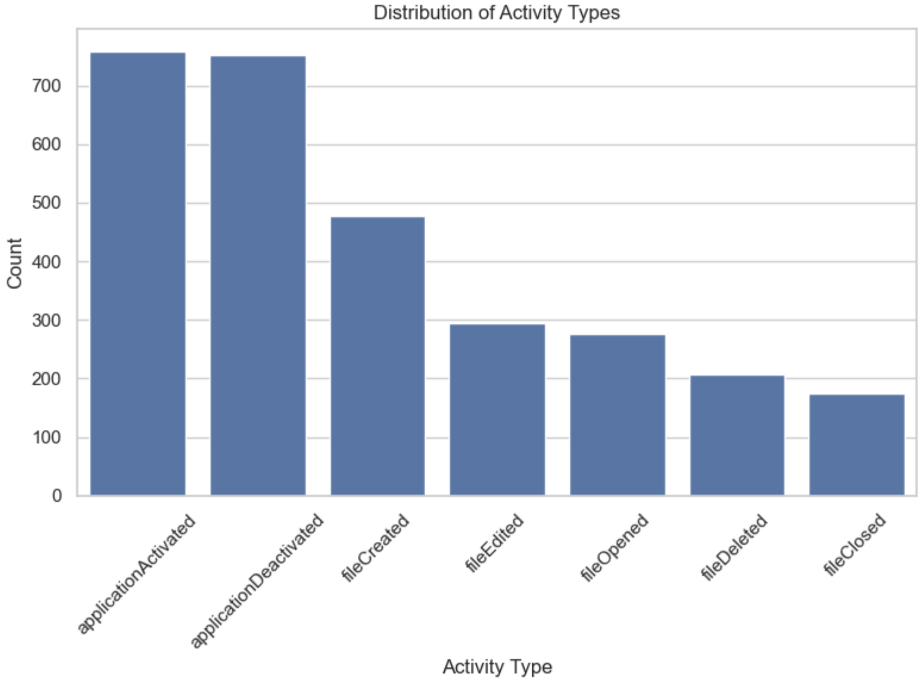


Figure 6.11: Distribution of activity types

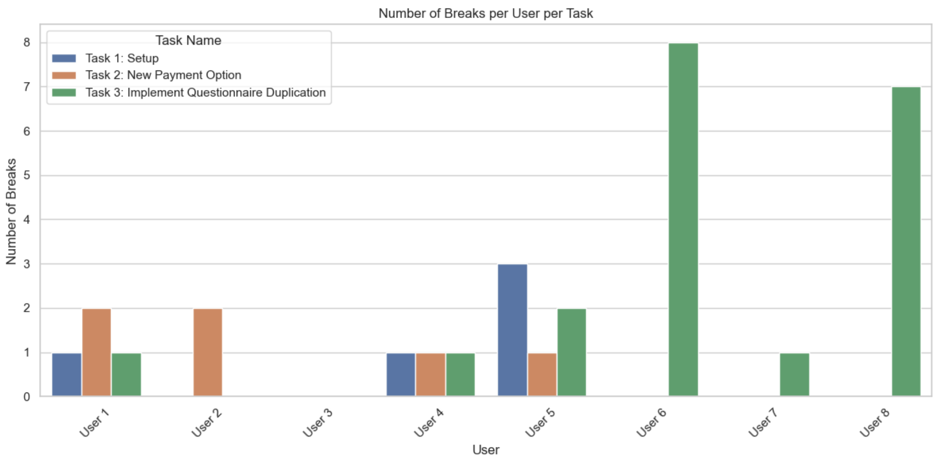


Figure 6.12: Number of breaks per user

6.12.2. RQ2: What relationships exist between developers' perceived usefulness of Onboarding Buddy and their task completion patterns?

The perceived usefulness of the Onboarding Buddy significantly influences the successful completion and seamless integration of onboarding tasks. Participants rated the helpfulness of the Onboarding Buddy at an average of 3 out of 4, indicating a generally positive perception. There is a strong positive correlation (0.70) between the frequency of using the Onboarding Buddy and the helpfulness ratings, suggesting that increased engagement with the assistant enhances its perceived usefulness.

Tasks that received higher helpfulness ratings, such as Task 2, also had shorter completion times and higher self-assessed ease of onboarding scores. For instance, Task 2 not only had the highest helpfulness rating but was also completed in the shortest average time. This indicates that when participants perceived the Onboarding Buddy as more useful, they were able to complete tasks more efficiently and integrate them seamlessly into their workflow.

Therefore, the self-assessed metrics confirm that the perceived usefulness of the Onboarding Buddy directly contributes to successful task completion and smooth integration of onboarding tasks.

6.12.3. RQ3: What is the relationship between user satisfaction with Onboarding Buddy and their engagement levels in effectively completing onboarding tasks?

There is a positive relationship between user satisfaction with the Onboarding Buddy and their engagement levels in effectively completing onboarding tasks. Participants reported high satisfaction levels, with self-assessed overall experience ratings averaging above 3 out of 4. Those who reported higher satisfaction also tended to use the Onboarding Buddy more frequently, as evidenced by the strong correlation between frequency of use and helpfulness ratings.

Higher user satisfaction is associated with increased engagement, leading to more effective task completion based on the correctness of participants' solutions. Participants who found the Onboarding Buddy helpful and easy to use spent more time within the IDE and completed tasks more efficiently. The self-assessed metrics for accuracy (average 3.15 out of 4), understanding (3.35 out of 4), and ease of onboarding (3 out of 4) further support this relationship, indicating that satisfied users are more engaged and effective in completing onboarding tasks.

6.12.4. RQ4: How feasible is the implementation of Onboarding Buddy as an LLM-based assistant for onboarding tasks?

Based on empirical analysis, the implementation of the Onboarding Buddy as an LLM-based development assistant demonstrates significant efficacy in facilitating onboarding processes. The quantitative metrics reveal compelling evidence of its practical utility, particularly in terms of task completion rates and user engagement metrics. Our research indicates that participants maintained substantial engagement within the integrated development environment. The empirical data demonstrates consistently high performance across key metrics:

- Task completion accuracy reaching close to 100%.
- User satisfaction metrics averaged above 3.0 on a 4-point scale.
- Moreover, user interest in this type of solution reached a mean score of 7.75/10.

While the actual implementation does provide some limits regarding the contextual awareness of instructions, these are normal challenges in LLM-based systems; such limitations have not affected its primary functionality. Most of the areas for improvement found involve enhancements toward allowing the system to yield even more detailed instructions. Telemetry data gathered and user feedback revealed that Onboarding Buddy serves well in codebase familiarization, navigation tasks and implementation tasks. This has a great potential to be widely used for onboarding LLM-assisted solutions into professional software development.

7 Conclusion

This thesis presented a novel solution for software engineering onboarding through the development of the Onboarding Buddy system, which utilizes large language models (LLMs) and retrieval augmented generation (RAG) enhanced by an automated chain-of-thought approach. The system's architecture was implemented across three distinct layers: an experimental backend handling LLM orchestration and data processing, a chat UI for user interaction, and an IDE plugin for IDE integration and telemetry collection.

