Thomas Langs, BSc

Large Language Models for Software Development: Scenario-Driven Model Selection and Practical Exploration

Master's Thesis to achieve the university degree of Master of Science
Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisors Roman Kern, Mark Kröll

Institute of Human-Centered Computing
Head: Assoc.Prof. Dipl.-Ing. Dr.techn. Denis Helic

Graz, January 2025

Acknowledgments

First and foremost, I would like to express my gratitude and appreciation towards my supervisors Assoc.Prof. Roman Kern and Dr.techn. Mark Kröll, for the continuous support, and the invaluable feedback I received from both of you. Your excellent supervision and insightful suggestions throughout the entire process were incredibly helpful. Furthermore, I would like to extend the gratitude and appreciation to my family for the continuous support on my journey.

Abstract

Following the recent surge in popularity of Large Language Models (LLMs), many models that were also trained on source code now exist and can aid with software development (SD). However, this raises the question of how to select and compare LLMs for SD and how well the chosen model can aid the implementation of an actual software project. To shed light on this, we defined a practical SD scenario of a startup implementing an expense-tracking application using Java (and React). Based on criteria extracted from said scenario, we selected eight models. These were then quantitatively compared on five aspects of Java SD (i.e. code generation, code completion, unit test generation, method comment generation, and automatic program repair) using benchmarks to select the overall best-performing model. We then used an IDE-integrated tool (GitHub Copilot) based on the selected model (GPT-4(o)) to implement part of the application. The quantitative model comparison revealed that the three best-performing and the three worst-performing models respectively stayed the same respectively for four aspects, with method comment generation being the only exception. Moreover, looking at three benchmarks that provide a more fine-granular evaluation, the three best-performing models primarily failed due to difficulties producing compilable and functionally correct code. However, syntactical correctness did not seem to be a problem in most cases. For the implementation phase, our observations revealed that GitHub Copilot was generally able to produce good starting points; however, the code frequently contained small issues and inconsistencies, as well as occasionally larger problems, like using libraries incorrectly (e.g., UI interactions and reactive state changes for test cases). This usually also depended on the complexity of the task the model had to solve. Furthermore, we observed that having a solid understanding of the programming language and framework, utilizing explicit prompting, and supplementing the prompt with relevant files are all essential to use the tool effectively. Generally, the tool was able to aid implementation efforts, although one must also be aware of potential pitfalls. Summing up, this thesis provides the intrigued reader with insights into the LLM selection process guided by a scenario as well as into subsequent implementation efforts with an LLM-based tool.

Contents

List of Figures

1. Introduction

Generative Artificial Intelligence (GenAI) for software development (SD) has become a highly active field of research in recent years, both in academic and commercial fields. This development has led to a situation where many different Large Language Models (LLMs) exist that can be used for SD. This is illustrated by the fact that Chatbot Arena (Chiang et al., 2024) currently[1] lists 187 LLMs in the code section of their leaderboard. Such a vast landscape of models raises the question of how one can select a model to use in the practical implementation of a software project and how helpful such a model can be.

Therefore, this thesis explores how to choose a model given a concrete scenario using a quantitative approach. This is followed up by qualitative assessment of the utility of the selected model in aiding the implementation of the software project from the scenario.

The theoretical part of our thesis is comprised of two chapters: Background (Chapter 2) and Related Work (Chapter 3). The background chapter provides information on the evolutions leading up to the first LLMs. Additionally, it offers an overview of current LLMs and commonly used evaluation measures. It closes by detailing approaches to tackling the five Java SD aspects (see Table 1.1) prior the advent of LLMs. The related work chapter provides an overview of important benchmarks and datasets for evaluating the five aspects of Java SD (see Table 1.1), informing the selection process for the model comparison. Moreover, this chapter illustrates different approaches to tracking developer behavior when using LLMs, informing our approach to tracking observations and interactions in the practical exploration. The last part of the related work chapter details similar works and subsequently highlights how our thesis differs.

The practical part of our thesis is structured into four main steps: (1) defining the practical scenario, (2) selecting an initial group of models to be considered, (3) comparing the models using quantitative methods to select the best-performing one, and (4) qualitatively exploring the practical scenario by implementing part of the application. The following paragraphs provide further details on these steps:

[1]https://lmarena.ai/?leaderboard (Accessed: 05.01.2025)

1. Practical Scenario - Definition (Chapter 4) The scenario we explore is that of a one-person startup that aims to develop a web application that allows users to track their expenses. The application is split into a frontend and a backend part. While this thesis focuses on Java SD, as this is the language we have the most experience with, the tech stack consists of a fronted based on Next.js, a React framework, and a backend based on Spring Boot, a Java framework. This is done to use an up-to-date approach for building a front end. Additionally, this allows us to experience the usefulness of the selected LLM for React, a language we were unfamiliar with prior to this thesis.

Furthermore, which models can be considered is constrained by the limited funds available to the startup. However, code privacy, regarding patents or intellectual property, are not a primary concern.

To simplify the exploration process in step 4 and to be able to focus on the actual implementation, we opt to provide user stories, a data model, and UI mockups.

This scenario is chosen as it provides an application that can (partially) be implemented by a single person. Although this scenario is quite specific, our approach to selecting and evaluating a model can arguably be applied to various scenarios and needs.

2. Initial Model Selection (Chapter 5) Considering the scenario outlined above, we define criteria that a model should fulfill in order to be useful for all parts of our evaluation. This includes, among other things, being able to run as additional software on a developer laptop or being available as a hosted offering.

Based on the defined criteria, we select eight fitting models, five of which are hosted offerings. The other three are smaller models that can realistically run on a developer machine in addition to existing applications like an IDE.

3. Model Comparison (Chapter 6) To narrow the selection to a single model, we quantitatively evaluate the eight models on five aspects of Java SD. These are chosen based on our experience with using Java in practice, further considering the availability of sensible benchmarks. The five aspects we evaluate are given in Table 1.1.

Note that the terms code generation and code completion are sometimes used inconsistently in literature, our thesis uses the following distinction:

Code generation is the task of generating complete source code snippets, primarily, but not necessarily exclusively, based on natural language text or comments.

Code completion is the task of completing the current statement, line, block, or method, at least based on the code before the caret position.

Aspect	Description
Code Generation	The task of synthesizing source code from a natural language (NL) description and context information.
Code Completion	The task of generating further code to complete an existing snippet, condition or statement.
Unit Test Generation	The task of synthesizing a test case for a method under test based on an NL description and additional context information.
Method Comment Generation	The task of generating a comment that describes what a method does. Judging based on the similarity to a reference description.
Automatic Program Repair (APR)	The task of automatically fixing bugs within a method based on the method in question and additional context information.

Table 1.1.: The **Aspects of Java SD** that are part of the quantitative model comparison.

We rank the eight models based on their performance on the five aspects, each of which are evaluated using a benchmark. The ranks range from 1 (best) to 8 (worst) for each aspect, using the average rank as a tiebreaker. These results are then combined using the arithmetic mean to arrive at the final rank for each model.

4. Practical Scenario - Exploration (Chapter 7) Using the results from the model comparison, in the last step, we set out to see how helpful the best-performing model can be for implementing (part of) the application. For this we utilize an existing tool (GitHub Copilot) that is built around the selected model (GPT-4(o)) and can be tightly integrated into the IDE. The implementation follows the user stories already present from the first step. For this qualitative evaluation, we start with an over-reliance on the tool, even relying on it for minor changes, and adapting the usage over time if we find it not helpful. We keep track of our our subjective observations and how our usage changes using pen and paper. Furthermore, we record the screen and IDE interactions to be able to refer back to them in case there is anything unclear in our notes. Furthermore, the maximum duration of our practical exploration is limited to 80 hours, with the ability to stop earlier if we cease to make notable new observations. The findings of this exploration are presented in Chapter 7.

In this thesis, we address the following research questions (RQs):

- **RQ1:** What aspects of an LLM are required to be useful for both our practical exploration and the comparison? (addressed in Step 2)
- **RQ2:** How do the eight models compare on their benchmark performances? (addressed in Step 3)
- **RQ3:** In which parts of the benchmarks do the highest-scoring models perform well, and where is room for improvement? (addressed in Step 3)
- **RQ4:** To what extent can GitHub Copilot aid in implementing a software project? (addressed in Step 4)

With this thesis, we make the following key contributions:

- We provide insights into how to find a fitting model based on a practical scenario.
- We identify where, in the benchmarks we use for model comparison, the best-performing models perform well and where they do not.
- We present our findings from implementing part of the application from the scenario with the aid of GitHub Copilot, which is based on a variant of OpenAI's GPT-4 models.

2. Background

This chapter serves as an introduction to many of the topics we touch on during this master thesis. These include the history of how LLMs came to be (2.1), an overview of current LLMs (2.2), an overview of commonly used evaluation measures for LLMs (2.3), previous approaches to tasks related to Java software development(2.4) and tools for measuring code quality (2.5).

2.1. What innovations in Deep Learning led to the advent of LLMs?

In this section, we summarize the history of how LLMs came to be, starting with Recurrent Neural Networks (RNNs) and ending with some of the first ”modern” models like BERT (Devlin et al., 2019) and GPT-1 (Radford and Karthik, 2018).

One key capability that enables large language models (LLMs) to be this powerful is incorporating previous inputs and outputs, say previous parts of a conversation, into the output generation process during normal execution.

This contrasts classical feed-forward neural networks (NNs), which cannot retain information about previous inputs and outputs outside the learning phase. The idea of enabling NNs to retain a state between executions is not new, however. The first explorations already took place in the twentieth century with early Recurrent Neural Network (RNN) versions. Graves (2014) and Schmidt (2019) provide further information on RNNs.

However, early RNNs had a fundamental problem preventing them from retaining information over many time steps: their gradients would tend to either blow up or vanish. The latter case is especially interesting here, as this leads to the downside that ”learning to bridge long time lags takes a prohibitive amount of time, or does not work at all” (Hochreiter et al., 2001). This problem was solved by the introduction of Long-Short-Term Memory (LSTM) cells, which enabled information retention across more than 1000 time steps (Hochreiter, 1997). LSTMs were then applied in, among other things, music (Boulanger-Lewandowski et al., 2012; Eck and Schmidhuber, 2002), text, and handwriting generation (Graves, 2014).

The next step along the journey to LLMs came with the introduction of Encoder-Decoder Architectures (Sutskever et al., 2014; Cho et al., 2014). This type of architecture allows to *encode* a variable-length input to an internal state with a fixed size. From this internal state, the output, again of variable length, can be *decoded*. One field of application for these models is Neural Machine Translation (NMT). Here, the goal is to *encode* text from the source language and then *decode* the text into the target language. This type of architecture was first introduced by Sutskever et al. (2014) and Cho et al. (2014) and subsequently improved by the introduction of attention mechanisms (Bahdanau et al., 2016; Luong et al., 2015; Jean et al., 2015).

Bahdanau et al. (2016) explain the attention mechanism fittingly in their paper: ”The decoder decides parts of the source sentence to pay attention to. By letting the decoder have an attention mechanism, we relieve the encoder from the burden of having to encode all information in the source sentence into a fixed-length vector. With this new approach, the information can be spread throughout the sequence of annotations, which can be selectively retrieved by the decoder accordingly.” (Bahdanau et al., 2016).

The next major evolution of the encoder-decoder architecture came with the introduction of transformers (Vaswani et al., 2017), which rely only on attention and not on recurrences or convolutions. This allows the model to be trained significantly faster. One key building block of the transformer architecture is the Multi-Head-Attention layer (see Figure 2.1). It is used in two ways:

1. Self Attention: with **K**ey, **V**alue, and **Q**uery coming from the same source, allowing it to attend to all outputs of the previous layer (Vaswani et al., 2017).

2. Encoder-Decoder Attention: with **K** and **V** coming from the encoder and **Q** coming from the decoder. Allows decoder input (all previously generated output) to attend to the input (that is fed through the encoder) (Vaswani et al., 2017).

Subsequently, the transformer architecture was then applied in (large) language models like BERT (Devlin et al., 2019) and GPT(-1) (Radford and Karthik, 2018). Both models are similar in that they are pre-trained on a general task and then fine-tuned on a specific task. However, they differ in architecture and training methods. GPT is trained on texts to predict the next token, while BERT is trained as a Masked Language Model (MLM). When training a MLM, the task is to predict randomly blanked-out tokens. Moreover, these models also differ in architecture: GPT is a decoder-only model, while BERT is an encoder-only model.

In this section, we gave a brief overview of the technical evolutions that led to the advent of LLMs, primarily focusing on the field of Natural Language Processing (NLP).

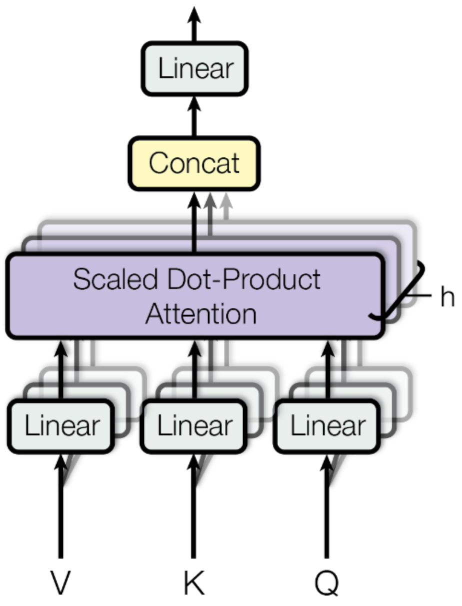


Figure 2.1.: **Multi-Head-Attention**. An illustration of a Multi-Head-Attention layer that consists of **h** attention heads and has the following inputs: **K**ey, **V**alue, and **Q**uery. Taken from Vaswani et al. (2017).

2.2. Overview of Current LLMs

Having introduced the history leading up to LLMs, we aim to provide an overview of currently available models.

We start with more general models like (Chat-)GPT (Radford and Karthik, 2018; Brown et al., 2020; OpenAI et al., 2024) or Gemini (Team et al., 2024a,b) continue to SD-specific models like OpenAI Codex (Chen et al., 2021) and StarCoder (Li et al., 2023; Lozhkov et al., 2024), before concluding at integrated offerings like GitHub Copilot[1] or Replit AI[2].

As the field of LLMs is highly active, with new models and model versions constantly being released, we expect the currently relevant models to be replaced by even better ones soon. Therefore, we want to refer the reader to a regularly updated leader-board[3] that provides scores of various models in different domains.

[1]https://github.com/features/copilot (Accessed: 14.10.2024)
[2]https://replit.com/ai (Accessed: 14.10.2024)
[3]For example: https://lmarena.ai/?leaderboard (Accessed: 25.12.2024)

2.2.1. General Models

These models were not specifically trained to perform well on code or coding-related tasks. However, as general models are usually at least *also* trained on code, they are, therefore, also relevant parts of the landscape of LLMs for code.

Prominent commercial models that fit into this criterion are, among others, ChatGPT (Brown et al., 2020; OpenAI et al., 2024) from OpenAI, Gemini (Team et al., 2024a,b) from Google, and Claude (Anthropic PBC, 2024) from Anthropic.

However, there also exists a multitude of prominent open models, which can be grouped into two categories: open-weight and open-source[4]. The difference between these two categories is that only the weights are publicly available for an open-weight model, whereas everything is for a full open-source model.

Prominent open-weight model families include LLaMA (Touvron et al., 2023a,b; Dubey et al., 2024), and Gemma (Team et al., 2024c), as well as Mistral (Jiang et al., 2023) and Mixtral (Jiang et al., 2024a). An example of a completely open-source model is OpenLLaMA [5]. This is an approach to creating a model similar to LLaMA but with more favorable licensing terms (Apache 2.0). Moreover, the creators of OpenLLaMA also detail how the model was trained, including the datasets and tools used.

General models, like the ones listed above, are usually trained on a various tasks, including open-source code. Intuitively, this would give them good ”world knowledge” and some capabilities for handling source code. This is underlined by the fact that currently, one Leaderboard for HumanEval[6] shows both GPT-4o and Claude-3 Sonnet in the Top 10. Moreover, all other models in the top 10 are based on a Model from OpenAI.

2.2.2. Models Specifically Trained for Code

There not only exist general models, but also ones that are trained or fine-tuned for specific tasks, like synthesising code. This means that such models were either trained with the coding task as a primary objective or the developers took a general model and fine-tuned it on coding-related tasks. One example of a code dataset commonly used in the training procedure is ”The Stack”(Kocetkov et al., 2022; Lozhkov et al., 2024). Moreover, when looking at the models below, it can be

[4]https://promptengineering.org/llm-open-source-vs-open-weights-vs-restricted-weights/#open-weights-vs-open-source-for-language-models (Accessed: 14.10.2024)
[5]https://github.com/openlm-research/open_llama (Accessed: 21.08.2024)
[6]https://paperswithcode.com/sota/code-generation-on-humaneval (Accessed: 21.08.2024)

noted that the training or fine-tuning data is usually collected from open-source repositories on pages like GitHub.

When looking at the category of fine-tuning approaches, noteworthy models include Codex (Chen et al., 2021) from OpenAI, which is based on GPT-3 (Brown et al., 2020), CodeLlama (Rozière et al., 2024) (based on Llama2 (Touvron et al., 2023b)), and CodeGemma (Team et al., 2024d) (based on Gemma (Team et al., 2024c)). For models that were trained from scratch, approaches include Starcoder (Li et al., 2023; Lozhkov et al., 2024), ReplitCode v1.5 3b (Replit, Inc, 2023), and AWS CodeWhisperer (O'Neil, 2021).

One model series that is not consistently within any of the above-mentioned categories is DeepSeekCoder (Guo et al., 2024). The first version of this model had been trained from scratch, with the developer switching to a fine-tuning approach for the second version. They based the model there on their more general DeepSeek-V2 (DeepSeek-AI et al., 2024a). Furthermore, with their newest model, DeepSeekV-2.5, they removed the separation between the general and code-specific models altogether, combining both V2 models into one[7].

In addition to the models mentioned above, a list of even more currently available models can be found in Jiang et al. (2024b) on page 4. There, the authors present them in the form of a chronological overview. They list 70 different models, most of which, but not all of them, are directly intended to generate code.

2.2.3. Small Models

Another interesting direction some model developers turn to is providing their models with varying amounts of parameters. This sometimes also includes models with few parameters (<4 billion parameters). Such a models size is particularly interesting, as it is the most likely to be able to run as an additional software on a developer machine with sufficient performance (token generation speed).

One example of this approach is Phi3 (Abdin et al., 2024), where the smallest model has 3.8b parameters. Moreover, the authors of this paper also managed to run this model on a phone with a memory footprint of 1.8GB using quantization. Another example that features a small and even very small model is Qwen2 (Yang et al., 2024). The authors released models with 0.5b and 1.5b parameters respectively.

One additional approach worth mentioning is the "Full Line Code Completion" model from JetBrains (Semenkin et al., 2024). There, the authors take a different approach by creating small models trained explicitly for a single language. This

[7]`https://api-docs.deepseek.com/updates/#deepseek-coder--deepseek-chat-upgraded-to-deepseek-v25-model` (Accessed: 14.10.2024)

allows them to provide line completion suggestions in a purely local manner.

2.2.4. Integrated Approaches

The following gives an overview of integrated approaches, which are either directly bundled with an editor or installable as a plugin. Integrated approaches usually consist of at least one LLM and are (deeply) integrated into the code editor. Intuitively, there are multiple advantages to such a deep integration: Firstly, it allows for context information to be automatically supplemented when writing prompts. This can allow users to write shorter prompts, as they do not need to explain the whole context themselves. Secondly, tight coupling enables using LLMs for code completion, which is similar to but more potent than current suggestion systems.