The technical implementation included an agent-centric design, that handles automated chain-of-thought onboarding plans of actions. Among these agents, there is the contextualization agent, which keeps coherent chat history; an onboarding agent that generates step-by-step plans in a guided manner; the step processor that executes multiple onboarding plans in parallel and the message enhancer agent, which refines the responses of the aforementioned agents.

Experimental evaluation evidences practical effectiveness in several key areas: users average 175 minutes of active IDE time versus 37 minutes on external resources, showing engagement with the development environment. The completion times ranged according to the task's complexity: medium complexity tasks took about 28 minutes, whereas the more complex implementations took up to 67 minutes. The completion rate was also quite high; nearly all participants managed to implement the required functionality. More interestingly, analyses of usage patterns showed that the experienced developers used it for longer periods on average than newer developers, presumably reflecting deeper engagement with the code rather than merely completing tasks quickly. The overall rating of the system's helpfulness was 3 out of 4, with a correlation coefficient of 0.70 between usage frequency and perceived helpfulness.

While effective in supporting the completion of tasks and reducing stress related to onboarding, feedback also revealed areas of improvement. These include enhanced context awareness, more explicit instructions, and refined user experience in the user interface. This system provides a smoother onboarding process and increases developer productivity, although some limitations concerning handling complex tasks remain to be addressed. The implementation of the Onboarding Buddy shows that LLM-based solutions are feasible for software engineering onboarding and provides a starting point for follow-on research into how automated developer support and software team knowledge sharing can be improved.

8 Future Work

While the Onboarding Buddy system demonstrates promising results in facilitating software engineering onboarding, several paths for future research and development remain to be explored. A primary direction for future work involves extending our current experimental setup into a controlled comparative study. Our platform was specifically designed to support such comparative analysis through its group-based architecture, allowing for both AI-assisted and traditional onboarding approaches. However, due to time constraints and the scope of this thesis, we were unable to include a control group in our current evaluation.

Future research could address the fundamental question: "How do usability and perceived usefulness of Onboarding Buddy compare to traditional onboarding methods in influencing user efficiency, task completion, and workflow integration?". This study would utilize our existing platform's capability to divide participants into two groups with comparable experience levels, one using Onboarding Buddy and the other following traditional documentation and mentoring approaches. These would give insight into the relative effectiveness of AI-assisted onboarding versus conventional methods by tracking quantitative metrics, such as completion times, error rates, and assistance requests. Qualitative insights could be derived from this through structured interviews.

Furthermore, recent advances in large language models point to exciting opportunities that enable further improvements in the capabilities of such a system: Claude 3.5 Sonnet[1] and OpenAI's GPT o1 models [2] with similar CoT approaches. These models present superior performance for code understanding and generation, therefore being promising candidates that could improve the accuracy and relevance of system responses. Moreover, future efforts should be spent on studying how such advanced models could be integrated into onboarding systems within an agent-centric architecture, resulting in more sophisticated reasoning chains and better contextualised assistance. Moreover, better performance of such models on longer contexts would allow for richer project understanding and more accurate retrieval-augmented generation.

[1] Claude 3.5 Sonnet - https://www.anthropic.com/news/claude-3-5-sonnet
[2] OpenAI's GPT o1 models - https://openai.com/o1/

References

[1] Gemini Team et al. *Gemini: A Family of Highly Capable Multimodal Models*. 2023. arXiv: 2312.11805 [cs.CL].

[2] Ricardo Britto et al. "Evaluating and strategizing the onboarding of software developers in large-scale globally distributed projects". In: *Journal of Systems and Software* 169 (2020), p. 110699. ISSN: 0164-1212. DOI: https://doi.org/10.1016/j.jss.2020.110699. URL: https://www.sciencedirect.com/science/article/pii/S0164121220301473.

[3] Ricardo Britto et al. "Onboarding software developers and teams in three globally distributed legacy projects: A multicase study". In: *Journal of Software: Evolution and Process* 30.4 (Nov. 2017).

[4] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL].

[5] Jim Buchan, Stephen MacDonell, and Jennifer Yang. *Effective team onboarding in Agile software development: techniques and goals*. 2019. arXiv: 1907.10206 [cs.SE]. URL: https://arxiv.org/abs/1907.10206.

[6] Kyle G.S. Daigle. *Octoverse: The State of Open Source and rise of AI in 2023*. The GitHub Blog. 2024. URL: https://github.blog/2023-11-08-the-state-of-open-source-and-ai (visited on 04/04/2024).

[7] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL].

[8] Matthijs Douze et al. *The Faiss library*. 2024. arXiv: 2401.08281 [cs.LG]. URL: https://arxiv.org/abs/2401.08281.

[9] Neil A. Ernst and Martin P. Robillard. "A study of documentation for software architecture". In: *Empirical Software Engineering* 28.122 (Sept. 2023). Accepted 25 May 2023, Published 15 September 2023, pp. 1–32. DOI: 10.1007/s10664-023-10347-2. URL: https://doi.org/10.1007/s10664-023-10347-2.

[10] Yunfan Gao et al. *Retrieval-Augmented Generation for Large Language Models: A Survey*. 2024. arXiv: 2312.10997 [cs.CL]. URL: https://arxiv.org/abs/2312.10997.

[11] Geert Heyman and Tom Van Cutsem. *Neural Code Search Revisited: Enhancing Code Snippet Retrieval through Natural Language Intent*. 2020. arXiv: 2008.12193 [cs.IR].

[12] Hamel Husain et al. *CodeSearchNet Challenge: Evaluating the State of Semantic Code Search*. 2020. arXiv: 1909.09436 [cs.LG].

[13] Karen Sparck Jones. "Natural Language Processing: A Historical Review". In: *Current Issues in Computational Linguistics: In Honour of Don Walker*. Ed. by Antonio Zampolli, Nicoletta Calzolari, and Martha Palmer. Dordrecht: Springer Netherlands, 1994, pp. 3–16. ISBN: 978-0-585-35958-8. DOI: 10.1007/978-0-585-35958-8_1. URL: https://doi.org/10.1007/978-0-585-35958-8_1.

[14] An Ju et al. *A Case Study of Onboarding in Software Teams: Tasks and Strategies*. 2021. arXiv: 2103.05055 [cs.SE].

[15] Takeshi Kojima et al. "Large Language Models are Zero-Shot Reasoners". In: *Advances in Neural Information Processing Systems*. Ed. by S. Koyejo et al. Vol. 35. Curran Associates, Inc., 2022, pp. 22199–22213. URL: https://proceedings.neurips.cc/paper_files/paper/2022/file/8bb0d291acd4acf06ef112099c16f326-Paper-Conference.pdf.

[16] Patrick Lewis et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2021. arXiv: 2005.11401 [cs.CL]. URL: https://arxiv.org/abs/2005.11401.

[17] Evan Li. *GitHub Ranking*. Accessed: 2024-09-11. 2024. URL: https://github.com/EvanLi/Github-Ranking.