Several IDEs come bundled with LLM features. These include JetbrainsAI and "Full Line Code Completion" (Semenkin et al., 2024) for JetBrains IDEs, Intellicode (Svyatkovskiy et al., 2020) for Visual Studio, or ReplitAI for the Replit[8] online IDE. Moreover, approaches that can be integrated via a plugin include GitHub Copilot (Chen et al., 2021) , AlphaCodium (Ridnik et al., 2024), and tabnine[9].

2.3. Evaluation Measures of LLMs (NLP + Code)

Having established the evolution of approaches that ultimately led to LLMs, this leaves the question of how to evaluate such models. One way to assess the performance of LLMs on generative tasks is to compare a model's output with a reference text or code. Multiple approaches exist to quantify the relation between the desired and actual output. In the following, we discuss prominent evaluation measures that can be used to evaluate the text and code output of LLMs. We start by intuitively defining the need for capable measures for Language Modelling and then move to code-specific measures for evaluating LLMs.

Arguably, the most straightforward approach to comparing two texts is using exact match (EM). EM is a binary measure that distinguishes whether two texts are exactly the same or not. While this measure is simple, it is not robust. This becomes obvious when comparing the following two strings: (1) "Happy Birthday" and (2) "Happy Birthday!". Whilst their meaning is practically the same, EM classifies them as entirely different.

[8]https://replit.com/ (Accessed: 14.10.2024)
[9]https://www.tabnine.com/ (Accessed: 26.12.2024)

This drawback leads to Edit Distance[10] (ED) being the next candidate. ED works by computing a cost function on a per-character basis. Matching characters incur a cost of 0, whilst additional or missing characters cost 0.5 each. A wrong character incurs a cost of 1 - this is intuitive when visualizing character substitution removing one character (0.5 cost) followed by the insertion of another one (0.5 cost). Whilst this approach is already more fine-grained than EM, it only compares texts on a per-character basis and ignores all surrounding context.

n-gram Text Measures One approach to incorporate the context are n-grams[11], where the text is segmented with a **n** element-wide sliding window, e.g., using a character or token granularity. N-grams are usually not used in isolation but as a building block for more complex approaches. One popular n-gram-based measure is BLEU (Papineni et al., 2001), initially developed for machine translation. It combines a modified n-gram approach with a "brevity penalty." This penalty exists to ensure that the translation does not get too short. In the original paper, the authors reported that the BLEU score correlates with human judgment.

Another approach is chrF(++) (Popović, 2015, 2017), which calculates the F-score based on character n-grams. The approach can be controlled using selectable system parameters, namely the n-gram length and β. β controls the importance of recall. The main difference introduced by chrF++ (Popović, 2017) is that the authors added word uni- and bi-grams to the formula.

Another approach that is based on n-grams is ROUGE (Lin, 2004). There the authors utilize the rate of n-grams that occur in both texts to compare them. Moreover, they also propose approaches based on the longest common subsequence (ROUGE-L) word uni- and bi-grams (ROUGE-1/-2).

One final approach we want to highlight is METEOR[12] (Denkowski and Lavie, 2014), which addresses shortcomings of the BLEU score. METEOR is based on alignment[13] with word unigrams and consists of two main parts. F_{mean}, the first part, is computed based on precision and recall of the alignment. It weighs recall significantly higher than precision. The second part is a fragmentation penalty that can reduce the score by up to 50% if the word groupings differ considerably from the reference output (i.e., if a phrase was torn apart).

[10] Also called Levenshtein Distance (see `https://en.wikipedia.org/wiki/Levenshtein_distance` (Accessed: 11.08.2024))

[11] `https://de.wikipedia.org/wiki/N-Gramm` (Accessed: 11.08.2024)

[12] `https://en.wikipedia.org/wiki/METEOR` (Accessed: 12.08.2024)

[13] What parts of the source text map to what parts of the target text

2.3.1. Code Measures

Theoretically, one could also take a measure like BLEU (Papineni et al., 2001) and evaluate it on code - after all, code is still just a unique form of text. However, these measures are not tailored to the specific characteristics of code. In the following, we therefore present two measures that are: (1) CodeBLEU and (2) ChrystalBLEU.

One approach that takes the advances of NLP-specific measures and extends them to code is CodeBLEU (Ren et al., 2020). It is based on BLEU and combines syntactic and semantic measures by adding (1) abstract syntax trees (ASTs) and (2) data flow graphs (DFGs). The AST[14] is a way of representing the code as a tree of operations with branches usually being parts of an if clause (Branches: condition, if, (else)) or a loop (Branches: condition, body). The score that then influences CodeBLEU is calculated based on how many sub-trees of the reference solution are in the output. The DFG[15] is a graph that illustrates how information flows in a function. The nodes represent operations, and the edges represent information flowing between the operations. For CodeBLEU, the DFG part of the score is calculated based on the number of data flows that match the reference solution.

Another approach that extends BLEU to code is ChrystalBLEU (Eghbali and Pradel, 2022). There, the authors compute the **k** most common n-grams of the code and remove them from the BLEU calculation. Their paper shows, that in Java, the most common 2- and 4-grams mainly consist of brackets and a few common keywords like "public" or "return." The authors show that their approach outperforms BLEU in accuracy, precision, and F1-score in code clone detection. Although, BLEU outperforms ChrystalBLEU in terms of recall value.

There also exists an approach called CodeBERTScore (Zhou et al., 2023), which compares two code snippets by encoding the instruction-code pairs[16] using language-specific CodeBERT (Feng et al., 2020) models. The code part (without punctuation) is then compared by pairwise cosine similarity which produces a similarity matrix. The precision value of their score is then computed by applying the *max* operation along all rows. The same is done for recall by applying the operation on all columns. Their paper shows, that their measure more strongly correlates with functional correctness than most other measures (only exception: METEOR).

[14]`https://en.wikipedia.org/wiki/Abstract_syntax_tree` (Accessed: 12.08.2024)

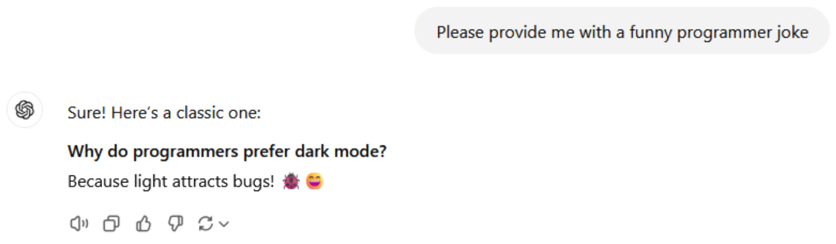
[15]`https://bears.ece.ucsb.edu/research-info/DP/dfg.html` (Accessed: 12.08.2024)

[16]Encodes (instruction, reference code) and (instruction, generated code)

Important Measure Whilst **pass@k** (Kulal et al., 2019; Chen et al., 2021) is not a measure that compares two texts or code snippets with each other, it is still important to highlight it here, as it is often used in the field of LLMs when evaluating them on their abilities related to code. Pass@k is the probability that **at least 1** out of **k** generated outputs passes all requirements. These requirements could, for example, be test cases or a code quality gate.

2.3.2. Human Evaluation of LLMs

In the previous parts, we have shown how different measures can be used to evaluate the performance. While these measures can easily be computed on thousands of examples and allow for a straightforward comparison of models, this approach might not be suitable for all use cases. This becomes clear when considering that multiple approaches and levels of detail exist with which a concept can be explained. Such a situation calls for a different form of evaluation, like using the users of LLMs to evaluate their performance.


Please provide me with a funny programmer joke

Sure! Here's a classic one:

Why do programmers prefer dark mode?

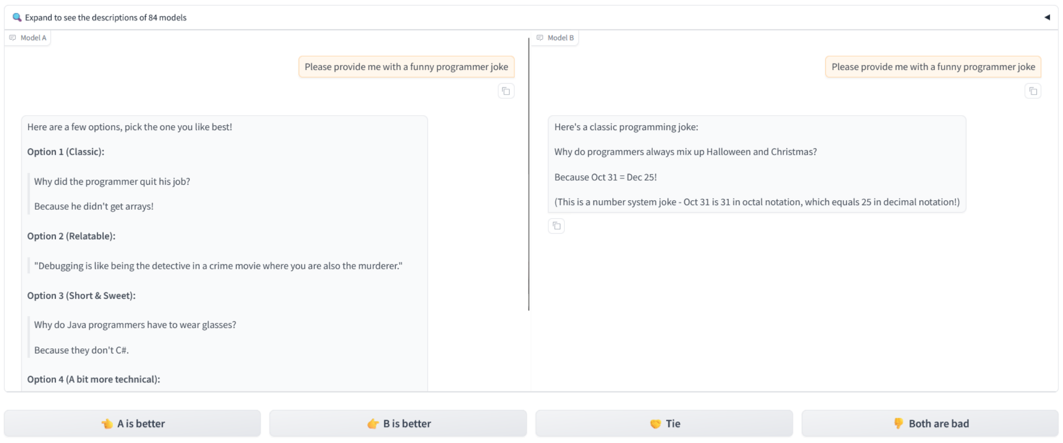
Because light attracts bugs! 🐞 😄

Figure 2.2.: **LLM Answer Feedback**. An example of how users can provide feedback on the answer received by the model. A thumbs up signifies a good answer, and a thumbs down signifies a bad answer. Screenshot taken from ChatGPT (GPT-4o).

One straightforward approach used by many LLM providers nowadays is to give users feedback buttons to allow them to mark especially good or bad answers (see Figure 2.2). While this allows the provider of a model to gain information on the perceived quality of the model it does not lend itself to model comparison. To address this shortcoming, one can let users rank the outputs of multiple models based on the same prompt. An example taken from Chatbot Arena[17] can be seen in Figure 2.3.

Moreover, human evaluations do not have to be collected from the actual users. To compare models, one approach is to ask human experts or evaluators to judge

[17]https://lmarena.ai/?leaderboard (Accessed: 26.12.2024)



Expand to see the descriptions of 84 models

Model A

Please provide me with a funny programmer joke

Here are a few options, pick the one you like best!

Option 1 (Classic):

> Why did the programmer quit his job?
>
> Because he didn't get arrays!

Option 2 (Relatable):

> "Debugging is like being the detective in a crime movie where you are also the murderer."

Option 3 (Short & Sweet):

> Why do Java programmers have to wear glasses?
>
> Because they don't C#.

Option 4 (A bit more technical):

Model B

Please provide me with a funny programmer joke

Here's a classic programming joke:

Why do programmers always mix up Halloween and Christmas?

Because Oct 31 = Dec 25!

(This is a number system joke - Oct 31 is 31 in octal notation, which equals 25 in decimal notation!)

Figure 2.3.: **LLM Answer Side-by-Side Feedback**. An example of an application that allows the user to rank the outputs of two different models. Screenshot taken from Chatbot Arena.

model output concerning specific questions. This can be seen in Iyer et al. (2016), where the authors asked humans to rate C# and SQL comments regarding their ”naturalness” and ”informativeness” on a scale from 1 to 5.

Another way to evaluate, for example, the usefulness of an LLM for software development is to conduct studies of how users use the tools and their experience with it. One example is the study by Tang et al. (2024a) where the authors tracked user interactions a tool called CodeGRITS (Tang et al., 2024b). Moreover, they had the participants fill out cognitive workload assessments and conducted semi-structured interviews.

2.4. Historic Approaches

The tasks of code generation, test case generation, code completion, unit test generation, method comment generation and automatic program repair (APR) are not purely a result of the advent of LLMs. Instead, they were explored and evaluated before the first LLM was even trained. In the following, we examine the five tasks more thoroughly and highlight some ”historic” approaches.

2.4.1. Code Generation

Code generation is the task of generating methods, classes or even entire projects using a non-code definition. In the past, the definitions took various forms, such as diagrams or models, in combination with templates.

Diagrams One popular diagram-based approach to generating code is using the Unified Modeling Language (UML)[18], specifically UML class diagrams[19]. This allows for the specification of details of the class itself, like methods or fields. Moreover, it also allows to define the relationship between classes, like inheritance. However, such modeling techniques usually have no way to specify a method's behavior directly. This means that tools usually generate empty method bodies that must be manually filled with code. Additionally, the fact that the actual behavior is not included in the model raises the question of how to deal with the method bodies when adapting the UML model.

Models and Templates One approach using text-based models and templates is Telosys[20], which is based on the Apache Velocity template engine[21]. The modeling part of Telosys is similar to UML in that it allows the user to model data classes and their relationships. Additionally, this tool offers the ability to define velocity templates that describe how models are translated to code. The combination of templates and models can even be used to generate entire applications.

For example, there exists a template[22] that takes a data model and converts it into a Spring Boot JPA REST application that allows Create, Read, Update, and Delete (CRUD) operations on the data.

2.4.2. Code Completion

The task of code completion involves generating code to complete a statement based on information about the context, which is usually gathered using static analysis[23]. Most IDEs provide some implementation of this feature out of the box. Moreover, code completion can be defined using various granularities ranging from the next token to the complete method.

[18] `https://de.wikipedia.org/wiki/Unified_Modeling_Language` (Accessed: 13.09.2024)

[19] `https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/` (Accessed: 13.09.2024)

[20] `https://www.telosys.org/` (Accessed: 13.09.2024)

[21] `https://velocity.apache.org/` (Accessed: 13.09.2024)

[22] `https://github.com/telosys-templates/java-rest-springboot-jpa-basic` (Accessed: 13.09.2024)

[23] `https://en.wikipedia.org/wiki/Code_completion` (Accessed: 18.09.2024)

Statistical Code Completion Allamanis et al. (2019) describe the original goal of statical code completion as ranking completion suggestions based on the context rather than simply alphabetically. One of the first works to improve upon this was the Best Matching Neighbours (BMN) algorithm, by Bruch et al. (2009), which uses an adapted k-Nearest Neighbours (kNN) approach to rank the suggestions based on similar usages in the code base. Another approach is to do code completion on the level of n-grams. One example of Hindle et al. (2012), where the authors developed an n-gram-based code completion model for Java, which outperformed the approach that shipped with the Eclipse IDE then.

Code Completion Tools Nowadays, IDEs like IntelliJ IDEA[24] approach the task by analyzing the context and providing suggestions based on the currently written code. The analyzed context here usually includes the current file and other classes within the project and project dependencies.

IntelliJ IDEA, specifically collects completion usage data to suggest better completions based on previous choices. Moreover, it allows the possibility to expand keywords to complete functions automatically. One example is the extension of "*psvm*" to the standard Java main method.

When comparing the traditional code completion in JetBrains IDEs for Java and Python, one can observe that suggestions for Java tend to be more accurate and the list of suggestions tends to be complete[25]. This is intuitive, as Java is a strongly typed language, and developers can therefore receive better suggestions than for Python, where a specific variable's datatype might only be known during execution. One example that illustrates the practical benefit of this is that if one writes "*String s = SomeClass.*" the IDE suggests all static methods of SomeClass that return strings and all visible constants.

Distinction from Code Generation The distinction from code generation can be fuzzy at times. One example is the CoderUJB (Zeng et al., 2024) benchmark, where the authors present a code generation benchmark but call it *coderujbcomplete* in their repository. In this benchmark, the task is to generate complete methods based on the method comment **and** the signature, and the other context information.

Lu et al. (2021) present a relatively straightforward categorization, where they list code generation as a text-to-code task and code completion as a code-to-code

[24] `https://www.jetbrains.com/help/idea/auto-completing-code.html#basic_completion` (Accessed: 09.09.2024)

[25] For an example see `https://medium.com/swlh/static-types-vs-dynamic-types-stop-fighting-and-make-my-life-easier-already-73f58bfe7d0` under "IDE Assistance" (Accessed: 25.12.2024)

task. Combined with the above example, this illustrates the source for ambiguity, namely that the code generation task also utilizes parts from the code as auxiliary information.

Therefore, to avoid confusion in this thesis, we use the following distinction: *Code generation is the task of generating complete source code snippets, primarily, but not necessarily exclusively, based on natural language text or comments. Code completion is the task of completing the current statement, line, block, or method, at least based on the code before the caret position.*

2.4.3. Unit Test Generation

Unit tests are the lowest-granularity tests in a software project. In Java, they are usually used to test individual methods or classes without depending on other components. The standard library for writing unit tests in Java is JUnit[26], which is frequently accompanied by Mockito[27], a library that allows one to mock the behavior of other classes that the current class depends on. This allows developers to circumvent actual implementations, like slow file system or network accesses, of dependencies and specify the class's desired behavior instead. Moreover, Mockito also tracks calls to the mocked object, essentially allowing tests to check how often a mocked method was called.

Automatically Assessing Test Quality While one could just write tests, execute them, and verify that they pass, this does not inform the developer whether they managed to cover all parts of the code or whether their test cases are robust or efficient. A metric called code coverage[28] can be used to assess whether the tests cover all parts. This measure usually indicates the percentage to which all methods, statements, and branches in a method or class are covered. For Java, this can be evaluated using the JaCoCo[29] library. Additionally, mutation testing can be employed to ensure that the test cases are robust. There, small changes, so-called "mutations," are introduced in the code, and it is checked whether the unit tests are able to catch them. One prominent example for Java is PITest[30].

Additional Considerations When testing software, there are not only automatic assessment tools to be considered but also other factors. These include the

[26]https://junit.org/junit5/ (Accessed: 13.08.2024)
[27]https://site.mockito.org/ (Accessed: 13.08.2024)
[28]https://en.wikipedia.org/wiki/Code_coverage (Accessed: 13.08.2024)
[29]https://www.jacoco.org/jacoco/trunk/doc/ (Accessed: 13.08.2024)
[30]https://pitest.org/ (Accessed: 13.08.2024)

time required to execute the test cases and their repeatability, maintainability, and understandability.

Automatic Test Generation Several tools exist to generate unit tests for Java code automatically. Here, we highlight two open-source applications that are also actively maintained: (1) EvoSuite and (2) Randoop. However, one important attribute of such systems to remember is that they generate these test cases based on the code. This means that bugs exisitng in the code can or do also become testing criteria. In the worst case, this can lead to test cases verifying that the bug is still present.

EvoSuite (Fraser and Arcuri, 2013) is an approach to automated test generation to maiximize code coverage while reducing the number of assertions. The authors achieve this behavior by using two key techniques: (1) generating the whole test suite based on the combined code coverage rather than focusing on covering individual elements. And (2) employing mutation testing to minimize the number of assertions while maximizing the coverage of each assertion.

Randoop (Pacheco and Ernst, 2007) works based on "feedback-directed random testing" (Pacheco and Ernst, 2007), which is used to generate test cases. In general, it distinguishes between two types of tests: (1) error-revealing tests that reveal already existing bugs and errors and (2) regression tests that currently pass but might reveal bugs that are introduced in the future. The test case generation is based on so-called contracts, which can be extended to define the desired behavior. The default contracts include, for example, "reflexivity" and "equality."

2.4.4. Method Comment Generation

Accurate and up-to-date comments are essential to keep software understandable and maintainable. In this section, we highlight two approaches for generating method comments/headers for Java before the advent of Transformers and also show some actual tools for documentation generation in Java.

Before LLMs, there were various approaches to generating method comments. One of these is the approach by Sridhara et al. (2010) where they generate Java method comments from the method signature and body. They present the problem as consisting of two steps: (1) selecting the correct content to include in the comment and (2) generating the actual NL comment. They use a Software Word Usage Model (SWUM) (Hill et al., 2009) to preprocess the input data to extract what they call "action," "theme," and "secondary arguments." These are used as input to their generator, which selects the most important parts, generates the NL representation, and subsequently combines them into the method comment. This generator deals with the method in terms of "Sunits," which usually represent individual statements, except for loops or branches, which can consist of multiple "Sunits." They evaluate their approach by letting 13 people with experience in Java programming judge outputs generated by their model. Among other things, they report the majority opinion on various criteria. Each of which have three answer possibilities. While their approach managed to gain primarily good reviews on "Accuracy" (7/8 times the majority for the best answer possible), the same can not be said for "Content Adequacy" (4/8) and "Conciseness" (3/8).

Another approach we want to highlight is Code-NN (Iyer et al., 2016), in which the authors created an LSTM to summarize code snippets in C# and SQL. Iyer et al. (2016) evaluate their model on the dataset they created based on the title of StackOverflow (SO) posts combined with the code snipped from the accepted answer. They subsequently clean the dataset by automatically classifying samples as "clean" or "not clean." The authors do this by utilizing a self-trained classifier. They also create two human-written titles for each sample in their test set to capture a wider variety of answer possibilities. In addition to that, Iyer et al. (2016) compare their model with various baselines on code summarization and code retrieval; however, here, we focus on summarization only. They evaluate the summarization capabilities using BLEU-4 (Papineni et al., 2001) and METEOR (Denkowski and Lavie, 2014) and show that their model outperforms the baseline methods. Moreover, the authors also let five human annotators rank the approach on a scale from 1 to 5 for "Naturalness" and "Informativeness." They showed that it statistically significantly outperformed all but one baseline in all measures except for one.

Java Landscape Java features various ways to write comments: (1) inline comments (//), (2) block comments (/*...*/), and (3) Javadoc comments (/** .... */). In the following, we focus on Javadoc comments (Oracle Corporation), which can be semi-automatically generated and feature the possibility of adding metadata about information like parameters, return values, exceptions, and more. The most prominent use of such documentation is in open-source libraries to provide information on how to use methods and what they do (i.e., API/library documentation). Moreover, this type of documentation is usually also tightly integrated into IDEs, meaning that inspections and code suggestions also benefit from good Javadoc documentation. Another feature is the possibility to generate HTML documentation from these Javadoc comments[31].

[31]See one example at `https://docs.spring.io/spring-framework/docs/current/javadoc-api/` (Accessed: 20.08.2024)

Modern Java IDEs feature the capability to generate parts of the Javadoc automatically. This means that they can prepopulate the comment with metadata tags and inform the user when, for example, a newly created call parameter is not in the Javadoc, or a deleted one still is. However, these tools usually only create the blueprint comment, not the NL description.

One separate tool that partially addresses that shortcoming is JAutodoc [32], a plugin for Eclipse[33] that allows the use of templates to define how to generate the content of Javadoc comments.

2.4.5. Automated Program Repair

Automated Program Repair (APR) is the task of automatically fixing bugs in code.

Tools and approaches to solve this task were already explored a long time before LLMs were introduced, with GenProg (Le Goues et al., 2012), one often-cited[34] approach, being released more than 10 years ago. This approach was released for use with C, utilizes genetic programming[35] (GP) to generate program fixes, and requires test cases to verify the fix. GP is an approach based on generational evolution/mutation and bears similarity to human genetic evolution. Here, the different evolution/mutations are compared using a fitness score computed based on the test case results. The authors of GenProg also apply three major changes to the GP approach, thereby improving the problem of potentially infinite search spaces. These are: (1) setting the granularity at which they operate to statements of ASTs, (2) introducing the assumption that correct code for this problem exists somewhere in the code base, and (3) restricting mutations to AST elements that are actually executed by the failing test case.

Martinez and Monperrus (2016) implemented the Java-specific version jGenProg2 and added the capability to choose the scope of the part of the codebase to be considered when looking for correct implementations. With this paper, the authors also publish further Java-adaptions[36] of other algorithms: jKali as an adaption of Kali (Qi et al., 2015) and jMutRepair as an adaption of MutRepair (Debroy and Wong, 2010).

Another approach we want to highlight is Pattern-based Automatic Repair (PAR) (Kim et al., 2013), where the authors improved upon GenProg by integrat-

[32]See `https://jautodoc.sourceforge.net/` (Accessed: 19.08.2024)

[33]`https://www.eclipse.org/downloads/` (Accessed:19.08.2024)

[34]Cited in 707 papers as of 26.09.2024: `https://ieeexplore.ieee.org/document/6035728`

[35]`https://en.wikipedia.org/wiki/Genetic_programming` (Accessed: 26.09.2024)

[36]`https://github.com/SpoonLabs/astor` (Accessed: 27.09.2024)

ing so-called ”fix patterns.” These patterns were uncovered by analyzing around 62.000 patches written by humans and were subsequently compiled into scripts that define how to fix these errors. The authors also conduct a user study among developers and students and show that, on average, fixes by PAR were considered more acceptable than those by GenProg.

A final work we want to cover in more detail is TBar[37] (Liu et al., 2019). There, the authors approached the topic of APR, more specifically, pattern-based APR, by taking a step back and analyzing existing approaches and their patterns. The authors end up identifying/compiling a ”superset” from various previous works, including PAR. Using this approach, they achieved a new level of performance on Defects4J. They also note that deciding which existing code snippet to use for the fix and having accurate fault localization is important to achieve good performance.

In this section, we primarily showed pattern-based approaches. Even more APR approaches can be found in the paper by Xia et al. (2023) or on a Program Repair Overview site[38].

2.5. Assessing Code Quality

Ensuring high code quality is an important consideration when developing software, as poorly written code can lead to all sorts of issues like unreliable, unmaintainable, or insecure software[39]. One approach that can be used to uncover and, therefore, prevent low-quality code is static analysis. In the following, we highlight three free and commonly used tools for that[40]: (1) PMD, (2) SonarLint and (3) CheckStyle.

PMD[41] is a static analysis tool for multiple programming languages and evaluates code based on rulesets executed against the program’s AST. For Java, PMD comes with a ”quickstart” ruleset that can, and according to their wiki page, should be extended and adapted to specific needs.

[37] https://github.com/TruX-DTF/TBar (Accessed: 27.09.2024)
[38] https://program-repair.org/tools.html (Accessed: 27.09.2024)
[39] https://www.sonarsource.com/learn/code-quality/ (Accessed: 27.09.2024)
[40] A large scale overview of static analysis tools can be found under https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis (Accessed: 27.09.2024)
[41] https://docs.pmd-code.org/pmd-doc-7.6.0/index.html (Accessed: 27.09.2024)

SonarLint[42] is a tool from SonarSource, the creator of SonarCube. It is a local tool that checks against a predefined ruleset, provides explanations and examples for the found rule violations, and suggests fixes. Moreover, it can be paired with SonarCube[43] to synchronize rules across teams. One final feature worth mentioning is the option to ignore existing code in the analysis, allowing the user only to receive feedback on newly written code. This has the advantage of not overwhelming users, for example, when they inherit a poorly maintained project.

CheckStyle[44] is another open source tool that can be configured according to the user's needs. However, they also provide configurations out of the box, such as the "Sun Code Conventions" or the "Google Java Style Guide."

[42]https://www.sonarsource.com/products/sonarlint/ (Accessed: 27.09.2024)
[43]SonarCube is a commercial product from SonarSource and not free!
[44]https://checkstyle.sourceforge.io/index.html (Accessed: 27.09.2024)

3. Related Work

This chapter provides information about related work in benchmarking LLMs for aspects of software development (SD), developer behavior when using LLMs, and implementing software projects with the help of LLMs. Firstly, we start by showing existing benchmarks for the five aspects we use for model comparison (3.1). The second part (3.2) provides a general overview of the approaches that have been used to track and evaluate how programmers utilize LLMs. Finally, we highlight specific works similar to our use case and subsequently clarify the differences (3.3).

Research for the background and related work chapters was primarily conducted using Google Scholar and Connected Papers (CP).

3.1. Model Benchmarking for Aspects of SD

Benchmarks are an essential tool that allows comparing different LLMs to each other and gaining some insight into their performance. In this section, we explore existing benchmarks for five aspects of SD with a special focus on Java. These aspects are code generation, code completion, unit test generation, method comment generation, and automatic program repair (APR).

3.1.1. Code Generation

The task of code generation, or code synthesis, involves generating program code from a natural language description and can be supplemented by additional context information. The natural language description can come from, among others, comments. The output from the LLM can then be evaluated, for example, by comparing it with a reference solution or using unit tests. In the following, we highlight benchmarks and datasets for code generation, moving from the most prominent ones (Python) to Java benchmarks.

There exists a multitude of code-specific benchmarks that can be used to evaluate LLMs. Some of the most prominent examples include HumanEval (Chen et al., 2021) and Mostly Basic Python Programming (MBPP) (Austin et al., 2021). These are centered around method-level code generation. Furthermore, there exist more specific benchmarks like ClassEval (Du et al., 2023), that evaluate an LLMs' ability to synthesize class-level code. What these have in common is, however, that they are all purely Python benchmarks. However, as the primary focus of this thesis is on Java, in the following we, therefore, focus on benchmarks and datasets for said language. It is worth mentioning that most datasets we found were split into train, test, and evaluation parts and are intended to be used to train a model from scratch or fine-tune it before evaluation.

One example of a dataset is CONCODE (Iyer et al., 2018), which was created to train and evaluate the model capabilities of generating methods based on the text part of the JavaDoc method comment and the other methods and fields from a class. Another very specific dataset, is Card2Code (Ling et al., 2016), which contains a task to generate Java classes based on card descriptions from the popular trading card game "Magic The Gathering."

For benchmarks, one example is CoderEval (Yu et al., 2024), which contains both a Python and a Java dataset. The main differentiating factor for this benchmark is using six levels of dependency, ranging from self-contained methods that only utilize code from the *System* class to methods that need the entire project context because they rely on information in Super- or Util-classes.

Moreover, variants of HumanEval, that were translated into multiple languages, also exist. These include HumanEvalPack[1] (Muennighoff et al., 2024) and HumanEval-X[2] (Zheng et al., 2023). Therein, the authors also extended the benchmark to Java (and other languages) by translating the 164 problems.

Another relatively new example is CoderUJB (Zeng et al., 2024), a Java benchmark suite that features multiple aspects and uses Defects4J (Just et al., 2014) as the underlying data and evaluation source. One task that is part of CoderUJB is *codeujbcomplete*, where the model under test is required to generate a method based on the method comment, the import statements, the fields, and the abstract header of the other methods in the class. The benchmark provides prompts for models that were trained on completion as well as for ones trained on instruct (chat).

Additional datasets and benchmarks that also include Java code can be found in López Espejel et al. (2023) and Jiang et al. (2024b).

In this section we detailed datasets and benchmarks ranging from straightforward translations of HumanEval problems to Java to fully automated benchmarks like CoderUJB. Another interesting approach we detailed is CoderEval, which introduces dependency levels that allow for a more fine-granular evaluation of models.

[1] `https://huggingface.co/datasets/bigcode/humanevalpack` (Accessed: 21.08.2024)

[2] `https://huggingface.co/datasets/THUDM/humaneval-x` (Accessed: 14.10.2024

3.1.2. Code Completion

Code completion is the task of completing partially written code snippets. This means generating new code based on already existing code, potentially also including further context information. This section highlights different datasets and benchmarks for Java code completion.

One collection that also contains a dataset to evaluate code completion for Java is CodeXGlue (Lu et al., 2021). There, the authors created datasets for next token prediction and line completion using the GitHub Java Corpus (Allamanis and Sutton, 2013). They evaluated model performance using accuracy for next token prediction and exact match (EM) and edit similarity for line completion. Moreover, they also provided two baselines. One of these is CodeGPT-adapted, a fine-tuned version of GPT-2 that achieved an accuracy of 77.73% for next token prediction[3]. For line-level code completion, the model achieves an EM of 30.6% and an edit similarity of 63.45%[4].

Another approach that includes a Java portion is the CrossCodeEval (Ding et al., 2023) benchmark, in which the task is to complete the current statement. Moreover, the authors provided the model with context information about the current project. They did this by strategically retrieving context from other files within the project. This was done using the retrieve-and-generate approach of Zhang et al. (2023). To select context information based on similarity, they evaluated three different approaches. The two best-performing ones for Java were *text-embedding-ada-002*, a text embedding model from OpenAI, and BM25 (Robertson and Zaragoza, 2009), an extension to ”term frequency - inverse document frequency” (TF-IDF). They obtained their datasets by crawling GitHub repositories over a specific time frame and selecting code snippets requiring cross-file information to be completed correctly. Moreover, CrossCodeEval evaluates model performance in two dimensions: (1) Code Match by computing EM and ES to a reference solution, and (2) Identifier Match, which evaluates whether the correct API calls are used.

A special case of code completion is code infilling - here, code exists before and after the current part that shall be completed (Fried et al., 2023). One benchmark tailored to this task, which also includes Java parts, is Syntax-Aware Fill-in-the-Middle (SAFIM) (Gong et al., 2024). There, the authors collected the data for their benchmark from GitHub and Codeforces (a website for programming contests). Using this data, they created three task types for their benchmark, namely: (1)

[3]https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/CodeCompletion-token (Accessed: 21.09.2024)
[4]https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/CodeCompletion-line (Accessed: 21.09.2024)

Algorithmic Block Completion (from Codeforces), (2) Control-Flow Completion (from Codeforces), and (3) API Function Call Completion (from GitHub). These are evaluated using unit tests, where applicable, and "syntactical matching" in all other cases. Moreover, they also compared multiple prompting approaches with regard to pass@1. There, they found that the Suffix Prefix Middle (for 4/7 models) and Prefix Suffix Middle (for 2/7 models) approach worked best.

In this part, we presented three approaches to evaluating LLM performance for code completion. Moreover, one of these approaches was for infilling, a special case of code completion where code exists before and after the part to be completed.

3.1.3. Unit Test Generation

The task of unit test generation involves synthesizing test cases based on various factors like the method under test, the method comments, or the specifications. This section highlights different datasets and benchmarks specifically for Java unit test generation.

One prominent dataset is SF110 (Fraser and Arcuri, 2014) from the authors of EvoSuite. There, the authors collected a total of about 28,000 classes from 110 Java projects. Moreover, the authors evaluated the performance of EvoSuite on this dataset with a generation time per class of 2 minutes. They achieved an average branch coverage of 71% (Median 94%), with significant variance depending on the "difficulty" of the class under test. Their work specifically highlights multi-threading and environment interaction (e.g., file system, network) as hindrances to higher branch coverage. Siddiq et al. (2024) used this dataset and a version of HumanEval that was translated to Java (Athiwaratkun et al., 2023) to evaluate the ability of various LLMs to generate unit tests. While they achieved above 90% branch coverage with some models on HumanEval, no LLM achieved more than 2% branch coverage on SF110. Unfortunately, the authors do not discuss this difference in scores in detail. However, a sizable performance drop seems logical, considering that HumanEval consists of 160 isolated problems and SF110 originates from code from 111 real repositories. Moreover, Evosuite, which the authors used as a baseline, also exhibited a sizeable drop in benchmark scores from about 95% for HumanEval to around 20-27% for SF110.

In general, SF110 also illustrates that unit test generation can theoretically be evaluated on any set of classes, as branch coverage, a commonly used criterion, requires no external validation information.

Another dataset for unit test generation is Methods2Test[5] (Tufano et al., 2022),

[5] `https://github.com/microsoft/methods2test` (Accessed: 16.08.2024)

where the authors extracted about 780k test cases from 91k Java GitHub repositories; moreover, the authors present their approach to finding a test case's ”focal method” (i.e. the method under test). Additionally, they also explored the use of further ”focal contexts” to extend the model input. This context can, for example, be as small as the individual method under test or as large as the whole class, excluding method bodies of all other methods. They finetuned their model on the various focal contexts and concluded that the largest explored context leads to the best results.

A similar but different approach was pursued by (Watson et al., 2020), where they focused on creating the correct JUnit assertions given a method under test and the test method, excluding the assert statement but including all other statements. Whilst they did not release their dataset, they shared their approach to data generation.

Another dataset was created by Yuan et al. (2024), where they started with a subset of 185 Java projects from CodeSearchNet (Husain et al., 2020) and extracted 1748 focal method test-case pairs. They subsequently sampled 1000 pairs and used them as the benchmark. Interestingly, they not only evaluated the generated unit test but also conducted a user study of 5 participants who had to score the outputs concerning readability and usability. To the best of our knowledge, the dataset is not publicly available.

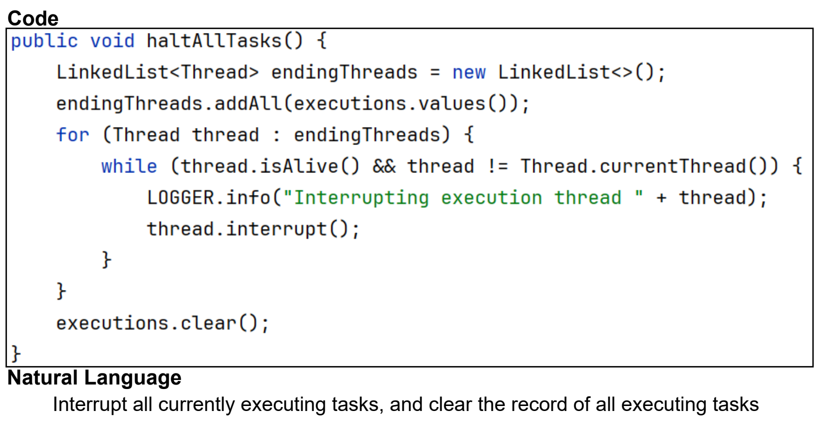
As a final benchmark collection, we want to highlight CoderUJB (Zeng et al., 2024) again, as it also features a benchmark for evaluating unit test generation. The authors again used the Defects4J (Just et al., 2014) dataset and extracted 140 test cases. The generated test cases are subsequently evaluated based on whether they compile and pass. Additionally, the benchmark also analyzes and reports code coverage.

For the aspect of unit test generation, we presented three datasets as well as a benchmark. Moreover, we detailed that, in theory, unit test generation can be evaluated on any code base as coverage criteria can be computed without additional information.

3.1.4. Method Comment Generation

The task of method comment generation is a special case of general program summarization. It involves taking at least the method signature and body and creating a descriptive natural language comment.

This description illustrates that all that is needed to obtain a dataset for this task is to scrape open-source code repositories or other sources for the methods and related comments. One example of such a pair can be seen in Figure 3.1. Approaches that do this include LeClair et al. (2019); Hu et al. (2018a); Liang and


Code

```
public void haltAllTasks() {
    LinkedList<Thread> endingThreads = new LinkedList<>();
    endingThreads.addAll(executions.values());
    for (Thread thread : endingThreads) {
        while (thread.isAlive() && thread != Thread.currentThread()) {
            LOGGER.info("Interrupting execution thread " + thread);
            thread.interrupt();
        }
    }
    executions.clear();
}
```

Natural Language

Interrupt all currently executing tasks, and clear the record of all executing tasks

Figure 3.1.: **Code NL Pair**. An example of a method and the corresponding comment. Taken from the dataset used by Hu et al. (2018a).

Zhu (2018). This is then usually accompanied by subsequent dataset cleaning and preprocessing.

In the following, we want to highlight three different datasets available for download.