[18] Rensis Likert. *A Technique for the Measurement of Attitudes*. Archives of Psychology 140. New York: The Science Press, 1932, pp. 1–55.

[19] Chenxiao Liu and Xiaojun Wan. *CodeQA: A Question Answering Dataset for Source Code Comprehension*. 2021. arXiv: 2109.08365 [cs.CL].

[20] Steven Loria. *TextBlob: Simplified Text Processing*. https://textblob.readthedocs.io/en/dev/. Accessed: 2024-10-15. 2018.

[21] Stephen MacNeil et al. "Experiences from Using Code Explanations Generated by Large Language Models in a Web Software Development E-Book". In: *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. ACM. Mar. 2023.

[22] Humza Naveed et al. *A Comprehensive Overview of Large Language Models*. 2023. arXiv: 2307.06435 [cs.CL].

[23] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL].

[24] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. 2023. arXiv: 2307.09288 [cs.CL].

[25] Hugo Touvron et al. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL].

[26] S. Umbrello. "AI Winter". In: *Encyclopedia of Artificial Intelligence: The Past, Present, and Future of AI*. ABC-CLIO, 2021, pp. 7–8.

[27] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903.

[28] Zhuosheng Zhang et al. *Automatic Chain of Thought Prompting in Large Language Models*. 2022. arXiv: 2210.03493 [cs.CL]. URL: https://arxiv.org/abs/2210.03493.

[29] Wayne Xin Zhao et al. *A Survey of Large Language Models*. 2023. arXiv: 2303.18223 [cs.CL].

A

Experiment Manual

Each user was provided with the following experiment manual[1] to guide their process into the experiment as seen below:

Onboarding Buddy

Experimental Manual

JETBRAINS

TU Delft

What is this Onboarding Buddy Experiment

In collaboration with TU Delft, JetBrains Software Engineering Research Department is conducting a study to enhance onboarding solutions for software developers.

Welcome to the Onboarding Buddy experiment. By participating, you will gain access to Onboarding Buddy, an innovative tool designed to offer natural language explanations and guidance directly within your development environment. This tool is specifically tailored to assist in onboarding new developers to software projects.

What Onboarding Buddy Offers:

- **Step-by-Step Explanations:** Receive detailed, step-by-step guidance about software projects based on your specific questions.
- **Focus on Understanding:** Emphasizes understanding the 'HOW' rather than just the 'DO', providing a deeper comprehension of the development process.
- **ChatGPT-like Environment:** Interact with the tool in an environment similar to ChatGPT, making it user-friendly and intuitive.

What does the experiment consist of?

We are experimenting to compare the onboarding performance of users utilizing our Onboarding Buddy solution versus those who do not. This experiment aims to determine the usefulness and reliability of such tools in real-world scenarios. Specifically, we want to assess whether Onboarding Buddy can make the onboarding process faster and better overall, enhancing the experience for new developers.

Participants in this experiment will simulate the onboarding process of a software engineer joining a new project and possibly a new team. By providing detailed, natural language explanations and step-by-step guidance directly within the development environment, Onboarding Buddy aims to streamline the onboarding process. Participants will clone a repository, which contains the backend application for the driving legislation project, and will be assigned their own branches. They will use Onboarding Buddy to understand and interact with the project's various features, from user authentication to test management and legislative content access.

By comparing the experiences and performance of users with and without Onboarding Buddy, we hope to gather valuable insights into the effectiveness of such tools in improving the onboarding experience for software engineers.

[1] Experiment Manual Source - https://docs.google.com/document/d/1rxHSZzcmyYq-N3TiQh_Twb6Ms8IxakmdPIJvjJPX6rU

The project involves a comprehensive backend application developed using Java Spring, designed to support a driving legislation application. This application addresses the theoretical components necessary for driving education by facilitating theoretical tests and providing resources for learners. The backend system is robust, offering functionalities such as user authentication, management of theoretical questions, creation of custom questionnaires, and administration of theoretical tests. Additionally, it includes a legislation e-book feature, enabling users to delve deeper into driving laws and regulations. This backend is critical for ensuring that users can seamlessly authenticate, access, and interact with the educational content, thereby enhancing their preparation for driving exams.

What do I have to do?

To see how the onboarding process works with our approach, we are using an IntelliJ Plugin called Onboarding Buddy. This plugin is designed to streamline the initial setup and integration for new team members, ensuring a smooth and efficient start. Here's a step-by-step guide to get you started.

Step 1: Install IntelliJ IDEA

The first step is to install IntelliJ IDEA, which is the Integrated Development Environment (IDE) we will be using. If you already have IntelliJ IDEA installed, that's great! If not, you can download it from the JetBrains website. Follow the installation instructions specific to your operating system (Windows, macOS, or Linux).

Step 2: Install the Onboarding Buddy Plugin

Once you have IntelliJ IDEA installed, the next step is to install the Onboarding Buddy plugin. This plugin is available on the IntelliJ Marketplace and can be installed directly through the IDE. Here's how to do it:

1. **Open IntelliJ IDEA**: Launch the IDE on your computer.
2. **Go to the Plugins Section**: Click on `File` in the top menu bar, then select `Settings` (or `Preferences` on macOS). In the left-hand pane, find and select `Plugins`.
3. **Search for Onboarding Buddy**: In the Plugins window, go to the `Marketplace` tab and enter "Onboarding Buddy" in the search bar.
4. **Install the Plugin**: Find the Onboarding Buddy plugin in the search results and click the `Install` button.

5. **Restart IntelliJ IDEA**: After the installation is complete, you will be prompted to restart IntelliJ IDEA. Click `Restart IDE` to apply the changes.

Alternatively, you can download the plugin directly from the JetBrains Plugin Repository and install it manually.

Step 3: Accept the GitHub Invitation

After installing the Onboarding Buddy plugin, ensure that you have accepted the GitHub invitation to the project. This invitation was sent to your email along with this document. If you haven't received the invitation, check your spam folder or contact your project administrator.

To accept the invitation:

1. **Open Your Email**: Check the email account where the invitation was sent.
2. **Find the Invitation**: Look for an email from GitHub with the subject "Invitation to collaborate on [Project Name]".
3. **Accept the Invitation**: Open the email and click the `View invitation` button, then click `Join` on the GitHub page that opens.

Step 4: Enter Your UserID

Upon accepting the GitHub invitation, you should have received a UserID in the same email. This UserID is crucial for configuring the Onboarding Buddy plugin to recognize your setup.

To enter your UserID:

1. **Open IntelliJ IDEA**: Launch the IDE if it is not already open.
2. **Open Onboarding Buddy**: Look for the Onboarding Buddy icon in the IntelliJ toolbar. Click on this icon to open the plugin interface.
3. **Enter UserID**: A configuration window will appear, prompting you to enter your UserID. Input the UserID you received in the email and click `Confirm`

.

Step 5: Complete the Onboarding Process In The Plugin

Once you have entered your UserID, the Onboarding Buddy plugin will guide you through the remaining steps of the onboarding process. Follow these instructions carefully:

1. **Consent Form**:
   - After entering your UserID, a consent form will appear. You must fill out this form with some personal information and provide your consent to participate in the experiment.
   - Submit the completed form to proceed.
2. **Accessing Tasks**:
   - After submitting your consent, you will be presented with three tasks titled: "Setup", "New Payment Option", and "Implement Questionnaire Duplication".
   - Begin with Task 1, "Setup". Or you have the option to talk with the onboarding buddy without any task assigned to it (after you close this custom chat, you will not be able to return to it).
3. **Starting a Task**:
   - For each task, press the `Start` button to initiate the task.
   - Complete the tasks one by one in the specified order: "Setup" first, followed by "New Payment Option", and finally "Implement Questionnaire Duplication".
4. **Using Tools Based on Your Group**:
   - Depending on your assigned group, you will either:
     - **Group A (ChatGPT-like Chat)**: Have access to our solution through a ChatGPT-like chat interface. Use this chat to get assistance and guidance as you work through each task.
     - **Group B (Traditional Onboarding)**: Do not have the chat window available. Instead, you can use any tools or resources you prefer to complete the tasks.

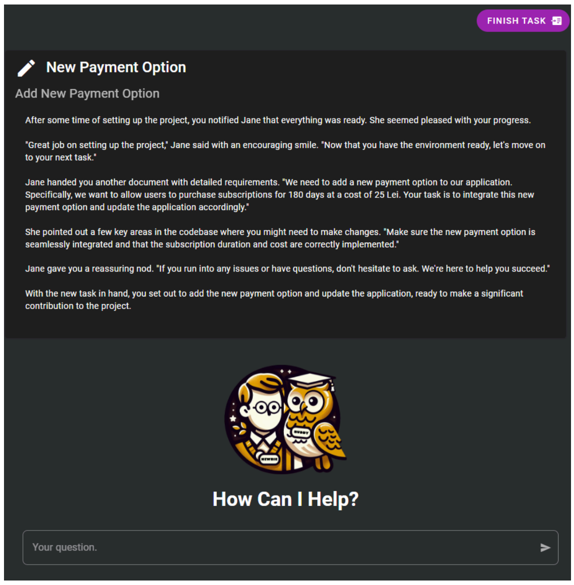
Refer to the next section for more detailed information about the tools and resources available to each group. This structured approach ensures you get the most out of the onboarding process, tailored to your assigned method.

Group A (With Onboarding Buddy Access):

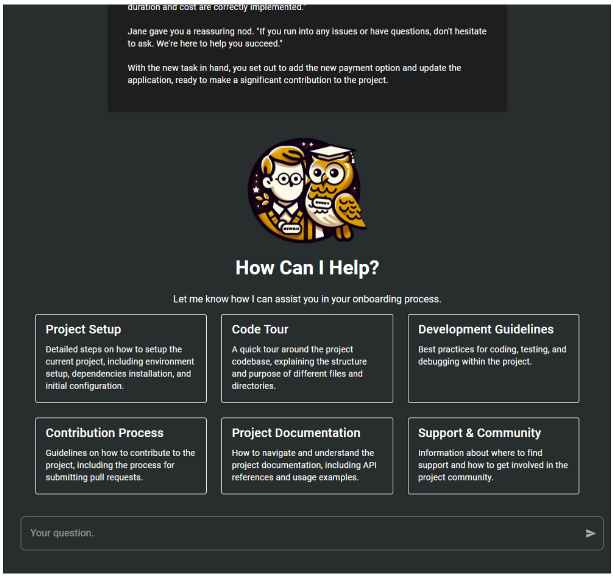
In group A you can use only the Onboarding Buddy chat presented above and no other resources or external help.

Please note, to receive a response from Onboarding Buddy after asking a question in the UI it can take up to 180 seconds based on the complexity of your question.

After you have started the task you will see a window that will display the task you have to complete and the finish task button (upper right):

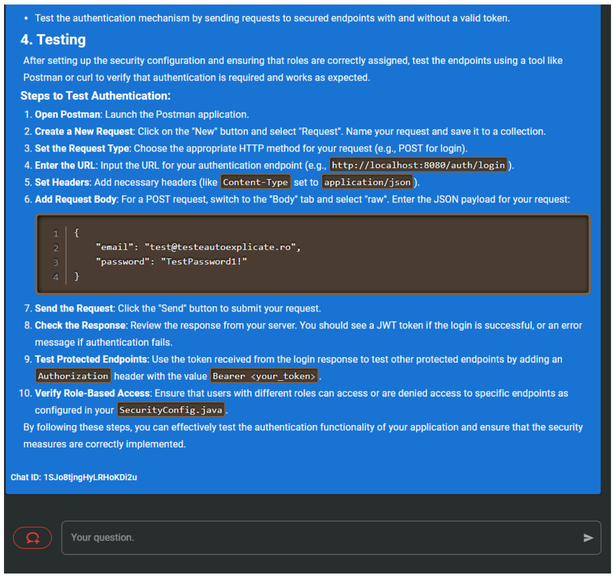


If you scroll down, before the first message you will see a toolbox with suggested steps:

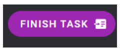


You can choose a suggested step e.g. Project Setup or ask a question by writing it in the "Your question box". In group A you can use only the Onboarding Buddy chat presented above and no other resources or external help.

After the first chat, you can always start a custom chat by pressing the bottom left red button which will create a new chat for you. Please note you can always return to a Task Chat, but never to a custom chat.

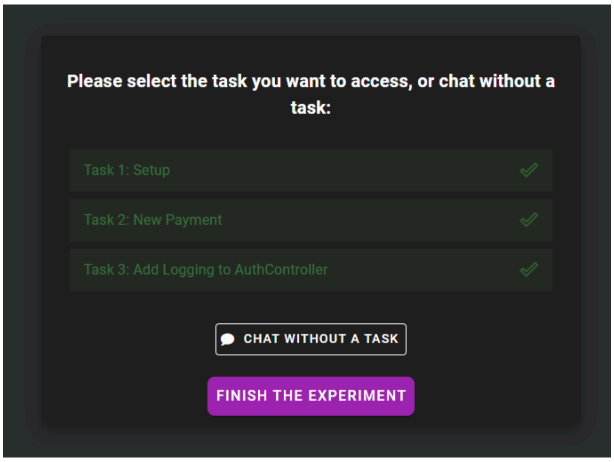


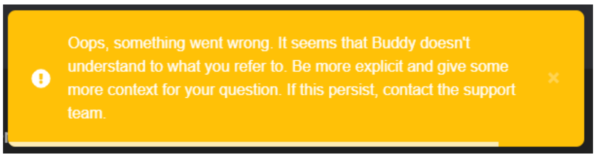
When you are done with the task you can press the top right finish task button and answer some questions about the task you just finished.



!IMPORTANT! Before submitting a task, commit and push your changes to your branch with a clear description. You will receive a reminder in the UI.

After you finish all the tasks you will see the Finish The Experiment button. Press it and you are done.