The first dataset was initially put forward by Hu et al. (2018a) and subsequently used and mentioned again by almost the same group of authors (Hu et al., 2018b). Here, the authors collected method and comment pairs from 9,714 Java projects on GitHub. Their paper reports that they have released a dataset of 69,708 pairs. However, when manually inspecting the dataset[6] we found significantly more pairs: Train: 470,486, Test: 58,811, Validation: 58,811. These sum up to a total of 588,108 samples. Nonetheless, their first paper (Hu et al., 2018a) evaluated the model performance on BLEU-4 (Papineni et al., 2001). Their second paper used metrics from MT and information retrieval (IR). The MT methods are BLEU-4 and METEOR (Denkowski and Lavie, 2014), and the IR methods are Precision, Recall, and F1-score.

The second dataset (LeClair et al., 2019) we want to highlight was created from 2.1 million comment and method pairs. These originate from the Sourcerer project[7]. The authors specifically selected methods that had Javadoc comments and took the description in the comment as an explanation of the method. Moreover, the authors filter out comments they consider too short or long. They evaluated model performance on various BLEU scores.

[6]https://github.com/xing-hu/DeepCom (Accessed: 20.08.2024)
[7]https://isr.uci.edu/content/sourcerer-project (Accessed: 20.08.2024)

The last dataset we want to highlight is HumanEvalExplain (Muennighoff et al., 2024), which is part of the HumanEvalPack[8]. It fundamentally differs from the approaches above in that it does not aim to evaluate the code's explanation directly. Instead, it synthesizes code based on the generated comment and assesses the pass@k score. One paper that uses HumanEvalExplain is Szalontai et al. (2024).

This subsection highlighted three datasets for evaluating LLMs' ability to synthesize method comments. One noteworthy approach we presented was HumanEvalExplain, where the authors generated code from the previously generated comment and evaluated it using a test suite.

3.1.5. Automated Program Repair

Automated program repair (APR) is the task of automatically fixing a code snippet. This can range from complete methods to single lines (Xia et al., 2023) . Multiple Java benchmarks exist for this task, four of which we highlight in chronological order by their publication.

The first benchmark we want to highlight is QuixBugs[9] by Lin et al. (2017), released before the advent of LLMs. The benchmark consists of problems in Python and Java, each containing one buggy line that must be fixed. Said fixes are then evaluated using test cases. The problems have their origin in the Quixey challenge[10] and were translated to both Java and Python - to allow a direct performance comparison between different languages.

The second benchmark we want to highlight is the Bears Benchmark (Madeiral et al., 2019), created for Java. There, the authors introduced an approach to automatically generating benchmark data from GitHub by leveraging Continuous Integration (CI) build results to detect buggy builds and to identify subsequent builds that fix that bug. This is determined by checking the results of test suite runs. Moreover, they specifically focused on Travis CI as a build platform, as according to them, it is "tightly integrated" into GitHub. Initially, their benchmark contained 251 bugs; however, as of September 2023, this number has reduced to 118 due to various issues (see GitHub[11]), including the unavailability of dependencies required to reproduce the bugs.

Moreover, there also exist newer benchmarks. One of which is GitBug-

[8] https://huggingface.co/datasets/bigcode/humanevalpack (Accessed: 20.08.2024)

[9] https://github.com/jkoppel/QuixBugs (Accessed: 24.09.2024)

[10] https://en.wikipedia.org/wiki/Quixey#Quixey_Challenge (Accessed: 24.09.2024)

[11] https://github.com/bears-bugs/bears-benchmark?tab=readme-ov-file#data (Accessed: 25.09.2024)

Java[12] (Silva et al., 2024), which was released earlier this year and focuses on bug recency and reproducibility. The authors addressed these requirements by collecting bugs and their fixes from commits in 55 repositories. All commits were made in 2023. To achieve this, they used their tool called GitBug-Actions[13] (Saavedra et al., 2024). This ensures that the environment stays reproducible by collecting all data required to run the benchmark and creating a self-contained docker image.

The last benchmark we want to mention here is CoderUJB (Zeng et al., 2024). This is a benchmark suite designed around the Defects4 (Just et al., 2014) dataset, which itself is a collection of bugs that are also reproducible. CoderUJB extends on this by adding a software suite to prompt LLMs and evaluate their output automatically. To do this, they automatically generate the prompts for the LLMs, ultimately simplifying the evaluation process.

For APR, we highlighted four benchmarks. We further described that the Bears benchmark was reduced to a smaller size of problems due to no longer existing dependencies. GitBug-Java addressed this issue by creating self-contained docker images.

3.2. Developer Behavior and LLMs

To gain insights into how developers use LLMs and what their experience is, it is necessary to understand how data can realistically be obtained in the first place. Therefore, this section focuses on different approaches to collecting data when programming and interacting with LLMs.

There are multiple possibilities for obtaining behavior and usage data. One straightforward approach several publications take is to use screen recording, which can subsequently be used to analyze and retrace user behavior (Asare et al., 2024; Tang et al., 2024a; Vaithilingam et al., 2022; Barke et al., 2023). In addition to that, Vaithilingam et al. (2022) and Barke et al. (2023) both record the user audio, with the latter explicitly instructing users to "talk through their interactions."

Another approach to collecting data on how tools, like GitHub Copilot, are being used is to utilize metadata from the tool itself. This approach was presented in papers from Microsoft (Mozannar et al., 2024) and GitHub (Ziegler et al., 2022). The authors used GitHub Copilot to track detailed usage data related to their research. However, this capability to collect data in this detail does not seem to be available to people outside of these companies, with GitHub only providing an API[14] to access

[12]https://github.com/gitbugactions/gitbug-java (Accessed: 25.09.2024)
[13]https://github.com/gitbugactions/gitbugactions (Accessed: 25.09.2024)
[14]https://docs.github.com/en/rest/copilot/copilot-usage?apiVersion=2022-11-28 (Accessed: 16.10.2024)

more general statistics related to Copilot use within teams and organizations.

One approach that combines both metadata and user tracking was pursued by Tang et al. (2024a), where they used the CodeGRITS [15] toolkit (Tang et al., 2024b). This toolkit can record the screen, track "eye gaze," and collect JetBrains IDE usage information simultaneously. The types of IDE usage data that the tool collects can be seen in Figure 3.2. Moreover, Tang et al. (2024a) also collected information about the experience of using LLMs by utilizing self-reporting and semi-structured interviews.

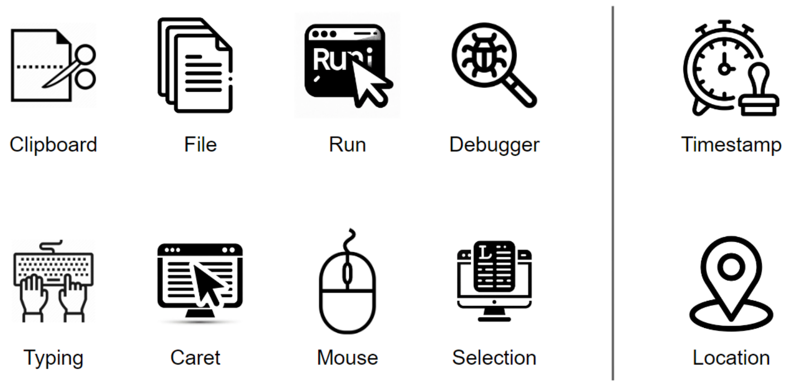


Figure 3.2.: **CodeGRITS IDE Tracker**. An overview of the different actions the CodeGRITS IDE tracker collects in JetBrains IDEs. Taken from `https://codegrits.github.io/CodeGRITS/usage-guide/` (Accessed: 16.10.2024)

In general, most papers we found also use surveys that are then tailored to the specific areas of interest for the research work (Tang et al., 2024a; Liang et al., 2024; Vaithilingam et al., 2022; Asare et al., 2024; Ross et al., 2023; Ziegler et al., 2022). However, some used semi-structured interviews to gain insights into the experience of using LLMs (Barke et al., 2023; Tang et al., 2024a).

In this section, we described multiple approaches to tracking developer behavior when using LLMs, which ranged from recording the screen and audio, over collecting IDE interactions to surveys and semi-structured interviews.

[15] `https://github.com/codegrits/CodeGRITS` (Accessed: 16.10.2024)

3.3. LLM-aided development of a complete Software Project

Few papers explore the use of LLMs to develop a software project. In the following, we highlight four similar papers and their different approaches. This is followed by an explanation of what differentiates our approach (see Subsection 3.3.1).

One approach similar to ours is Peng et al. (2023), where the authors conducted an experiment with two groups of developers. Both were tasked with implementing a very rudimentary HTTP web server in JavaScript as fast as possible. One group was allowed to use GitHub Copilot, while the other was not. The experiment primarily focused on productivity, measured in task completion time. They observed a 55,8% faster completion time for developers that used Copilot. Moreover, participants were also asked to complete a survey to gain further insights into which group benefited most from the coding assistant. They found that Copilot was most beneficial to developers with much experience, developers older than 25 years, and developers who spend a large amount of time writing code.

A different approach was taken by Rasnayaka et al. (2024) as they conducted an experiment with multiple student groups during a group project for a university course. There, the groups had to develop a C++ software project, and they were encouraged to use LLMs to aid with development efforts. However, the students had to annotate the code that was AI generated with the model name and indicate how much of the original output was retained (all, $> 90\%$, $< 90\%$). They found that LLM usage (#prompts) decreased throughout the project whilst the complexity of the synthesized code increased. The authors also highlight that students with more coding experience tended to use LLMs more extensively than students with less proficiency. They attribute this to the requirement of essentially auditing the synthesized code before adding it to the project. Moreover, they did not observe any statistically significant implications on "correctness and quality" based on the amount of LLM code in the project.

Heitz et al. (2024) took a different approach to evaluating the code generation capabilities of the free Versions of ChatGPT (GPT-3.5) and Gemini. In the first part of their evaluation, they evaluated them on HumanEval (Chen et al., 2021) and ClassEval (Du et al., 2023), both Python benchmarks. Here, ChatGPT outperformed Gemini; however, the authors were not completely satisfied with either model due to their tendency to create semantic errors. The second part consisted of multiple developers being tasked with implementing a small Java application (330 Lines of Code (LoC)) using either one of the models. The primary focus here was to assess the productivity improvement and code quality. The authors report that the developers had to refine their prompts often iteratively to ensure that the model completely understood the task details. Moreover, they note that the code generated by these models showed minor issues and code smells.

The final approach we are presenting here is the one from Monteiro et al. (2023), where they explored the use of GPT-4 in developing a simple web-based forum. Three developers were tasked with individually implementing the six short user stories, and their results were then compared to a manually implemented reference solution. Their tech stack consisted of Vue.js (Frontend), TypeScript (Backend), and an SQLite database. Their work mainly focused on prompting, with a special focus on bottom-up and top-down prompting. While the authors only had a limited set of observations, they found the bottom-up approach to require fewer prompts overall, especially fewer bug fixing steps. However, the model seemed to struggle more with correctly integrating features in the code base.

3.3.1. What differentiates our approach from others (Research Gap)

Here, we explain what sets our approach apart from the previous works mentioned above. The scenario we explored is defined in Chapter 4, and the exploration is detailed in Chapter 7.

The first main difference in our approach is the model selection process, which started with a selection of models that would theoretically fit the requirements of our practical scenario. This was then further reduced to a single model by assessing and comparing their performance on five different aspects of software development. Our approach contrasts the others mentioned above, which used a single or two models for their work. The closest paper to our approach in this regard is Heitz et al. (2024), where they also compared two models. However, they evaluated the models on Python benchmarks before using them in a Java task. Moreover, they continued to use both models for the implementation.

Another key difference is our practical exploration (see Chapter 4). Our approach differs from Monteiro et al. (2023), which also implemented a project based on user stories, by using a larger number of user stories, and by using two different languages instead of two variations of JavaScript. One we were familiar with and one that we were not. Similarly to the existing literature, we also considered the code quality of the resulting project. However, we did note our observations and aimed to fix them during the implementation so as not to cause any unnecessary hindrances. Moreover, we also considered prompting approaches as a means to an end rather than as a primary focus.

To the best of our knowledge, our approach bears some similarities yet is distinctively different from the existing approaches.

3.4. Research Information

Research on this topic was conducted between August and mid-October 2024. As LLMs for code is a highly active research topic, we decided to use a research cutoff after this research phase. Papers released around this time or later could be missing as they might not have been indexed by the tools used (Google Scholar, Connected Papers). ConnectedPapers (Eitan et al., 2024) was used in the research process to gain a better overview and reduce the chance of overlooking papers in this vast and fast-moving research field.

4. Practical Scenario - Definition

Before selecting and comparing LLMs for software development (SD), we first define a scenario, the realization of which is detailed in Chapter 7. It serves as a practical example, providing constraints and expectations to consider.

The scenario we explore involves a one-person startup with limited funds that aims to develop a web application for expense tracking with a separate frontend and backend. Some of the application requirements, including simplified user stories, user interface (UI) mockups, and the data model, have already been defined upfront.

In the following, we explore the practical scenario based on various viewpoints. The exploration of this is detailed in Chapter 7.

4.1. Company

The company is a one-person startup with a minimal budget to be used on AI tools for SD. Therefore, the company heavily favors tools and solutions that do not require significant one time investments. Such offerings are subscriptions or pay-as-you-go models. Crucially, this means that hosting or running large models locally is out of question due to the high upfront cost for specialized hardware. Additionally, (very) small models may be considered, as they can realistically run on a developer machine without significantly impacting system performance.

4.2. Tech Stack Constraints

The application we explore needs to consist of a separate backend and frontend to future-proof the application by separating most of the logic from the visual representation.

Moreover, the backend is implemented in Java as the developer is most familiar with that language. More specifically, the Spring Boot framework[1] is used to simplify the implementation of REST APIs, authentication, and data persistence. In

[1] https://spring.io/ (Accessed: 23.10.2024)

this project, the data is persisted in a PostgreSQL[2] database.

The frontend is built using React[3], more specifically, the Next.js[4] framework. For faster UI development, shadcn[5], a collection of blueprint UI components, is being used. While shadcn is built on top of RadixUI[6], a component library, it differs from classic component libraries as it gives the developer full access to how the components are implemented. This is done by adding them as files to the project. The developer has no relevant prior experience using React or Next.js but has previously worked with typescript (TS), the underlying language.

4.3. Application Constraints

The application has to be developed based on functional requirements, presented as a data model, simplified user stories, and UI mockups.

Figure 4.1 shows the data model that was defined before the implementation of the scenario. It is based on the user stories and does not include relationships between the entities.

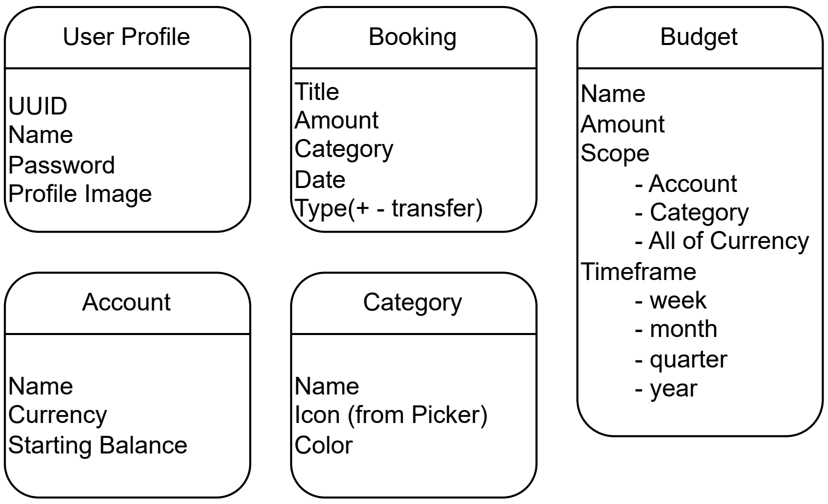


Figure 4.1.: **Data Model** for the different entities in the application.

[2]https://www.postgresql.org/ (Accessed: 23.10.2024)
[3]https://react.dev/ (Accessed: 23.10.2024)
[4]https://nextjs.org/ (Accessed: 23.10.2024)
[5]https://ui.shadcn.com/ (Accessed: 23.10.2024)
[6]https://www.radix-ui.com/ (Accessed: 27.11.2024)

Individual features for the application are given in the form of simplified user stories[7]. Simplified means that the benefit is omitted. Table 4.1 shows the user stories for the account entities; a complete list of user stories is provided in Appendix A.2.

Accounts	
As a user, I want to be able to...	create a default account on my first log-in
	always have a default account
	create additional accounts
	change default accounts
	delete all of my accounts except the default one
	edit all my accounts
	see the current balance of my accounts
	see the account delta for the current month

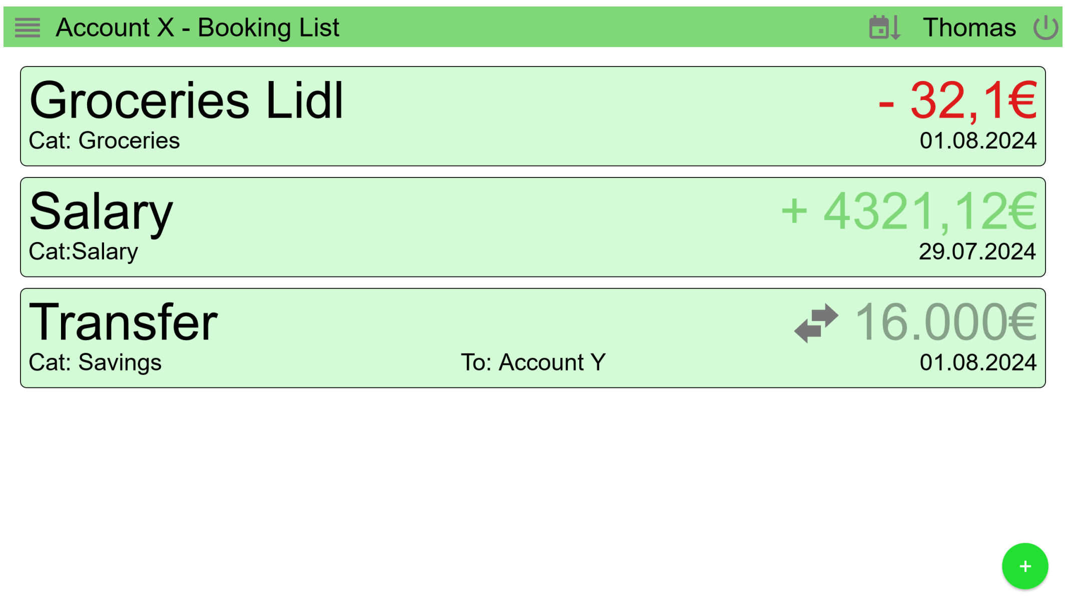
Table 4.1.: **Account User Stories**, detailing the account requirements for the practical exploration.

In addition to the data model and user stories, UI mockups are provided. These serve as a coarse guideline on how the UI should look like, allowing the developer to focus on functionality and details. One UI mockup can be seen in Figure 4.2, and a complete list of mockups is available in Appendix A.3.

4.4. Further Remarks on Tech Stack Considerations

Initially, the idea was to develop the entire application, including the frontend, in Java (using, e.g., Swing, JavaFX or Thymeleaf), as the main focus of this work is evaluating LLM tools for the Java SD process. However, we ultimately decided against it as this is no longer a frequently used approach in practice. Instead, we chose to split the application into a Next.js/React frontend and a Java/Spring Boot backend. Given the fact that the developer did not have any prior experience with React or Next.js, this decision provided us with the opportunity to evaluate the effect of using a language/framework, the developer was not familiar with, prior to this project. Furthermore, we decided to add shadcn, a relatively new and different UI library[8] to see how the tool would fare.

[7]https://en.wikipedia.org/wiki/User_story (Accessed: 23.10.2024)
[8]First GitHub release: 08.03.2023


Account X - Booking List

Thomas

Groceries Lidl

- 32,1€

Cat: Groceries

01.08.2024

Salary

+ 4321,12€

Cat:Salary

29.07.2024

Transfer

16.000€

Cat: Savings

To: Account Y

01.08.2024

Figure 4.2.: **UI Mockup** of the bookings overview page for an individual account showing the three different booking types. Moreover, the ability to sort is indicated in the header.

5. Initial LLM Selection

Considering the quantitative comparison of the models and the following practical exploration, we selected an initial set of LLMs to be evaluated. Section 5.1 details the constraints that needed to be considered, and Section 5.2 provides more detailed information on the models that we selected for the comparison. Section 5.3 presents the rationale for choosing the individual models and answers RQ1.

5.1. Method

In this section, we highlight the constraints of our practical exploration and how they influenced the selection of models that could be considered.

5.1.1. Practical Exploration Constraints

Due to the constraints for the practical exploration in Chapter 4, we had specific requirements for the models we could consider. The following section highlights the most important requirements and explains their implications.

In Chapter 4, we established that we would need a model that could be run locally on a laptop[1], or one that is hosted by an external party. This explicitly excluded setting up "large" LLMs on local or cloud infrastructure ourselves. Considering local models did require us to ensure that the model execution would not interfere with the performance of other software that runs on that computer, like an IDE. Moreover, including hosted offerings enabled us to use large LLMs, like Claude 3.5 Sonnet, which are usually billed monthly using a flat fee or usage-based. Although it would also have been possible to set up arbitrary LLMs in a Cloud, we considered this out of scope. We argue that assuming a typical developer would go this extra step for a one-person startup is not logical.

Another constraint originating from our practical scenario was that the training dataset for the LLM should also have contained code - ideally, Java and, to a lesser extent JavaScript / Typescript. However, this requirement was not strictly enforced for the small models, allowing us to include Phi 3.

[1]The following specification forms the upper limit for the hardware that can be expected: AMD Ryzen 5800H, 16GB DDR4 3200 MT/s, NVIDIA GeForce RTX 3060 Laptop GPU

Moreover, since we did not need to adhere to strict privacy requirements regarding code confidentiality, we could even consider offerings that might use model outputs for further training.

A final point that we considered beneficial is an integration into the IDE via tools such as GitHub Copilot, Replit, or JetBrains AI. However, if a model that had not been integrated would have performed significantly better, we would have selected it over an already integrated one.

5.1.2. Downstream Task Constraints

As we wanted to ideally complete the practical exploration using a single model, the selected models needed to be evaluated and compared. We did this using five benchmarks (see Chapter 6). However, to run the benchmarks, there had to be a way to automatically interact with the models, either through an API or by running the model locally. Moreover, to achieve a fair comparison, we had to compare the models with each other without potentially surrounding tools. To illustrate this, if we wanted to evaluate GitHub Copilot, we needed to assess GPT-4(o)[2] to keep it comparable. Furthermore, considering the primary mode of interaction in both the model comparison and practical exploration is chat-based, the models should have been optimized for instructions (i.e., instruct). The only exception to this is code completion in the practical exploration. There it would be best if the had been optimized for completions.

5.2. Results

Based on the constraints we derived from our practical scenario we selected eight out of 11 models (see Table 5.1) to compare. All of the selected models were optimized for chat-based interactions (i.e., instruct models). Furthermore, for most models we found it difficult to verify which exact programming languages they were trained on. However, considering that both Java and Typescript/JavaScript are popular languages[3] it is highly likely that they were included in the training datasets.

In the following, we provide further details on the individual models:

The **GPT-4o** model is the current ”flagship model” from OpenAI[4], which was

[2]Please note that according to OpenAI, Copilot is no longer powered by Codex but some non-disclosed version of GPT-4 (Source: support chat with OpenAI - Screenshots available on request)

[3]https://survey.stackoverflow.co/2024/technology#most-popular-technologies-language (Accessed: 22.01.2025)

[4]https://platform.openai.com/docs/models/gpt-4o (Accessed: 28.10.2024)

Model	**Context Window [tokens]**	**How Provided?**	**Selected**
GPT-4o (2024.08.06)	128k	Hosted	✓
Claude-3.5 Sonnet (20240620)	200k	Hosted	✓
Gemini 1.5 Pro	2M	Hosted	✓
Gemini 1.5 Flash	1M	Hosted	✓
Qwen 2 0.5B Instruct	32k	Local Runtime	✓
Qwen 2 1.5B Instruct	32k	Local Runtime	✓
Deep Seek Coder V2	-	Hosted	✗
Deep Seek Chat V2	-	Hosted	✗
Deep Seek V2.5	128k	Hosted	✓
Phi 3 3.8B instruct	128k	Local Runtime	✓
Replit Code V1.5 3B	4k	Local Runtime	✗

Table 5.1.: A list of initially considered models, including those we eliminated before running the comparison. The selected models are highlighted in green, and the eliminated ones are in red. Dashes indicate that no verifiable information could be found. We used Ollama as a local runtime for the models.

trained on data, including code (OpenAI, 2024), ranging up until October 2023[5]. It is a proprietary model that features a context window size of 128k tokens, with the maximum number of output tokens being limited to 16k. The price[6] for this model via the API is 2.50$ per 1 million input tokens and 10$/1M output tokens. The specific model version we selected is ”gpt-4o-2024-08-06”, the most recent version at that time.

Claude 3.5 Sonnet[7] is the largest model from Anthropic AI and features a cutoff date for training data of April 2024. While Anthropic does not explicitly mention that the model was trained on code, their blog post highlights working with code as one of the areas where the model performs well[8]. The size of the context window is

[5]https://learn.microsoft.com/en-us/azure/ai-services/openai/concepts/models?tabs=global-standard%2Cstandard-chat-completions (Accessed: 21.01.2025)
[6]https://openai.com/api/pricing/ (Accessed: 28.10.2024)
[7]https://docs.anthropic.com/en/docs/about-claude/models#model-comparison-table (Accessed: 28.10.2024)
[8]https://www.anthropic.com/news/claude-3-5-sonnet (Accessed: 22.01.2025)

200k tokens and the maximum number of tokens to be generated is 8k. The usage of this model costs 3$/1M input tokens and 15$/1M output tokens. The model version we used is ”claude-3-5-sonnet-20240620”.

Gemini 1.5 Flash and **Gemini 1.5 Pro** are proprietary models from Google with a knowledge up until May 2024[9] and have been trained on code (Team et al., 2024b). The Flash variant features a context window size of 1M tokens and costs[10] up to 0.15$/1M input tokens and up to 0.60$/1M output tokens. The Pro variant has double the context window size (2M) and costs[11] up to 2.50$/1M input tokens and up to 5.00$/1M output tokens. The versions we used are ”gemini-1.5-flash-001” and ”gemini-1.5-pro-001”, which, up until the end of our comparison (on 07.10.2024), were called ”gemini-1.5-flash” and ”gemini-1.5-pro” respectively.

Qwen2 0.5B and **Qwen2 1.5B** are open models from Alibaba Cloud which were also trained on code. Both feature a context window size of 32K tokens, and have an undisclosed knowledge cutoff date. Because they are small models, they can be run locally using a tool like Ollama[12]. The specific variants we used are identified as ”qwen2:0.5b-instruct” and ”qwen2:1.5b-instruct” within Ollama.

DeepSeek V2.5 is an open model from DeepSeek that is the result of ”merging”[13] DeepSeek V2 Chat and DeepSeek Coder V2, which are both based on ”DeepSeek-Coder-V2-Base”[14]. It features a 128k token context window and can generate up to 4k tokens as output. Using this model costs 0.14$/1M input tokens and 0.28$/1M output tokens. Initially, we aimed to evaluate both v2 models; however, to the best of our knowledge, these are no longer provided by their API, and both model specifiers, ”deepseek-coder” and ”deepseek-chat,” now refer to the new model. Furthermore the base model was trained on data ranging up until November 2023. The training dataset for this model was comprised of 60% code containing 338 programming languages (DeepSeek-AI et al., 2024b).

Phi 3 mini is an open model, created by Microsoft Research and includes variants with context lengths of 4k and 128k tokens. It was trained on data ranging up until October 2023[15]. The variant we used is run via Ollama and is called ”phi3:3.8b-instruct”[16], which is a quantized (Q4_0) version with a 128k token context window.

[9] https://cloud.google.com/vertex-ai/generative-ai/docs/learn/models (Accessed: 21.01.2025)
[10] https://ai.google.dev/pricing#1_5flash (Accessed:28.10.2024)
[11] https://ai.google.dev/pricing#1_5pro (Accessed: 28.10.2024)
[12] https://ollama.com (Accessed: 28.10.2024)
[13] https://api-docs.deepseek.com/quick_start/pricing/ (Accessed: 28.10.2024)
[14] https://api-docs.deepseek.com/news/news0905 (Accessed: 21.01.2025)
[15] https://huggingface.co/microsoft/Phi-3-mini-128k-instruct#model (Accessed: 21.01.2025)
[16] https://ollama.com/library/phi3:3.8b-instruct (Accessed: 28.10.2024)

It has to be noted that code in the training data was primarily comprised of python code that uses basic packages[17].

We initially considered but ultimately did not use the **Replit Code V1.5 3b** model as we could not verify whether and to what extent Replit actually uses it in production.

5.3. Discussion

Here, we discuss the rationale for choosing the specific models for our comparison.

GPT-4o We selected a GPT-4 model because it is used in GitHub Copilot and JetBrains AI and is provided as a premium model in Replit. Moreover, we specifically selected the **GPT-4o-2024-08-06** snapshot as it was the most recent and cheapest full-size GPT-4(o) variant at that time and since we were not able to verify which specific models are used by tools like GitHub Copilot or Replit.

Claude 3.5 Sonnet This model by Anthropic was selected as it is a competing offering to GPT-4. Moreover, it is also part of the premium models in Replit. We specifically selected **claude-3-5-sonnet-20240620** as this was the most recent version of Claude 3.5 at the time of evaluation.

Gemini 1.5 We selected two Gemini models, Gemini 1.5 Flash and Gemini 1.5 Pro. The Flash variant was chosen as it is the freely available model on Replit, and the Pro variant was selected to have a ”reference” offering similar to Claude and GPT.

Qwen 2 The 0.5B and 1.5B instruct models were selected as they are both very small and should be able to run on a developer machine without interfering with other tasks.

Deep Seek V2.5 Initially, we wanted to evaluate both Deep Seek Coder and Deep Seek Chat. The primary motivation was that the model is open and available on HuggingFace, their webpage, and via an API. Moreover, we aimed to evaluate the difference between a model trained explicitly on code and a more general one. However, on 05.09.2024, they released Deep Seek V2.5[18], combining the two models

[17] https://huggingface.co/microsoft/Phi-3-mini-128k-instruct#responsible-ai-considerations (Accessed: 27.11.2024)

[18] https://platform.deepseek.com/api-docs/updates/ (Accessed: 16.09.2024)

into one and making the other two models unavailable in their API. Therefore, we ultimately decided to use **Deep Seek V2.5**.

Phi 3 We selected the 3.8b instruct version to add an additional step in model sizes between the small Qwen 2 models and the large models. Moreover, we wanted to include a local model that requires more computational power than the small Qwen 2 variants.

In this section, we detailed the initial model selection process and can, therefore, now answer the first research question:

Answer for RQ1: What aspects of an LLM are required to be useful for both our practical exploration and the comparison?

For the models to useful to both the exploration and the comparison, they should have been trained on code, with the training dataset including Java and JavaScript/Typescript. Furthermore, ideally they would also be integrated directly into an IDE.
For the quantitative comparison based on benchmarks, there needs to be a means of directly and automatically interacting with the models (i.e.. an API). Moreover, the model needs to either be small enough to run as an additional application on the machine or be available on a subscription or pay-as-you-go basis.
Furthermore, the models should also have been optimized for instructions rather than completions as the evaluation of all five aspects in the model comparison is based on prompts. Additionally, in the practical exploration all interactions with the exception of code completion are chat-based.

6. Model Comparison

This chapter details how we compared the models and how they performed. To compare the eight LLMs, we evaluated them on five different aspects of Java SD. Please note that while we later decided to write the frontend part in the exploration using React for multiple reasons, the original idea was to implement the entire application using Java (see Section 4.4). Therefore, this chapter only focuses on aspects of Java SD.

We selected each aspect based on practical experience with both Java SD and LLMs. Each aspect was evaluated with either a preexisting benchmark or a new evaluation based on an existing dataset. The five evaluations are detailed in Sections 6.1 to 6.5.

However, we needed not only to compare the LLMs but also to determine which model we should use for our practical evaluation. This ranking process is given in Section 6.6.

6.1. Code Generation

Code generation is the task of synthesizing code based on a description. In this comparison, we focused on generating individual Java methods from comments. We selected this code generation aspect as datasets and benchmarks are readily available. In the following, we elaborate on the benchmark we used, the observed results and the conclusions we could draw from them.

6.1.1. Method

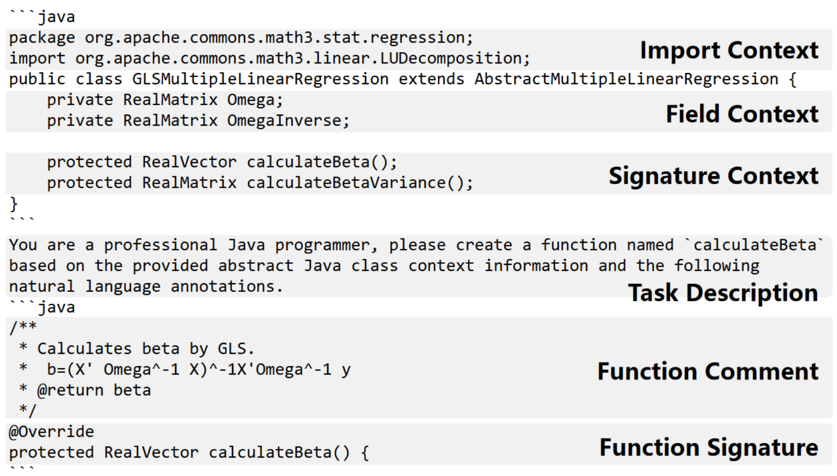
To evaluate models on their ability to synthesize code, we used the **CoderUJB**[1] benchmark (Zeng et al., 2024), specifically **"codeujbcomplete."** The task in this benchmark is to generate a method body based on the method comment, the method signature, and additional context information. The individual tasks in this benchmark were derived from Defects4J (Just et al., 2014). It also provides the required environments and test cases to evaluate the generated code.

[1]See `https://github.com/WisdomShell/ujb` (Accessed: 07.10.2024)

There are 238 tasks in "codeujbcomplete," which are evaluated in four stages. A prerequisite for passing a stage is that the previous stage must have been passed. In the first stage ("pass_syntax"), the method is inserted into the surrounding class, and the syntax is verified using a parser. The second stage ("pass_compile") consists of an attempt to compile the project using Defects4J in the background. If this fails, the first compilation error is reported, which we categorize (see 6.1.1) and report. The third stage ("pass_trigger") is a remnant from Defects4J and not relevant here, as it only executes part of the test cases. Instead, the important stage for code generation is "pass_all," which executes all available test cases.

We executed the benchmark according to the instructions provided in the project readme file and adapted the code-base to support all our models. Regarding default values, we kept the default temperature of 0.2 and the maximum number of newly generated tokens at 1024. We had initially set the number of samples per task to five. However, since we did observe little to no difference between the individual outputs for each task, we ultimately decided to use a single sample instead.

The performance of the models is reported as **pass@1**, computed as the average of the pass@1 scores of each task. As we generated one candidate per task, the pass@1 score of an individual task is a binary measure. Due to the nature of the results, they can also be interpreted as passing percentages.


Import Context
Field Context
Signature Context
Task Description
Function Comment
Function Signature

~~~
```java
package org.apache.commons.math3.stat.regression;
import org.apache.commons.math3.linear.LUDecomposition;
public class GLSMultipleLinearRegression extends AbstractMultipleLinearRegression {
    private RealMatrix Omega;
    private RealMatrix OmegaInverse;

    protected RealVector calculateBeta();
    protected RealMatrix calculateBetaVariance();
}
```
You are a professional Java programmer, please create a function named `calculateBeta`
based on the provided abstract Java class context information and the following
natural language annotations.
```java
/**
 * Calculates beta by GLS.
 *  b=(X' Omega^-1 X)^-1X'Omega^-1 y
 * @return beta
 */
@Override
protected RealVector calculateBeta() {
```
~~~

Figure 6.1.: **Code Generation Prompt**. A prompt example for code generation with CoderUJB. Taken from Zeng et al. (2024).

Moreover, this benchmark provides a prompt generation strategy (see Figure 6.1). The generated prompt includes the method comment and the method header for the body that has to be generated. Furthermore, the prompt also contains a more general ”Task Description” that instructs the model on what to do. Additionally, each prompt includes the import statements of the file and the fields and abstract method signatures from the class.

Error Categories

In addition to the output already provided by the benchmark, we collected the compilation error messages during our evaluation. We grouped them based on categories introduced in a document from 2007 by the University of Princeton[2]. Since Java has evolved since then and the document is not exhaustive, we also went through the uncategorized errors raised during the evaluation. We created rules to assign them to a preexisting category or create a new one. The categories and a short explanation are given in Table 6.1.

Error Category	**Description**
abstract	Issues related to the use of the abstract keyword/feature in Java.
computation	Among others: Issues related to type conversion, operations with and on data, uninitialized variables
exception declaration	Issues related to exceptions in Java.
identifier	Among others: Issues with undefined or already defined symbols, unimplemented method bodies, access modifiers
return statements	Issues with return statements: unreachable or missing
static	Issues related to the use of the static keyword/feature in Java.
syntax	More complex syntax errors that the parser missed
version	A feature from a newer version has been used in the code. Example: Defects4J has projects that still use Java 1.4. Therefore, they do not support for-each statements or similar.
timeout	When the compilation and the pass_syntax evaluation together took more than 120 seconds.

Table 6.1.: The eight error compile error categories we settled on.

It is essential to mention that the compilation step in this benchmark is done using Defects4j, and it only reports the error that caused the compilation to fail. This means that more than one error could have been made in a task, but only the

[2]https://introcs.cs.princeton.edu/java/11cheatsheet/errors.pdf (Accessed; 02.11.2024)

first one would have been reported.

We chose this specific benchmark since it offers two desirable properties: (1) An evaluation based on **executed code** by using unit test cases and (2) **additional context information** about the class that the method is a part of.

6.1.2. Results

Figure 6.2 and Table 6.2 show the performance of all eight models as pass@1 scores. These were computed as the mean of the *binary* pass@1 values from the 238 individual tasks. Since the pass@1 scores of the individual tasks are binary, the resulting combined values can also be interpreted as percentage values, which we do in the following.

Model	**pass_syntax**	**pass_compile**	**pass_all**
	[pass@1]	**[pass@1]**	**[pass@1]**
Claude	97.5	69.3	39.5
GPT-4o	97.5	61.8	30.3
DeepSeek V2.5	96.6	59.2	29.0
Gemini Pro	86.1	47.1	23.1
Gemini Flash	92.0	45.8	19.7
Qwen2 1.5B	78.6	21.4	3.4
Phi3 3.8B	22.3	2.9	0.0
Qwen2 0.5B	37.4	6.3	0.0

Table 6.2.: Pass rates for all relevant stages of ”codeujbcomplete” with a precision of 1.