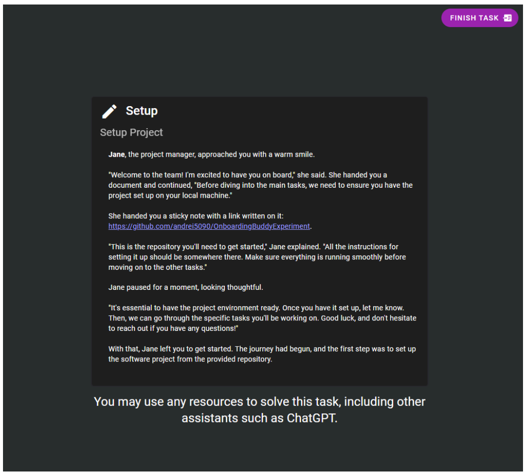


If during the chat you encounter the yellow warning box it means that your question is not specific enough. Try to reformulate it to be more exact. Onboarding Buddy is specifically instructed to refuse to answer questions that are more 'DOs' than 'HOWs'. Onboarding Buddy aims to make the onboarding and learning faster, not to do the onboarding tasks for yourself, so you are allowed to explore the codebase to ask specific questions.

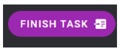
Group B (Without Onboarding Buddy Access):

In group B you can solve the task with any tools, documentation or any other resources including external help, just the sky is the limit.

After you have started the task you will see a window that will display the task you have to complete and a finish task button (upper right):

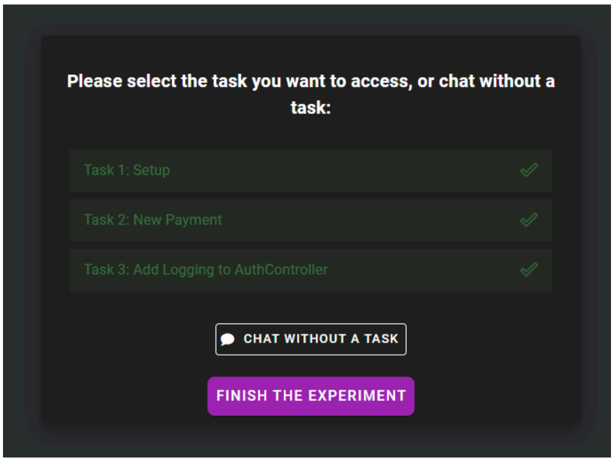


In group B you can solve the task with any tools, documentation or any other resources including external help, just the sky is the limit. When you are done with the task you can press the top right finish task button and answer some questions about the task you just finished.



!IMPORTANT! Before submitting a task, commit and push your changes to your branch with a clear description. You will receive a reminder in the UI.

After you finish all the tasks you will see the Finish The Experiment button. Press it and you are done.



Contact:

For any questions regarding the experiment or technical difficulties we can be contacted at:

- Email: onboardingllms@gmail.com (outside JetBrains)
- Slack: andrei.ionescu@jetbrains.com (inside JetBrains)

Congrats! You Have finished reading the experiment manual. You can now start the experiment!

B

Mini-Study on Embedding performance in RAG

In this mini-study, we will present the techniques and strategies employed in our research to choose the proper embedding for our RAG system. Initially, we will introduce the framework and the data we utilized. Lastly, we will explain the models incorporated into our framework and the metrics we used for evaluation.

B.1. Framework

In this framework section, we present the strategy adopted for the assessment of multiple embeddings in the context of semantic code search, aiming to augment the Retrieval-Augmented Generation (RAG) process. Our study concentrates on evaluating the performance of diverse embeddings, including both open-source and closed-source models, in their capability to accurately retrieve code snippets based on textual descriptions. This section describes our methodology pathway, encompassing data collection, preprocessing, the embedding mechanism, the development of the retrieval system, and the evaluation criteria.

B.1.1. Data

To achieve our goal, we needed to find a dataset that contains textual descriptions of code, paired with the code that was described. For this, we identified three different datasets that might be suitable for our task, but just one fully fitted our goal:
CodeQA [19]: It contains a textual component and a code snippet. The textual component is a question that is paired with the code snippet that requires an answer represented as the ground truth. Since the purpose of our study is not the Q&A capability of the models this dataset was discarded.
Neural Code Search Revisited (Nokia Dataset) [11]: Contains a textual component and a corresponding code snippet. The ground truth is composed of multiple datasets. However, it is limited to Python code and is hard to use it outside the replication package so this dataset was discarded.
CodeSearchNet [12]: Contains a textual component and a corresponding code snippet. The ground truth is composed of multiple datasets. It contains 6 language sub-datasets (go, java, javascript, php, python, ruby) and is easy to use outside the replication package.

Also part of our criteria was to find a dataset that contains a language in the top 10 most used languages on GitHub. On top of that, we wanted to pick a language that is underrepresented in the embedding trainsets due to the limited amount of projects in that language (e.g. ruby is the number 9 language in the number of pushes in 2024, quarter 1 on GitHub[1]). In this way, we will be able to tell if the embeddings tend to memorize or not certain code snippets. Furthermore, we will also be able to tell if the model can generalise for languages with a smaller codebase. This is why, as seen in figure B.1, we chose ruby as the programming language we are going to test the embeddings on. Only **CodeSearchNet**

[1] Language statistics - `https://madnight.github.io/githut/#/pushes/2024/1`

[12] respects the language choice criterion, so it was the natural choice to make for our experimental framework.

B.1.2. Preprocessing

Before any embedding or retrieval activities, a preprocessing phase is essential to cleanse and standardize the input data, mainly tokenization, and document transformation in order to evaluate the result easily. Furthermore, the removal of irrelevant characters or tokens is one of the fundamental preprossessing steps undertaken to ready the data for effective embedding.

B.1.3. Embedding creation

In this phase, we transform the preprocessed code snippets and queries into vector representations through an embedding. This process involves utilizing various embedding models, including 8 open-source models such as all-roberta-large-v1, GloVe, and sentence-t5-large, along with 2 proprietary embeddings, namely OpenAI's text-embedding-3-large and Google's Gecko embedding. The conversion of text and code into numerical vectors is critical, as it allows the semantic search system to comprehend and draw correlations between textual descriptions and code. More details about what embeddings we used will be provided in section B.1.6 and in section B.2, experimental setup and results.

B.1.4. Retrieval System

Another important pivot of our framework is the development of a semantic search system capable of pinpointing relevant code snippets in response to a natural language query. This system likely employs Faiss[2] for efficient similarity search, coupled with a simple L2 distance metric. The choice of L2 distance stands on its efficiency and simplicity in quantifying the similarity between the query vector and code snippets vectors. This retrieval mechanism forms the backbone of our research, enabling the assessment of different embeddings in a practical, real-world scenario.