The results show that the five larger models managed to pass more individual tasks than the three smaller models, with Claude 3.5 Sonnet performing the best. The larger models passed all test cases on 19.7% to 39.5% of tasks. For the smaller models, Qwen2 1.5B was able to pass 3.5% of test cases, with the other two passing none. Moreover, the data shows that all models have failed tasks due to incorrect syntax (see *pass_syntax*) or compilation issues (see *pass_compile*). While the percentage that passed the syntax and compilation step is between 45,8% and 69,3% for the larger models, the same cannot be said for the smaller ones. They could only produce compiling code for 2.9-21.4% of tasks.

Table 6.2 also shows that Phi3 3.8B consistently achieved lower pass@1 scores than the smaller Qwen2 models. Moreover, it can be observed that Qwen2 1.5B produced syntactically correct code in 78.6% of instances, which is close to the

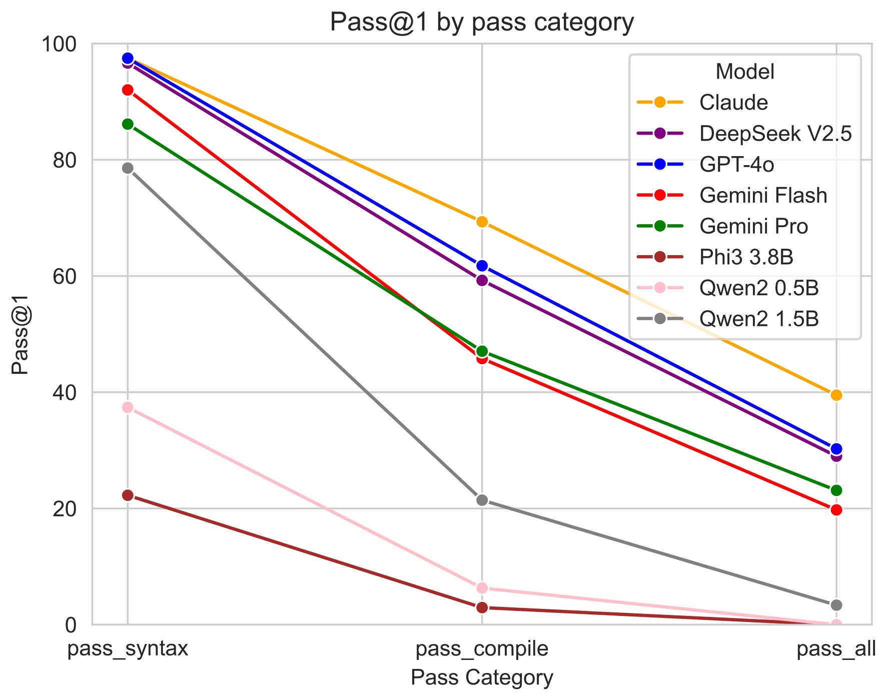


Figure 6.2.: **Code Generation Results** are given in pass@1 for each of the eight models over all three criteria. ”pass_syntax” denotes that the output passed a syntax check, with ”pass_compile” denoting that the code could also be compiled. ”pass_all” denotes that the synthesized code additionally passed all test cases. Results for ”pass_trigger” omitted, as not relevant for this benchmark.

larger models. However, it failed 57.2% of tasks at the compilation stage and another 18% due to test cases, ending up close to the other small models.

6.1.3. Detailed Compilation Results

The compiler results were analyzed to gain a more detailed understanding of why some test cases failed at the compilation stage.

In Figure 6.3, we show the number of occurrences for each error category and model as a heatmap.

The most common error category we found for each model is **identifier**. This is followed by either **computation** or **version**, depending on the model. For all

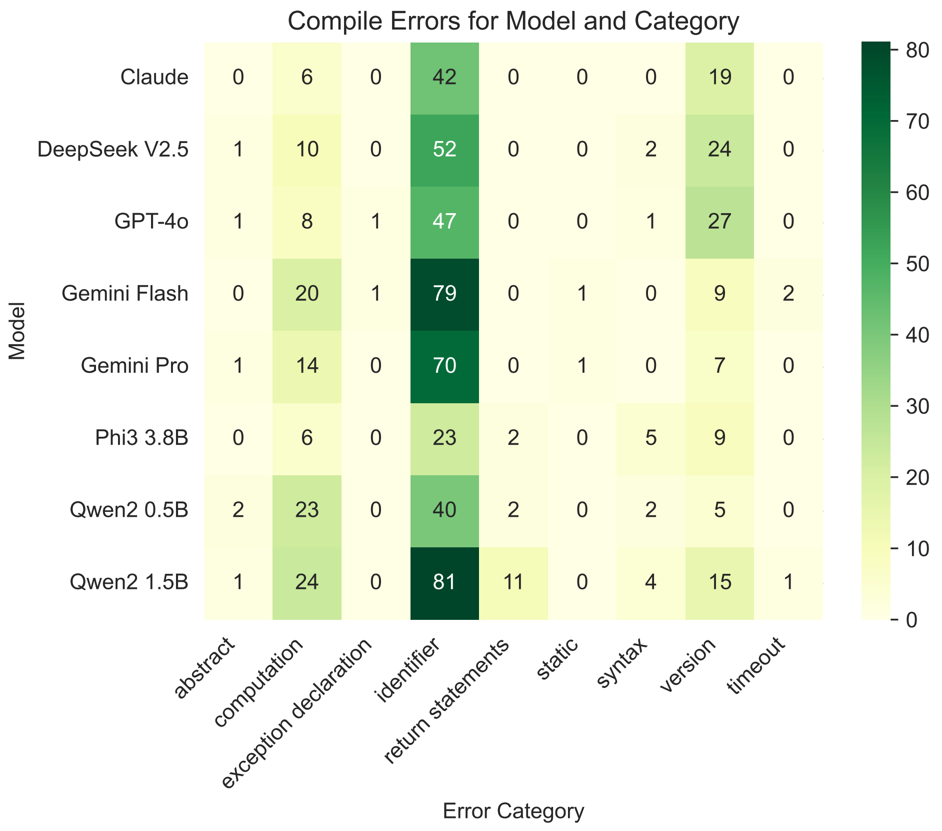


Figure 6.3.: **Code Generation Detail Results** are given in the absolute number of occurrences per category and model.

models except Claude, Gemini Flash, and Phi 3, the compiler found **syntax** errors that the syntax checker in the previous stage did not. Moreover, errors in the **abstract** category occurred for GPT-4o, Gemini 1.5 Pro, DeepSeek V2.5, and both Qwen2 models. Compilation-related issues with **return statements** only occurred in the smaller models, with Qwen2 1.5B producing a total of 11 tasks that failed with this error and the other two models producing two erroneous tasks each. Furthermore, the **exception declaration** and **static** error categories only occurred in individual cases.

The **timeout** category is not an error by the compiler but the surrounding framework. It denotes that the project compilation and the *pass_trigger* tests did not run through completely in 120 seconds. This happened twice for Gemini 1-5 Flash and once for Qwen2 1.5B.

6.1.4. Discussion

In the overall evaluation, we found that the five larger models consistently scored higher than the three smaller models. Claude 3.5 performed the best, followed by GPT-4o and DeepSeek V2.5. However, even though the large models performed better, all models also produced answers that failed to adhere to the syntax or could not be compiled, ultimately leading to an overall pass rate for the best-performing model of 39,5%. The most common compile errors for code that passed the initial syntax check were due to **identifier**, **computation**, or **version**-related issues. It is unsurprising that **identifier** and **computation** occurred often as both are broad categories. Moreover, the compilation errors in the **version** category can be attributed to the fact that some tasks in this benchmark have to be compiled using old Java versions and, therefore, do not support the newer language features the models used. However, we cannot mitigate this issue without major changes to the benchmark or Defects4J.

When examining the plot for the overall evaluation (see Figure 6.2), the lines for the three best-performing models look relatively straight, indicating that the models failed roughly equally due to compile errors and test cases. All other models exhibited a lower syntax pass rate and a visible kink. The latter indicates that the overall performance is disproportionately affected by the compilation errors rather than the test cases.

The benchmark results indicate that functional correctness is one, but not the only, limiting factor regarding model performance on this task. Depending on the model, incorrect syntax and compilation issues also caused a large performance drop. Although this seems to indicate that the practical utility of such models for generating code is limited, we argue that it can still be beneficial to use. Primarily because solid support systems like IDEs already exists. These can check the syntax and flag compiler errors using static analysis, addressing a sizable chunk of the failures in the evaluation. This allows a developer to see potential issues with the code immediately. We, therefore, argue that LLMs for code generation lend themselves to be used in a joint setup with IDEs.

Moreover, the results of the experiment seem to suggest that increasing the size of a model can potentially increase the ability to synthesize correct code. This can be observed in both the Gemini models as well as the Qwen2 models. However, this does not seem to hold across model families, as Phi3, a model that is larger than both Qwen2 models, performs consistently equal or worse. One likely reason for the poor performance of Phi3 could be that the model was primarily trained on Python code with a focus on standard packages[3].

[3] https://huggingface.co/microsoft/Phi-3-mini-128k-instruct#responsible-ai-considerations (Accessed: 25.11.2024)

One last limitation we need to mention is that the compiler only reports the error that caused the compilation to fail. This might mask further errors that may also be present in that code snippet. However, this does not affect the overall performance.

Answer for RQ2.1: How do the eight models compare in their code generation capability?

The larger models performed better than the smaller ones, but even the best-performing model (Claude 3.5 Sonnet) only managed to pass 39,5% of tasks, with two small models passing none. Moreover, all models struggled to some extent with syntax and compilation errors.

Answer for RQ3.1: Where in the code generation benchmark do the highest-scoring models perform well and where is room for improvement?

The three highest-scoring models, Claude 3.5 Sonnet, GPT-4o, and DeepSeek V2.5, were able to synthesize syntactically correct code in $> 95\%$ of cases. However, they ultimately failed more than half of all tasks due to compilation and test failures. The primary compilation errors observed are **identifier**, **version** and **computation**. However, version can not be mitigated without changing the to the structure of the underlying framework.

6.2. Code Completion

Another aspect of Java SD is code completion, which is the process of completing partially written code snippets. Traditionally, this feature was provided by IDEs using static analysis and usually only supported single statements.

However, with LLMs, code can be completed at various scopes ranging from single API calls to complete blocks.

6.2.1. Method

We evaluated the models using SAFIM[4] (Gong et al., 2024), a benchmark for the Fill-in-the-Middle (FiM) task. FiM means that a part within a text is blanked out and needs to be filled in.

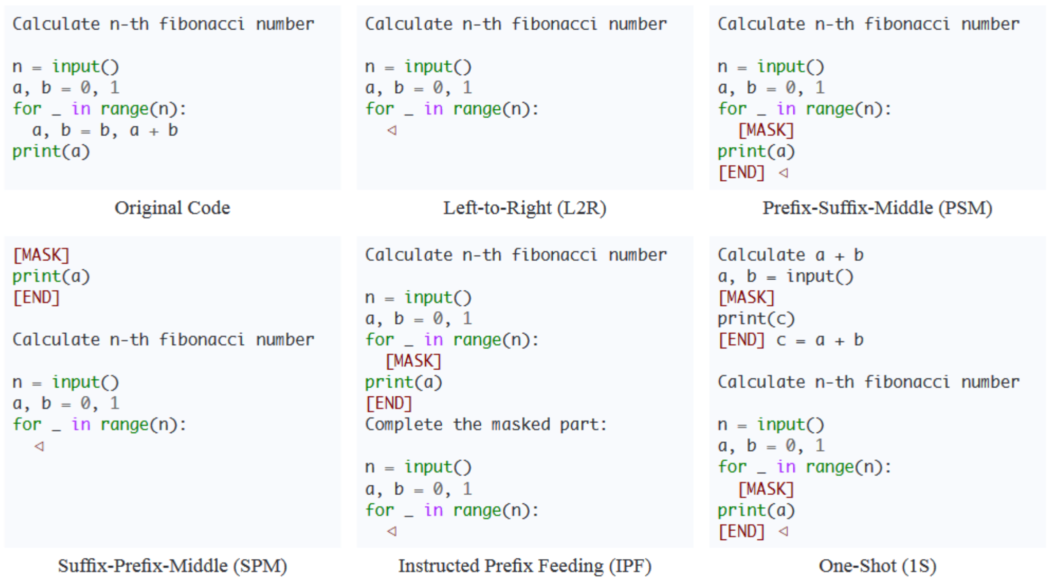
SAFIM evaluates three distinct categories and provides data for multiple languages. The categories are (1) block completion, (2) control-flow completion, and

[4]See https://github.com/gonglinyuan/safim (Accessed: 07.10.2024)

(3) API function call completion. Block completion involves completing a partially written block from, among others, methods, conditions, or loops. Control-flow completion requires the models to complete the conditional statements of control-flow statements. API function call completion involves finding the correct API call.

In SAFIM, the data for block completion and control statement completion originate from the programming contest website Codeforces[5] and the results are evaluated using unit tests. The "API function call" completion data was extracted from GitHub repositories, and the completions are evaluated using exact match (EM).

While block completion and control-flow completion feature many (2.478 and 2.464) individual Java tasks, we decided to keep the three categories equal in size and match the size of the Java API task set (56).



```
Calculate n-th fibonacci number

n = input()
a, b = 0, 1
for _ in range(n):
  a, b = b, a + b
print(a)
```

```
Calculate n-th fibonacci number

n = input()
a, b = 0, 1
for _ in range(n):
  ◁
```

```
Calculate n-th fibonacci number

n = input()
a, b = 0, 1
for _ in range(n):
  [MASK]
print(a)
[END] ◁
```

```
[MASK]
print(a)
[END]

Calculate n-th fibonacci number

n = input()
a, b = 0, 1
for _ in range(n):
  ◁
```

```
Calculate n-th fibonacci number

n = input()
a, b = 0, 1
for _ in range(n):
  [MASK]
print(a)
[END]
Complete the masked part:

n = input()
a, b = 0, 1
for _ in range(n):
  ◁
```

```
Calculate a + b
a, b = input()
[MASK]
print(c)
[END] c = a + b

Calculate n-th fibonacci number

n = input()
a, b = 0, 1
for _ in range(n):
  [MASK]
print(a)
[END] ◁
```

Figure 6.4.: **Code Completion Prompt**. Overview of the different prompt styles offered by SAFIM. We initially went with SPM but ultimately landed on 1S due to poor model performance. Taken from Gong et al. (2024). A shortened example for both SPM and 1S prompting can be seen in Appendix B.

Moreover, the SAFIM benchmark supports five different prompting approaches, as shown in Figure 6.4. Ideally, we would have wanted to use the standard infilling prompt (Prefix Suffix Middle (PSM)). However, this would require the models to

[5] https://codeforces.com/ (Accessed: 11.11.2024)

support FiM prompting explicitly, which relatively few do. Therefore, we initially went with Suffix Prefix Middle (SPM) prompting, which according to the authors is a way to extend PSM prompts to a wider range of models, by allowing them to process information in a ”left-to-right manner” (Gong et al., 2024). However, due to very poor performance (see Table 6.3), we ultimately went with One-Shot prompting (1S). We suspect that the poor performance of SPM stems from the fact that the code that comes after the infilling location (suffix) is placed at the very start of the prompt (see Appendix B). This deviates from a classic instruction structure. In contrast to that 1S prompting, provides an complete example before the actual task and denotes the infilling location with a comment.

For model parameters, we followed their approach of setting the temperature to 0.2, as this aligns with the value chosen for the other benchmarks. Moreover, they set top_p sampling to 0.95 for all models. We differ from that and use the default parameters for all models to stay consistent with our other evaluations.

The individual categories provided pass@1 scores, and we computed the overall score by taking the arithmetic mean for all three. For block and control-flow completion, the score is determined using input-output (IO) test cases, and for API function call completion, it is determined using a ground truth string.

Additionally, we employed a simple extraction step for Claude 3.5 Sonnet on Control and API tasks, as the benchmark framework could not extract the completions from the model answers in multiple cases. The regular expression[6] (RegEx) for this extraction is: ”:\\ n\\n(.+) (\\n\\n.)?”. However, we only extracted the first RegEx group to get the output.

Moreover, SAFIM provides error statistics for each model and category, which we also evaluated to understand the generated completions better. The only evaluation for ”API function call” is to check if the generated code matches the ground truth using EM. Therefore, we cannot provide further details here.

We chose this benchmark as it evaluates three different completion tasks and features the ability to select from five prompt styles. Additionally, the benchmark evaluates the generated completions using IO tests where applicable and falls back to text-based evaluation otherwise (API tasks).

6.2.2. Results

The performance for code completion is reported as pass@1, both for each category and for the combined (averaged) result. First, we briefly report the overall results for SPM prompting before presenting the results for one-shot (**1S**) prompting in

[6] https://en.wikipedia.org/wiki/Regular_expression (Accessed: 1.12.2024)

detail. The latter is the one we used for the final ranking.

The results for code completion using SPM prompts are presented in Table 6.3. There it can observed that many cells show a 0, indicating that no passing result was achieved on any of the tasks. Moreover, GPT-4o was the only model to score above 0% for all three tasks. In total, three models scored 0% on all three tasks. Based on these results, we concluded that the SPM prompt was not suited for the selected models and opted to use 1S prompting instead.

Model	**Block [Pass@1]**	**Control [Pass@1]**	**API [Pass@1]**	**Combined [Pass@1]**
GPT-4o	16.1	5.4	71.4	31.0
DeepSeek V2.5	10.7	0.0	39.3	16.7
Qwen2 1.5B	0.0	3.6	7.1	3.6
Qwen2 0.5B	0.0	1.8	3.6	1.8
Gemini Pro	1.8	0.0	0.0	0.6
Claude	0.0	0.0	0.0	0.0
Gemini Flash	0.0	0.0	0.0	0.0
Phi3 3.8B	0.0	0.0	0.0	0.0

Table 6.3.: The rounded results for of code completion with **SPM** prompting. Many entries are 0, meaning that no passing results were produced.

Figure 6.5 and Table 6.4 show the performance of the eight models on the benchmark using one-shot (1S) prompting. Here, the results show that all models, except for Qwen2 0.5B, scored above 0% for at least one of the categories.

For block completion, all larger models managed to pass tasks, with GPT-4o scoring highest at 73.2%, followed by DeepSeek V2.5 (69.6%), Claude 3.5 Sonnet (67.9%) and Gemini 1.5 Flash (64.3%). Gemini 1.5 Pro scored lowest for the larger models at 34%. Moreover, all smaller models scored 0%.

For control statement completion, GPT-4o and Claude scored 80.4%, followed by DeepSeek V2.5 at 76.8%. Regarding the Gemini models, it can be observed that the Pro variant (67.9%) scored higher than the Flash variant (64.3%). Both Qwen2 models achieved the lowest score at 0%, while Phi3 performed marginally better at 1.8%.

We found GPT-4o to perform best for API category, scoring 84%, followed by DeepSeek V2.5 (82.1%) and Claude (80.4%). Moreover, Gemini Flash once again outperformed the Pro variant. Furthermore, we found that for the first time in this benchmark, two of the three smaller models achieved a score above 0%, with Qwen2 1.5B scoring 5.4% and Phi3 3.8B reaching 1.8%.