B.1.5. Evaluation

In our system, due to the nature of the task we want to perform and the dataset, we perform a query on the retrieval system and obtain a list of the closest code snippets to the query, but among these queries, we have at most 1 correct one. Because of this, we considered the following metrics:

- **Hit@K** measures the proportion of relevant documents among the top K retrieved documents. It's suitable for our case because we are interested in the relevance of the top K documents. The formula for Hit@K is:

$$\text{Hit@K} = \begin{cases} 1 & \text{if the relevant document is in the top } K \text{results} \\ 0 & \text{otherwise} \end{cases}$$

- **Average Hit@K (AHit@K)** calculates the average of precision values for each position where a relevant document is found, up to position K. It's useful when the order of the documents matters, as it gives higher scores to systems that return relevant documents earlier. The formula for AHit@K is:

$$\text{AHit@}K = \frac{1}{\min(m, K)} \sum_{k=1}^{K} \text{Precision@}k \times \text{rel}(k)$$

- **Mean Reciprocal Rank (MRR)** calculates the reciprocal of the rank of the first relevant document. It's suitable when because we are interested in the rank of the first relevant document (exactly our case). The formula for MRR is:

$$\text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}(i)}$$

where |Q| is the total number of queries and rank(i) the rank of the relevant document if the document is present, otherwise infinity.

[2] Meta FAISS - `https://ai.meta.com/tools/faiss/`

Model	Parameters	Architecture	Training Data	Vector space
Jina Embeddings v2 Base	137 million	BERT	WikiAnswers, Stack Exchange, Yahoo answers	768
All-MPNet Base v2	109 million	MobileBERT	Wikihow, Stack Exchange, Code search	768
MSMARCO DistilBERT Cosine v5	66.4 million	DistilBERT	MS MARCO Passages dataset	768
All-RoBERTa Large v1	335 million	RoBERTa	WikiAnswers, Reddit comments, S2ORC	1024
GloVe 6B 300d	-	GloVe	Wikipedia 2014 + Gigaword 5	300
Sentence-T5-Large	335 million	T5	Web Formus, SNLI dataset	768
LaBSE	471 million	BERT	CommonCrawl, Wikipedia	768
All-MiniLM-L6	22.7 million	MiniLM	Wikipedia, BookCorpus	384
Google Gecko	-	-	-	768
OpenAI large	-	-	-	3072

Table B.1: The name, parameters, architecture and the training data of the models used for comparison

The Hit@K metric represents the precision of retrieving the correct code snippet within the top K results. High Hit@K values indicate that a model is more likely to return the correct answer within its top K selections. It's especially crucial at lower values of K (e.g., Hit@1, Hit@2), where the "tightness" of the model's accuracy can significantly impact user experience. AHit@K is slightly more involved, considering not just the presence of correct answers within the top K results but also their position. MRR (Mean Reciprocal Rank) focuses on the rank of the first correct result only, offering a single-figure measure of the quality of the top-ranked results.

We also considered F1@K, but Recall@K in this context will always be equal to Hit@K because, among the retrieved documents from our dataset, there is just one possible correct outcome. In the same manner, even if at first glance Normalized Discounted Cumulative Gain (NDCG) seems suitable, it is not. This happens due to what we want to use the embeddings for (RAG) and the relevance established by our method (there is just one relevant document per query).

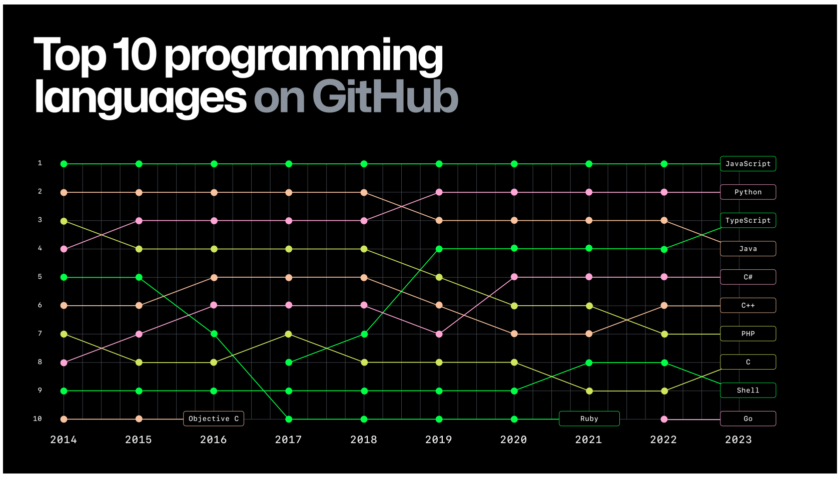


Figure B.1: Top Languages on GitHub 2023 retrieved from Octoverse [6] - the state of open source software

B.1.6. Models

In an attempt to evaluate the performance of different embeddings in the context of query-code semantic search, we have selected 10 embedding models that enable us to transform code and textual inputs into dense, high-dimensional representations that encapsulate their semantic meaning. This section provides an overview of the embedding models selected in the project in Table B.1, outlining the number of parameters used, the architecture they are based on, and the datasets (or examples of some datasets) that the models were trained on.

B.2. Experiments and Results

Our experimental framework was designed to assess the effectiveness of various embeddings in augmenting the RAG process for semantic code search. Specifically, we investigated the ability of different embeddings, encompassing a mix of open-source and proprietary models, to accurately retrieve Ruby code snippets given natural language descriptions. This section outlines the setup of our experiments, detailing the dataset selection, preprocessing steps, embedding models employed, the retrieval system, and the results we obtained based on the experiments and the metrics detailed in the section B.1.5.

B.2.1. Dataset Selection and Preprocessing

The experimental studies were conducted using the code_search_net dataset, specifically targeting the subset of the Ruby programming language. The dataset was accessed and loaded through the Huggingface Datasets library where the language subset was predefined as 'ruby' with the training subset. The loaded dataset was then converted into a Pandas DataFrame for easier manipulation and further processing.

For data filtering and preparation, the DataFrame was filtered to include only test splits and documents of the Ruby programming language. The filtered dataset was then saved to a CSV file in Google Drive based on a naming convention reflecting the language and a timestamp for easy identification and reproduction transparency. Each entry in the dataset was converted into a document with the following structure and usage:

Document =

- Payload to index represented the code snippet ('func_code_string') $\xrightarrow{\text{usage}}$ Vector Database Storing
- Metadata
  - index $\xrightarrow{\text{usage}}$ Identify Initial Document in the Vector Database
  - Function Query ('func_documentation_string') $\xrightarrow{\text{usage}}$ Query the system and check if there is a match

  Each document was then preprocessed using a recursive character text splitter which divided the text based on specified character lengths considering language-specific adaptations (for ruby in this context).

B.2.2. Embedding and Retrieval Pipeline

The core of the experimental setup involved converting the aforementioned documents into embeddings using various pre-trained models (see Models section B.1.6) for comparison. The models tested included both Huggingface, custom VertexAI embeddings, Custom Hugging Face Embeedings and LangChain Embeedings, with specific adaptations for rate limiting when using APIs with request limits. A key custom adaptation in this experiment was the CustomVertexAIEmbeddings class, designed to handle rate limiting by pacing the requests sent to the VertexAI embeddings service. The class included a rate limit mechanism to ensure compliance with the API's query-per-minute (QPM) restrictions.

B.2.3. Retrieval system

The embeddings generated were used to populate a FAISS vector store, creating a retrievable database of document embeddings. A setup was developed to either create a new vector database from the document embeddings or load a pre-existing one from a serialized state, facilitating efficiency in repetitively testing retrieval performance without re-generating embeddings. The FAISS vector store was configured to retrieve the first 15 most similar documents with respect to the query based on L2 distance.

B.2.4. Results

As mentioned in methodology evaluation metrics B.1.5, the retrieval performance was quantitatively assessed using a range of metrics relevant to information retrieval tasks in the context of RAG. These metrics included hit at k (Hit@k), average hit at k (AHit@K) and mean reciprocal rank (MRR). These metrics were computed by comparing the rankings of retrieved documents against the ground truths, measuring both the accuracy of the highest-ranked results and the overall effectiveness of the retrieval across various K-values.

At a glance, as seen in the tables B.2 and B.3, OpenAI's model outperforms others across all metrics, displaying remarkably high precision at every level of retrieval (Hit@1 through Hit@15) and an MRR of 0.798. This suggests that OpenAI's model is not only adept at identifying the most relevant code snippet at the top position but also maintains a high level of accuracy as more results are considered. Gecko follows, showing impressive results, though not as high as OpenAI's. On the other end of the spectrum, the GloVe 6B300d model shows significantly lower scores, indicating its limited capability in the context of RAG. This difference is caused by the architecture of Glove which is a word-level model that generates word embeddings by leveraging global word-word co-occurrence statistics from a corpus. It does not take into account the order of words, which can lead to a lack of understanding of the context or semantics of a code sequence. This can be a significant drawback in tasks where understanding the context is crucial for retrieving the most relevant documents for code.

B.2.5. Insights and possible causes

The high performance of OpenAI's model could be attributed to its cutting-edge language understanding capabilities, likely owing to vast pre-training on a diverse corpus that includes coding languages or contexts similar to the queries presented. This demonstrates the importance of model architecture and training comprehensiveness in achieving high precision in specialized retrieval tasks.

The intermediate performances of models like Jina and Sentence-T5-Large suggest varying degrees of effectiveness in understanding the query context and mapping it accurately to code snippets. Their diverse results across the metrics suggest they may be more suitable for specific types of queries or benefit from further fine-tuning. Important to mention is also the architecture, model size and vector space they output are way smaller than OpenAI's text-embedding-3-large (the length of the embedding is 3072 dimensions) compared with 768-dimensional dense vector space for the other open source models.

Since the purpose of this paper is the performance of augmenting LLM prompts through RAG, based on our experimental results we recommend:

- Model Choice for RAG: Given its superior performance, OpenAI's embedding model emerges as the most promising candidate for a RAG task focused on code snippet retrieval. Its high precision, even at lower K values (e.g., Hit@1 = 0.726), suggests it can effectively identify the most relevant code snippet with fewer retrieved documents. This is crucial in practical scenarios where computational resources and response times are limited.
- Cost-Benefit of Retrieval Size (K): While larger K values ensure a higher likelihood of retrieving the correct result (notably Hit@15 reaches above 0.6 for all models besides GloVe), it also increases the computational load on the subsequent attachment and ranking process but also increases the costs of prompting the LLMs. Given the drop-off rate in precision gain observed beyond K=10 in most models, it may be more cost-efficient to limit K to 10, balancing between performance and resource utilization.
- Implications of MRR: With MRR scores ranging from 0.373 to 0.798, there's a clear indication that some models are significantly more efficient at ranking the correct answer higher within their retrieval sets. As RAG systems often depend on the top-ranked retrievals to generate responses or augmentations, a high MRR score (such as OpenAI's 0.798) is indicative of a model that not only retrieves relevant results but does so in an order that minimizes the downstream processing needed to identify the best output.

Given these insights, it's recommended to prioritize the closed source models like the OpenAI model for RAG tasks for code retrieval. Its proven efficiency not only in identifying relevant results but also in accurately ranking them provides a compelling advantage. However, for applications with stringent computational resource constraints, the trade-off between retrieval size and precision needs to be carefully considered. Limiting K to 10 might offer a balanced approach for most models without significantly compromising the chances of retrieving the correct result. Furthermore, the costs of using the closed-source embeddings were cheaper than getting resources for the open-source ones so the cost involved for such a task is not that high (0.001 euros to generate the embeddings for closed-source vs 0.35$ to generate the open-source ones - but for most of the programming RAG tasks, the pay-to-use embeddings are more handy and cost-effective).

B.3. Threats To Validity and Limitations

Firstly, our investigations were done based on the Ruby programming language, limiting the generalizability of our findings across different programming languages. The specificity of Ruby's syntax and constructs might influence the performance of embeddings in a manner not representative of other languages, thereby constraining our ability to extrapolate our results broadly across the domain of semantic code search (but this is one of the decisions in our research to choose a language that is underrepresented in the training sets).

On the other hand, our experiments employed a mix of open-source and proprietary models without looking deeply into the architectures and training datasets of the proprietary models due to access restrictions (closed source). This limitation limits our capacity to fully understand and interpret the reasons behind the performance disparities observed among the different embeddings, particularly the superior performance of OpenAI's model - memorization or contamination is hard to assess in this case. Furthermore. the choice of metrics, while standard for information retrieval tasks, may not fully capture the nuanced effectiveness of embeddings in retrieving code snippets.

Finally, our cost discussion did not consider the long-term implications of using proprietary models over open-source ones, focusing instead on immediate costs. This short-term perspective does not account for potential changes in pricing or access restrictions that could affect the sustainability of using certain embeddings in production environments.

Model	**Hit@1**	**Hit@2**	**Hit@5**	**Hit@10**	**Hit@15**
jina-embeddings-v2	0.462	0.572	0.685	0.743	0.775
allmpnetbasev2	0.518	0.642	0.765	0.818	0.847
msmarcodistilbertcosv5	0.287	0.38	0.48	0.552	0.585
allrobertalargev1	0.439	0.558	0.689	0.77	0.8
glove6B300d	0.051	0.075	0.102	0.125	0.137
sentencet5large	0.472	0.593	0.701	0.761	0.794
LaBSE	0.294	0.384	0.503	0.577	0.616
allMiniLM-L6	0.514	0.635	0.748	0.805	0.832
Google[3]	0.608	0.716	0.81	0.849	0.867
OpenAI[4]	**0.726**	**0.822**	**0.89**	**0.92**	**0.933**

Table B.2: Performance of different models (Hit metrics)

Model	**AHit@1**	**AHit@2**	**AHit@5**	**AHit@10**	**AHit@15**	**MRR**
jina-embeddings-v2	0.462	0.517	0.6	0.662	0.696	0.558
allmpnetbasev2	0.518	0.58	0.675	0.738	0.771	0.625
msmarcodistilbertcosv5	0.287	0.334	0.405	0.466	0.502	0.373
allrobertalargev1	0.439	0.498	0.595	0.67	0.71	0.549
glove6B300d	0.051	0.063	0.083	0.1	0.11	0.075
sentencet5large	0.472	0.533	0.618	0.679	0.713	0.573
LaBSE	0.294	0.339	0.419	0.486	0.524	0.385
allMiniLM-L6	0.514	0.575	0.663	0.725	0.757	0.616
Google[5]	0.608	0.662	0.738	0.787	0.812	0.695
OpenAI[6]	**0.726**	**0.774**	**0.834**	**0.872**	**0.891**	**0.798**

Table B.3: Performance of different models (AHit metrics and MRR)

[3]Google textembedding-gecko model
[4]OpenAI text-embedding-3-large model

C

Choices of Study Tasks

During our experimental set-up we have considered the following task possibilities for the users to perform, all the tasks are defined concerning the codebase presented in section 5.1:

Experiment Task 1 - New Payment (Medium)

Description: Add a new payment option in the `PaymentOption` enum and update the application to use this new payment option.