Following the individual evaluations, the scores from the models were combined using the arithmetic mean. This resulted in GPT-4o scoring the highest at 79.2%, followed by Claude and Deep Seek V2.5, scoring 76.2%. The worst-performing model in this benchmark was Qwen2 1.5B, scoring 0% in all three categories.

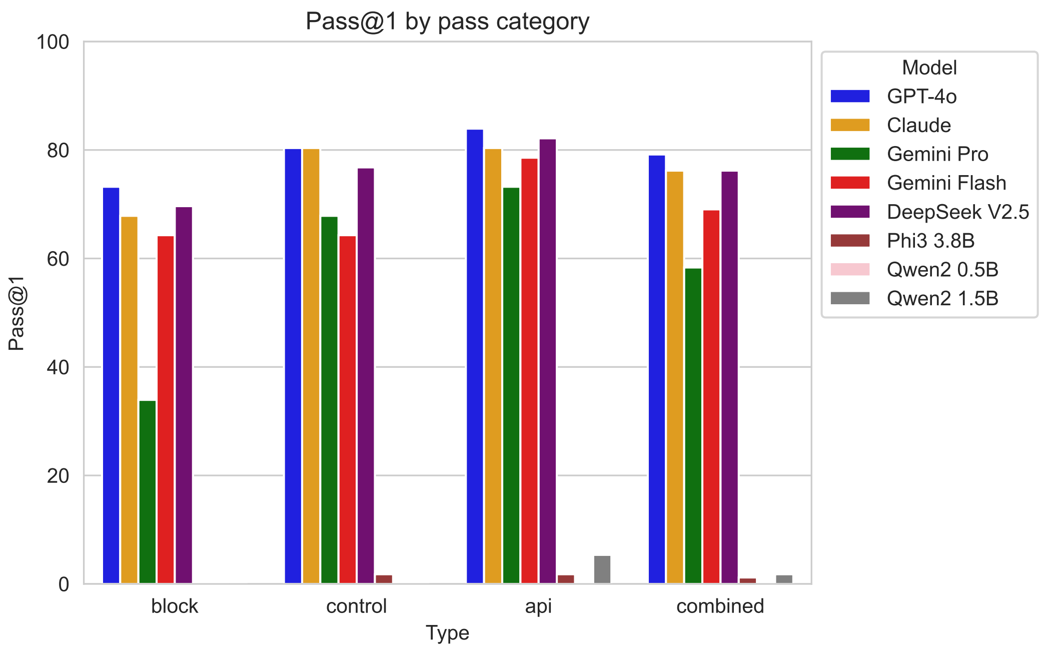


Figure 6.5.: **Code Completion Results** for the different models using one-shot (1S) prompting. Each model has a performance measured in pass@1 for each category. The combined score is the arithmetic mean computed over all three categories.

We visualized the error categories for block and control completion to understand the score differences better. API completion does not feature error categories, as the tasks are evaluated using EM.

Figure 6.6 details the failure types we encountered for each model when evaluating block completion. There, we found that both Qwen2 models primarily failed due to code that could not be compiled, which, based on manual inspection of the extracted snippets, was usually caused by the model not following the instructions. Moreover, Phi3 3.8B primarily failed due to empty responses or compilation errors. Upon further inspection of the produced output, we observed that most of it appears to be nonsensical text mixed with code. Moreover, the larger models primarily failed due to producing a "Wrong Answer," meaning that at least one of the test cases did not pass. However, Gemini 1.5 Pro was the exception, mainly

Model	Block [Pass@1]	Control [Pass@1]	API [Pass@1]	Combined [Pass@1]
GPT-4o	73.2	80.4	83.9	79.2
Claude	67.9	80.4	80.4	76.2
DeepSeek V2.5	69.6	76.8	82.1	76.2
Gemini Flash	64.3	64.3	78.6	69.0
Gemini Pro	33.9	67.9	73.2	58.3
Qwen2 1.5B	0.0	0.0	5.4	1.8
Phi3 3.8B	0.0	1.8	1.8	1.2
Qwen2 0.5B	0.0	0.0	0.0	0.0

Table 6.4.: The rounded results for code completion with **1S** prompting and a precision of 1.

failing due to compilation and runtime errors. When inspecting the outputs for Gemini 1.5 Pro, we noticed that, in many cases, the model outputted parts of the one-shot example.

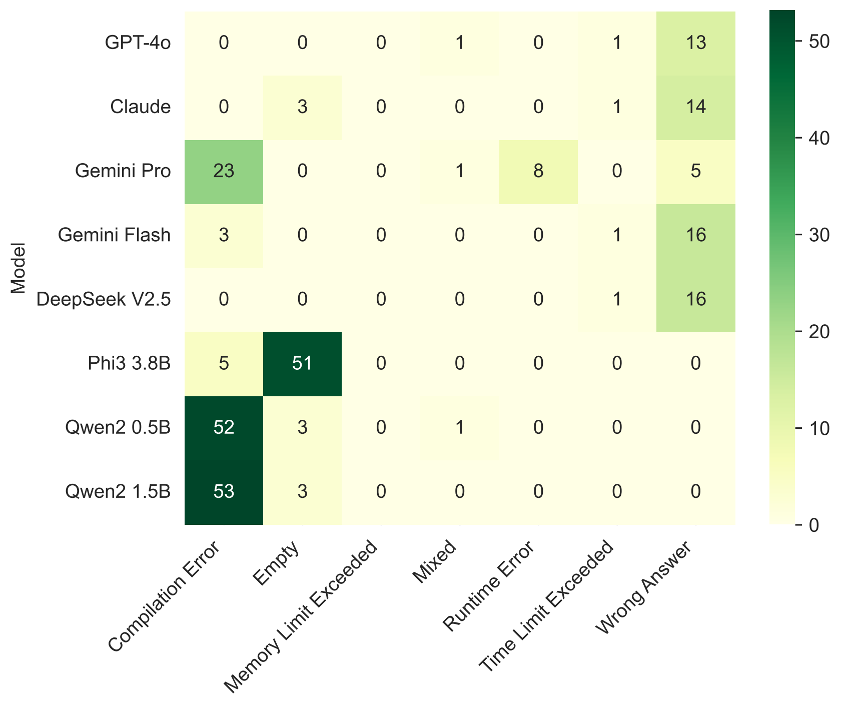
In Figure 6.7, we present the failure types encountered during control-flow completion. There, we found that for all larger models, the primary issue was ”Wrong Answer,” which indicates that at least one IO test failed. The small models all primarily failed due to compilation errors. Based on manual inspection, we found that Phi 3 often produced nonsensical texts. Moreover, the Qwen2 models sometimes included additional import statements in their output, leading them to fail tests this way. However, for both models, this only occurred in 14 or 15 cases, respectively.

6.2.3. Discussion

In the evaluation using the three categories in this benchmark, we found that the larger models performed remarkably better than the smaller models, which scored 0% in most instances. Notably, GPT-4o, Claude 3.5 Sonnet, and DeepSeek V2.5 formed the top 3 in all three categories. Based on the evaluation of the error categories, we found that primary reason for failure for most of the larger models was functional (”Wrong Answer”) and not syntax or compilation.

Moreover, we observed that Gemini 1.5 Flash outperformed the Pro variant in 2 of 3 categories. This occurred for block completion due to the Pro variant outputting the example in the one-shot prompt.

However, there exist some limitations to this benchmark. Firstly, the API completion is based on an exact match (EM) criterion, which could lead to an equivalent



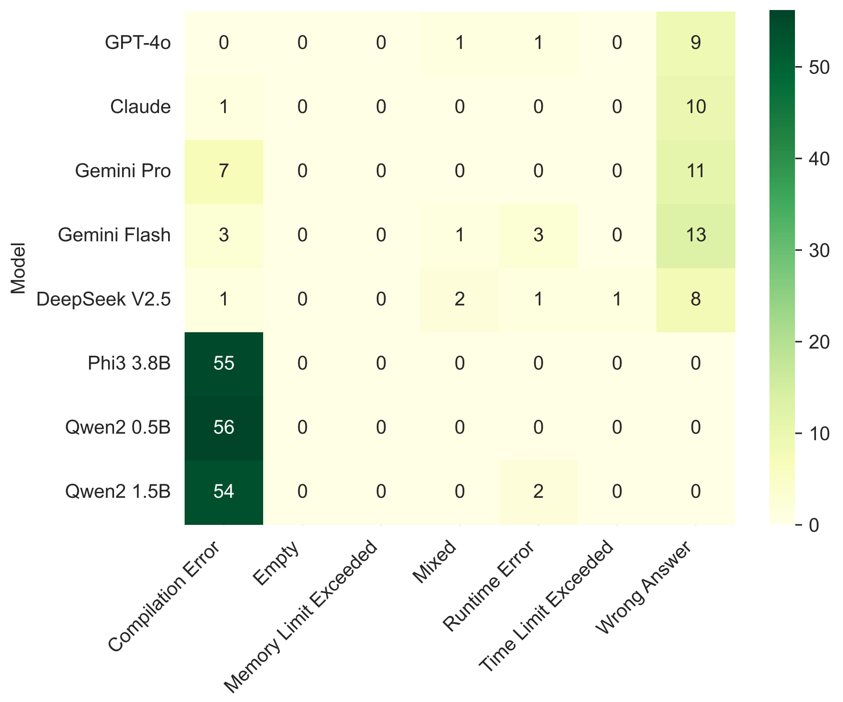
Model	Compilation Error	Empty	Memory Limit Exceeded	Mixed	Runtime Error	Time Limit Exceeded	Wrong Answer
GPT-4o	0	0	0	1	0	1	13
Claude	0	3	0	0	0	1	14
Gemini Pro	23	0	0	1	8	0	5
Gemini Flash	3	0	0	0	0	1	16
DeepSeek V2.5	0	0	0	0	0	1	16
Phi3 3.8B	5	51	0	0	0	0	0
Qwen2 0.5B	52	3	0	1	0	0	0
Qwen2 1.5B	53	3	0	0	0	0	0

Figure 6.6.: **Block Completion Error Categories** for the different models and 1S prompting.

but textually different answer being regarded as wrong despite being a viable alternative. Especially since seemingly inconsequential changes, like adding a blank space, can lead to changes in the answer (Salinas and Morstatter, 2024). Secondly, the benchmark features an approach to extraction that either assumes that the models adhere to the prompt and only output the snippet in question or that the models provide the snippet in question in a markdown style code notation. Notably, Claude 3.5 did not do this, initially leading to a low score. Fortunately, the model always included a colon (:) followed by two newlines before each snippet[7], allowing us to easily add an additional extraction step to extract the snippets properly.

Based on the evaluation we theorize that GPT-4o, Claude 3.5 Sonnet, and DeepSeek V2.5 should be well suited for code completion in practice, with the Gemini models likely performing okay. However, we would expect the smaller models not to be helpul in practice.

[7]If there was text after the snippet, there also always were two newlines as a separator.



Model	Compilation Error	Empty	Memory Limit Exceeded	Mixed	Runtime Error	Time Limit Exceeded	Wrong Answer
GPT-4o	0	0	0	1	1	0	9
Claude	1	0	0	0	0	0	10
Gemini Pro	7	0	0	0	0	0	11
Gemini Flash	3	0	0	1	3	0	13
DeepSeek V2.5	1	0	0	2	1	1	8
Phi3 3.8B	55	0	0	0	0	0	0
Qwen2 0.5B	56	0	0	0	0	0	0
Qwen2 1.5B	54	0	0	0	2	0	0

Figure 6.7.: **Control Statement Completion Errors** for the different models and 1S prompting.

Answer for RQ2.2: How do the eight models compare in their code completion abilities?

The larger models performed better than the smaller ones with GPT-4o, the best-performing model achieving a combined score of 79.2%. All the smaller models scored below 2% on average, with Qwen2 0.5B scoring 0% on all benchmarks.

Answer for RQ3.2: Where in the code completion benchmark do the highest-scoring models perform well and where is room for improvement?

The three highest-scoring models GPT-4o, Claude 3.5 Sonnet and DeepSeek V2.5 were able to synthesize correct completions in most cases, leading to scores between 76% and 80%. Moreover, they performed well consistently, forming the top 3 in each category. In cases where they failed, the primary issue was functional correctness (”Wrong answer”), with the occasional other error.

6.3. Unit Test Generation

Unit test generation is the task of synthesizing unit tests based on the method under test. In the following, we elaborate on the approach we used to evaluate and compare the capabilities of the models. Moreover, we show how the models performed and what learnings we could draw from them. Furthermore, we selected unit test generation, an aspect of test generation as a whole, as it is straightforward to evaluate and datasets/benchmarks are readily available.

6.3.1. Method

To evaluate the ability of the models to synthesize unit tests, we used the **”codeujbtestgen”** part of CoderUJB (Zeng et al., 2024). The task in this benchmark is to generate a unit test for a method under test, the surrounding context, and additional test-related information. This includes the method comment and signature of the test method and the surrounding context. All tasks in this benchmark were derived from Defects4j (Just et al., 2014), which also provides the environments to execute the test cases.

There are 140 tasks, and the execution structure is similar to code generation (see 6.1). However, ”pass_all” does not exist for ”codeujbtestgen,” and ”pass_trigger” is the relevant measure that indicates the following: (1) the test case passed and (2) the test case achieved a line coverage > 0.

We evaluated the benchmark as we did for code generation with a temperature of 0.2, a maximum number of newly generated tokens of 1024, and a sample size per task of 1.

The performance of the models is reported as **pass@1**, computed as the average of the pass@1 scores of each task. As we only generated one candidate per task, the pass@1 score of an individual task is a binary measure. Due to the nature of the results, they can also be interpreted as passing percentages.

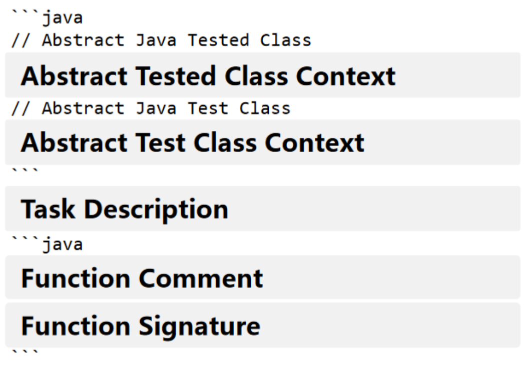

~~~
```java
// Abstract Java Tested Class
Abstract Tested Class Context
// Abstract Java Test Class
Abstract Test Class Context
~~~
Task Description
```java
Function Comment
Function Signature
```
```

Figure 6.8.: **Unit Test Generation Prompt**. A prompt example for unit test generation with CoderUJB. Taken from Zeng et al. (2024).

This benchmark also provides a prompting strategy, as shown in Figure 6.8. Here, it contains context information about the class under test and the test class.

In addition to the primary pass@1 scores, the benchmark also provides code-, branch- and diff-coverage values as measures for each task. As the result for "pass_trigger" does not consider these coverage values apart from a simple check if the test case achieves any coverage at all, we incorporated coverage values into our evaluation. Equation 6.1 shows how we computed and added the average of the two coverage values with a weight of 30%. Moreover, we calculated code- and branch-coverage based on all test cases that passed "pass_trigger." Crucially, we omitted the diff-coverage, as it represents the same as the code-coverage in this situation.

$$score = 0.7 * \frac{pass\_trigger}{100} + 0.3 * \frac{code\_coverage + branch\_coverage}{2} \tag{6.1}$$

We used the same categorization for compilation errors as for the code generation benchmark. This can be seen in Table 6.1.

The rationale for choosing this benchmark is the same as for code generation, namely: (1) An evaluation based on **executed code** by compiling and executing the unit test cases and (2) **additional context information** about both the classes of the method under test and the class of the test method.