Tasks:

1. Add a new value to the `PaymentOption` enum.
2. Update the `PaymentController` to support the new payment option.
3. Create integration tests to verify the new payment option works correctly.

File References:

- Enum: `PaymentOption.java`
- Controller: `PaymentController.java`
- Integration Tests: `integration/*`

Experiment Task 2 - Add Logging to AuthController (Easy)

Description: Enhance the `AuthController` by adding logging statements that log every endpoint execution.

Tasks:

1. Add logging statements at the beginning and end of each endpoint in `AuthController`.
2. Ensure that logging captures parameters passed to each endpoint.
3. Create unit tests to verify that logging occurs as expected.

File References:

- Controller: `AuthController.java`
- Unit Tests: `AuthControllerIntegrationTest.java`

Experiment Task 3 - Add a New API Endpoint to List All Users (Easy)

Description: Add an endpoint to `UserController` that lists all registered users.

Tasks:

1. Add a new endpoint `GET /users` that retrieves a list of all users using `UserServices`.

2. Ensure the endpoint is protected and accessible only to users with the `ADMIN` role.
3. Create integration tests to ensure the endpoint returns the correct data and respects security constraints.

File References:

- Controller: `UserController.java`
- Service: `UserServices.java`
- Integration Tests: `UserControllerIntegrationTest.java`

Experiment Task 4 - Add a Feature to Filter Questionnaires (Medium)

Description: Add a feature to `QuestionnaireController` that allows filtering questionnaires based on certain criteria (e.g., visibility, category).

Tasks:

1. Extend `QuestionnaireController` with a new endpoint `GET /questionnaires/filter` that supports filtering based on query parameters.
2. Modify the `QuestionnaireServices` to handle the filtering logic.
3. Create integration tests to ensure the filtering works as expected.

File References:

- Controller: `QuestionnaireController.java`
- Service: `QuestionnaireServices.java`
- Integration Tests: `QuestionnaireControllerIntegrationTest.java`

Experiment Task 5 - Implement a Custom Exception Handler (Medium, time demanding)

Description: Implement a custom exception handler to manage application-wide exceptions and return meaningful error responses.

Tasks:

1. Create a `GlobalExceptionHandler` class annotated with `@ControllerAdvice`.
2. Implement methods to handle different types of exceptions (e.g., `ResourceNotFoundException`, `BadRequestException`).
3. Update existing controllers to remove individual exception handling.
4. Create integration tests to ensure exceptions are handled correctly and return appropriate HTTP status codes.

File References:

- Controllers: All controllers with individual exception handling (e.g., `AuthController.java`, `UserController.java`)
- New: `GlobalExceptionHandler.java`
- Integration Tests: Tests that trigger exceptions to verify handling.

Experiment Task 6 - Add Caching to User Profile Retrieval (Medium)

Description: Add caching to the user profile retrieval endpoint to improve performance.

Tasks:

1. Use Spring Cache to cache the response of the `GET /user/me` endpoint in `UserController`.
2. Ensure the cache is invalidated whenever the user's profile is updated.
3. Create integration tests to ensure caching works correctly and the profile is updated as expected.

File References:

- Controller: `UserController.java`
- Integration Tests: `UserControllerIntegrationTest.java`

Experiment Task 7 - Implement Comment Flagging (Easy)

Description: Add functionality to `CommentController` that allows users to flag inappropriate comments.

Tasks:

1. Add a new endpoint `POST /comments/flag` that allows users to flag comments.
2. Modify `CommentServices` to handle flagging logic.
3. Create integration tests to ensure the flagging functionality works correctly.

File References:

- Controller: `CommentController.java`
- Service: `CommentServices.java`
- Integration Tests: `CommentControllerIntegrationTest.java`

Experiment Task 8 - Add User Role Management (Medium)

Description: Add functionality to `AdminUserServices` to manage user roles.

Tasks:

1. Add methods to `AdminUserServices` to assign and revoke roles.
2. Create endpoints in `AdminController` to manage user roles.
3. Create integration tests to verify role management works correctly.

File References:

- Service: `AdminUserServices.java`
- Controller: `AdminController.java`
- Integration Tests: `AdminControllerIntegrationTest.java`

Experiment Task 9 - Add an API to Retrieve Questionnaire Statistics (Easy)

Description: Add an endpoint to `StatisticsController` to retrieve statistics for a specific questionnaire.

Tasks:

1. Add a new endpoint `GET /statistics/questionnaire/{id}` to `StatisticsController`.
2. Modify `StatisticsServices` to retrieve and format the statistics.
3. Create integration tests to ensure the endpoint returns correct statistics.

File References:

- Controller: `StatisticsController.java`
- Service: `StatisticsServices.java`
- Integration Tests: `StatisticsControllerIntegrationTest.java`

Experiment Task 10 - Implement Bulk User Deletion (Medium)

Description: Add functionality to `UserController` to delete multiple users at once.

Tasks:

1. Add a new endpoint `DELETE /users` that accepts a list of user IDs.
2. Modify `UserServices` to handle bulk deletion.
3. Create integration tests to ensure bulk deletion works correctly.

File References:

- Controller: `UserController.java`
- Service: `UserServices.java`
- Integration Tests: `UserControllerIntegrationTest.java`

Experiment Task 11 - Implement Questionnaire Duplication (Medium)

Description: Add functionality to `QuestionnaireController` to duplicate an existing questionnaire.

Tasks:

1. Add a new endpoint `POST /questionnaires/duplicate/{id}` to duplicate a questionnaire.
2. Ensure `QuestionnaireServices` handles the duplication logic, including questions and metadata.
3. Create integration tests to ensure the duplication works correctly.

File References:

- Controller: `QuestionnaireController.java`
- Service: `QuestionnaireServices.java`
- Integration Tests: `QuestionnaireControllerIntegrationTest.java`

Experiment Task 12 - Extend User Authorization (Medium)

Description: Add a new role with specific permissions and ensure that these permissions are enforced.

Tasks:

1. Define a new role in `Roles.java`.
2. Add necessary checks in `SecurityConfig.java` and any other relevant classes.
3. Update `Authorizations.java` and `AuthorizationRepository.java` to handle the new role.
4. Add integration tests to ensure that the new role functions correctly and permissions are enforced.

File References:

- Model: `Roles.java`
- Config: `SecurityConfig.java`
- Integration Tests: `/integration/*`