6.3.2. Results

Figure 6.9 illustrates the performance of all eight models on this benchmark, and Table 6.5 lists the detailed pass@1 values and the combined final score. The results
```

were computed as the arithmetic mean of the binary pass@1 values from the 140 individual tasks.

The results in Figure 6.9 show that the five larger models performed better on the original benchmark than the smaller ones, with GPT-4o achieving the highest pass rate, for "pass_trigger." Overall, the larger models produced a passing test case on 12.9-22.1% of tasks, while the smaller models achieved between 0.7% and 2.1%.

Moreover, GPT-4o, Claude 3.5 Sonnet, and DeepSeek V2.5 all passed the syntax check in $\geq 95\%$ of tasks. Furthermore, the two Gemini models produced passing code in 68,6% (Pro) and 76,4% (Flash) of cases. For the small models, Qwen2 1.5B had the highest pass rate at 52.9%, followed by Phi 3 3.8B (37.1%) and Qwen2 0.5B (16.4%). **Claude 3.5 (47,9%)** achieved the highest pass rate for the compilation stage, followed by GPT-4o and DeepSeek V2.5, which both passed in around 38% of cases. The Gemini models reached 31.4% (Pro) and 33.6% (Flash). Both Qwen2 models and Phi 3 were closer together, passing the stage with rates ranging from 3.6% to 5.7%.

In Figure 6.10, we provide the average line- and condition-coverage rates per model and the number of tasks on which this could be computed. It must be noted that the results for the smaller models are based on either one or three passing tasks, which is a relatively small number compared to the large models. Considering only the large models, Claude produced code that, on average, covered the largest percentage of lines at 44.3%, followed by DeepSeek V2.5 (43.1%) and Gemini Flash (41.8%). For condition coverage, the order of the three best-performing large models was reversed, with Gemini Flash covering the highest percentage at 32.4%, followed by DeepSeek V2.5 (29.5%) and Claude (28%).

Moreover, Table 6.5 also shows the final weighted score that combines the results from *pass_trigger* with the coverage rates at a 70-30 ratio. Based on this final score, GPT-4o performed best, achieving 0.244, followed by Claude 3.5 Sonnet (0.228) and DeepSeek V2.5 (0.209). Qwen2 0.5B and Phi 3 3.8B performed the worst.

For unit test generation, we also analyzed the compilation errors for all models, which we present in Figure 6.11. The categories are the same as for the code generation benchmark and can be seen in 6.1. The most common error category for every model was **identifier**. Moreover, for all models except Phi3, the second and third most common error categories were **computation** and **version** in varying order. The second most common error category for Phi3 was **version**, followed by **syntax**. Generally, **syntax**, **abstract**, and **exception** categories occurred infrequently. No compilation errors occurred in the categories **return statements** or **static**. However, there was a single **timeout**.

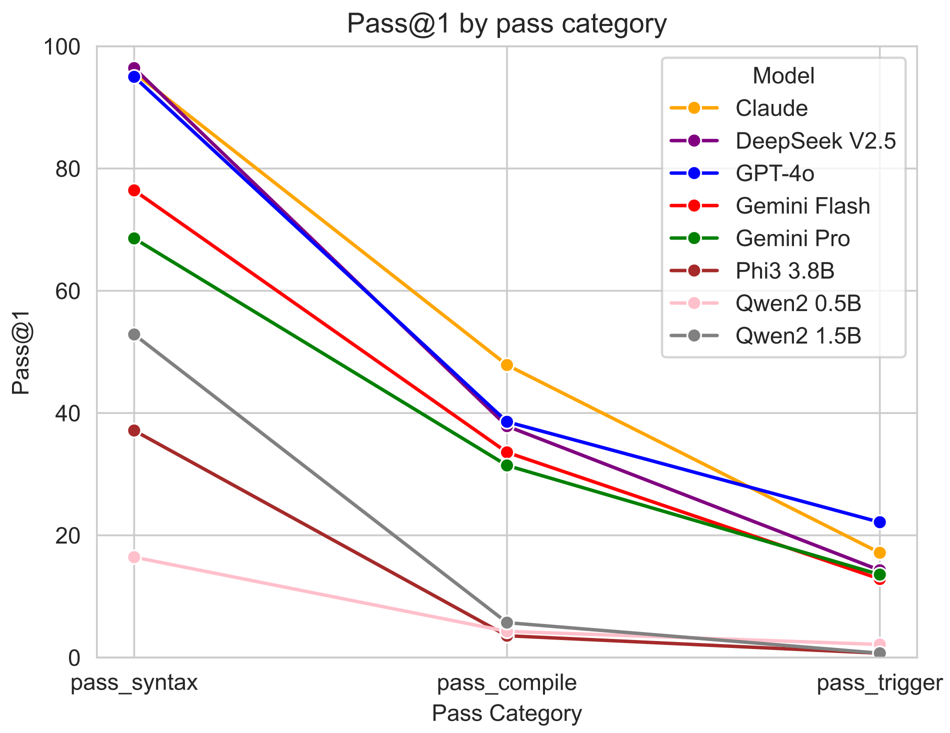


Figure 6.9.: **Unit Test Generation Results** on the *codeujbtestgen* benchmark. Each model has a performance measured in pass@1 for each task. The relevant score here is pass_trigger.

### 6.3.3. Discussion

In the overall evaluation, we found that the larger models consistently outperformed the smaller ones, even though all models seem to have substantial problems reliably producing compileable code. After weighing the coverage percentages, we found GPT-4o scored the highest at 0.175, followed by Claude 3.5 Sonnet (0.139) and DeepSeek V2.5 (0.116). The most common compiler errors we observed were **identifier**, **computation**, and **version**.

When looking at the pass@1 values illustrated in Figure 6.9, it can be observed that the three best-performing models were scoring roughly the same in terms of syntactical correctness. In general, Claude seems more capable of generating unit tests that successfully compile, as code from GPT-4o and DeepSeek V2.5 failed to compile more frequently. However, despite having fewer passing tasks after the

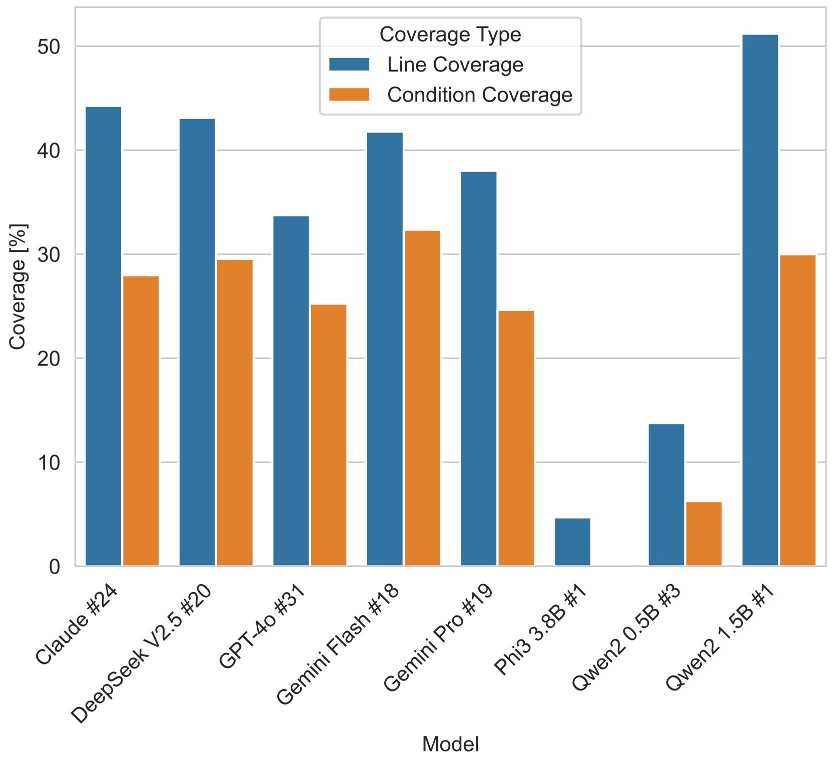


Figure 6.10.: **Unit Test Generation Coverage** for lines and conditions averaged over all passing tasks per model. The number of passing samples for each model is listed in the x-axis description.

compile stage, GPT-4o produced more passing test cases than Claude. This could indicate that GPT-4o is better suited to generate unit tests. However, the coverage rates reported in Figure 6.10 seem to contradict that indication, seeing as GPT-4o achieved the lowest line and second lowest condition coverage rate of the larger models. However, one limitation to the expressiveness of these observations is that the benchmark provides a method comment for each test case detailing what precisely should be tested. Therefore, the goal for line- and condition-coverage rates is not given. This led us to weigh "pass_trigger" 70% and the coverage 30%. Ultimately, GPT-4o scored the highest, followed by Deep Seek V2.5, with the other three larger models all being within a window of 0.02.

The smaller models appear unsuitable for unit test generation, with the best of the three having a pass rate of only 5.7% after the compilation stage. One potential explanation for this could be that the prompt contains information about

| **Model** | **pass_syntax [pass@1]** | **pass_compile [pass@1]** | **pass_trigger [pass@1]** | **Weighted Score** |
|---|---|---|---|---|
| GPT-4o | 95.0 | 38.6 | <u>22.1</u> | <u>0.244</u> |
| Claude | 95.7 | <u>47.9</u> | 17.1 | 0.228 |
| DeepSeek V2.5 | <u>96.4</u> | 37.9 | 14.3 | 0.209 |
| Gemini Flash | 76.4 | 33.6 | 12.9 | 0.201 |
| Gemini Pro | 68.6 | 31.4 | 13.6 | 0.189 |
| Qwen2 1.5B | 52.9 | 5.7 | 0.7 | 0.127 |
| Qwen2 0.5B | 16.4 | 4.3 | 2.1 | 0.045 |
| Phi3 3.8B | 37.1 | 3.6 | 0.7 | 0.012 |

Table 6.5.: Pass rate for all stages of ”codeujbtestgen” with a precision of 1. The weighted score is given with a precision of 3.

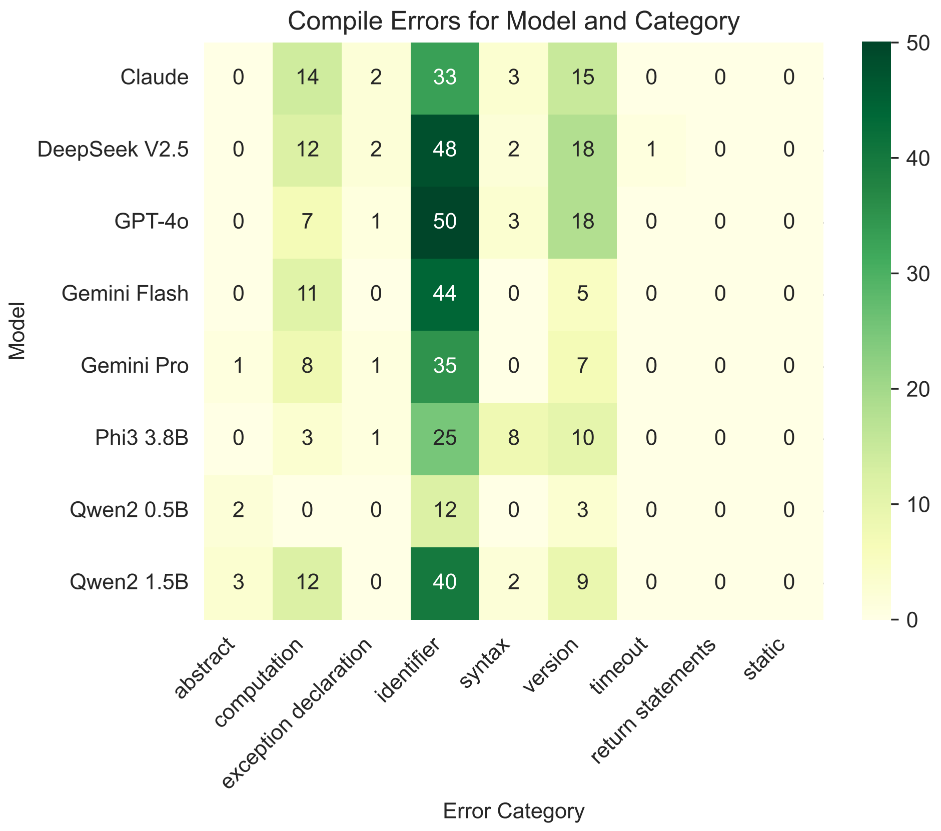
two classes and that the smaller models could have issues correctly assigning the information.

Similar to code generation, we generally found the most common compilation errors in the **identifier**, **computation**, or **version** categories. Similar to the other CoderUJB benchmarks, the compilation step only reports the first error, potentially hiding further ones that may also exist. Moreover, **version**-related errors can unfortunately not be avoided for the same reasons as in code generation. Furthermore, we argue that in practice, the other two error types could easily be mitigated using the LLM **and** an IDE capable of detecting such problems using static analysis.

Based on the sizable score difference between the smaller and larger models, model size seems to impact model performance for unit test generation. However, based on the ultimately slight score difference between Gemini 1.5 Pro and Flash (less than 0.04) and between the smaller models, we would argue that it is less pronounced than in code generation.

Answer for RQ2.3: How do the eight models compare in their unit test generation abilities?

The larger models performed better than the smaller ones, but even the best-performing model (GPT-4o) only achieved a pass rate of 22.1% and a final weighted score of 0.244. Two of the smaller models did not achieve a score higher than 0.045. All models failed most of their tasks due to compiler and syntax errors.



| Model | abstract | computation | exception declaration | identifier | syntax | version | timeout | return statements | static |
|---|---|---|---|---|---|---|---|---|---|
| Claude | 0 | 14 | 2 | 33 | 3 | 15 | 0 | 0 | 0 |
| DeepSeek V2.5 | 0 | 12 | 2 | 48 | 2 | 18 | 1 | 0 | 0 |
| GPT-4o | 0 | 7 | 1 | 50 | 3 | 18 | 0 | 0 | 0 |
| Gemini Flash | 0 | 11 | 0 | 44 | 0 | 5 | 0 | 0 | 0 |
| Gemini Pro | 1 | 8 | 1 | 35 | 0 | 7 | 0 | 0 | 0 |
| Phi3 3.8B | 0 | 3 | 1 | 25 | 8 | 10 | 0 | 0 | 0 |
| Qwen2 0.5B | 2 | 0 | 0 | 12 | 0 | 3 | 0 | 0 | 0 |
| Qwen2 1.5B | 3 | 12 | 0 | 40 | 2 | 9 | 0 | 0 | 0 |

Figure 6.11.: **Unit Test Generation Detail Results** are given in absolute number of occurrences per category and model.

Answer for RQ3.3: Where in the unit test generation benchmark do the highest-scoring models perform well and where is room for improvement?

The three highest-scoring models, GPT-4o, Claude 3.5 Sonnet, and DeepSeek V2.5, exhibited next to no problems with syntax. The primary issues were compilation errors and test case failures, with compilation errors accounting for the larger drop in pass rate. The most common compilation error categories were **identifier**, **version** and **computation**. However, version-related issues cannot be mitigated without fundamental changes to the underlying framework. The Gemini models failed more often due to syntactical problems but less frequently due to compilation errors than the other three, ultimately reaching a similar weighted score to the other three larger models.

## 6.4. Method Comment Generation

Method comment generation is the task of generating a descriptive comment about the method in question. We selected method comment generation as an aspect of documentation generation as it is straightforward to apply and evaluate. Moreover, datasets containing Java method and comment pairs are readily available.

### 6.4.1. Method

We employed a quantitative evaluation to assess the ability of the models to generate comments from code. We utilized the DeepCom dataset from Hu et al. (2018a), which consists of Java method and comment pairs and is split into train, test, and validation sets.

From the test set, we sampled 300 pairs and evaluated the models on them. Since we could not fine-tune the LLMs on this task, we utilized one-shot prompting to convey specific information about this task. The prompt is based on our experience with LLMs and contains influences from the prompts in the other comparisons. The prompt we used contains a manually selected sample[8] from the part of the test set we did not use[9].

```
You are a professional Java programmer, please create a
short comment that explains what the following method does.
Only answer with the comment.
Write nothing else.

protected String renderUri(URI uri){
  return uri.toASCIIString();
}
Comment: Render the URI as a string.

[METHOD_TO_SUMMARIZE]
Comment:
```

We evaluated the output of the model using ROUGE-1 (Lin, 2004) and computed the average over all 300 samples. We specifically chose ROUGE-1 with word stemming as it is a measure that can be used to evaluate automatic summarization, and code comments can be seen as a summary of code. The comparison was done by comparing the word unigram overlap between the reference and the generated

[8]This sample was selected as it is simple, self-contained (i.e., not dependent on external context information) and the comment accurately describes the method.

[9]Please note that the line-break between ”a” and ”short” is not present in the actual prompt.

summary. We evaluated the model performance based on the F1-score (see Formula 6.4) as it balances precision (see Formula 6.2) and recall (see Formula 6.3).

$$precision = \frac{\#overlapping\ unigrams}{\#prediction\ unigrams} \tag{6.2}$$

$$recall = \frac{\#overlapping\ unigrams}{\#target\ unigrams} \tag{6.3}$$

$$F_1 = \frac{2 * precision * recall}{precision + recall} \tag{6.4}$$

All three formulas were taken from the source code of the "rouge-score" Python package[10]. Moreover, as an additional preprocessing step, we removed the comment notation[11] present in some generated answers before comparing the texts.

We selected this approach as method-comment pairs for Java are readily available. Moreover, we utilized ROUGE-1 as we dealt with short method comments and commenting code can be viewed as a special form of summarization.

6.4.2. Results

The results for the model performance are given as the F-1 score of ROUGE-1 and are plotted in Figure 6.12 with the mean on the x-axis and the median on the y-axis. It can be observed that the mean and median tend to be close, although the median is always slightly lower. The best-performing model based on the mean here was the smaller Gemini Flash (0.31) model, followed by Gemini Pro (0.308) and DeepSeek V2.5 (0.3). GPT-4o scored 0.286 on this benchmark, and the worst-performing larger model, Claude 3.5 Sonnet, achieved 0.256. Moreover, Qwen2 1.5B (0.288) managed to outperform both GPT-4o and Claude 3.5 in this benchmark, and Phi3 (0.09), as well as Qwen2 0.5 (0.029) had clearly lower scores than all other models.

Table 6.6 shows the precision, recall, and F1 scores for each model. Please note that all three values were first computed based on the individual tasks and then combined using the arithmetic mean. It can be observed that Qwen2 1.5B achieved the highest precision but the third-lowest recall value. Moreover, Claude 3.5 Sonnet, the worst-performing larger model according to the F1-score, achieved the highest recall but the third-lowest precision value. Gemini 1.5 Flash, which achieved the highest F1 score, scored second-highest for precision and fifth-highest for recall.

[10]https://pypi.org/project/rouge-score/#description (Accessed: 24.11.2024)

[11]This includes //, /*, */, /**, ...

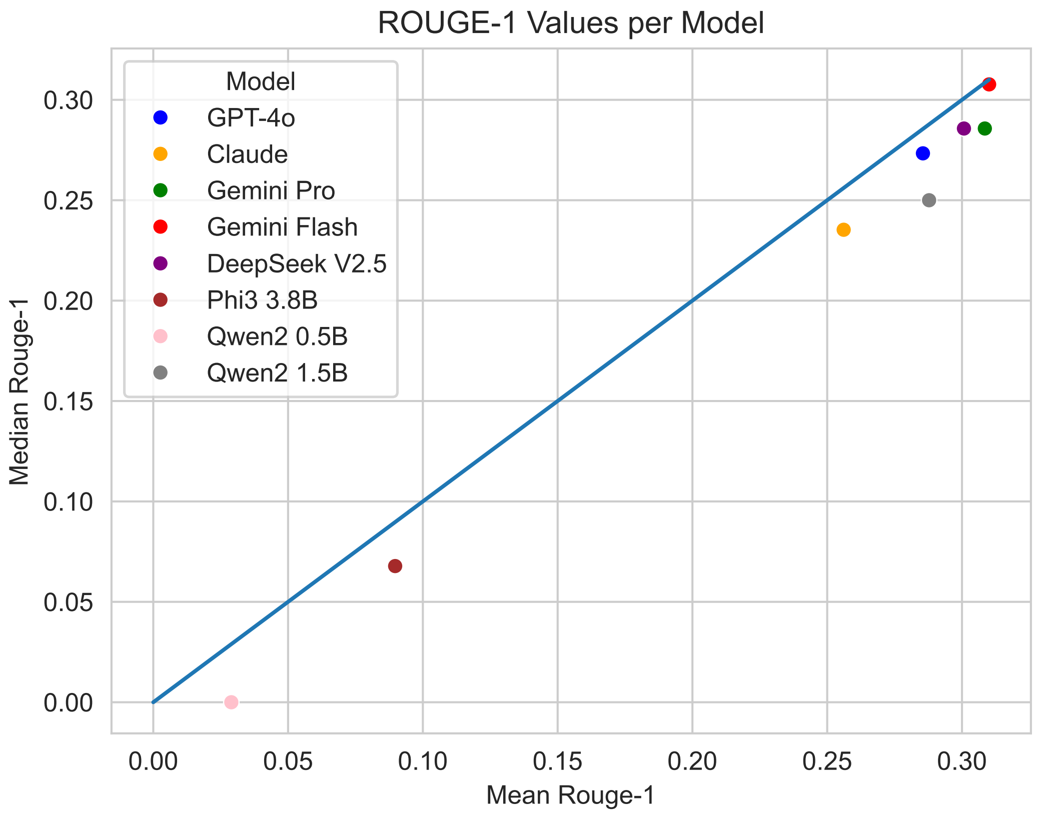


Figure 6.12.: **Method Comment Generation Results** of the sampled (n=300) benchmark. Scored using ROUGE-1 F1 and plotted as median over mean. The blue line is at d=0, k=1 and represents mean=median.

In Figure 6.13, we report the length of the comments in the dataset and the length of the comments generated by the eight models. We found that most models produced comments similar in length to the dataset. However, three models clearly differed. Firstly, we observed that comments generated by Claude 3.5 Sonnet were longer than for most other models. Secondly, Phi3 3.8B created very long comments, and a manual inspection revealed that the output was often nonsensical. Finally, Qwen2 0.5B answered with the function from the one-shot example in most cases.

Model	Precision	Recall	F1
Gemini Flash	0.350	0.354	0.310
Gemini Pro	0.330	0.378	0.308
DeepSeek V2.5	0.301	0.392	0.301
Qwen2 1.5B	0.381	0.290	0.288
GPT-4o	0.278	0.389	0.286
Claude	0.216	0.470	0.256
Phi3 3.8B	0.068	0.254	0.090
Qwen2 0.5B	0.031	0.052	0.029

Table 6.6.: The average ROUGE-1 scores by model rounded to a precision of 3. The values for all columns were computed using the arithmetic mean of the results for the individual tasks. This means that computing the F1 score based on the values in the table would differ from the ones present.

6.4.3. Discussion

In the evaluation we, observed that all larger models outperformed all but one smaller model. Qwen2 1.5B achieved a higher F1 performance than GPT-4o and Claude 3.5 Sonnet. While the best-performing model only scored a mean ROUGE-1 F1 score of 0.31, which is relatively low for a value that can, in theory, range from 0 to 1, we argue that this is to be expected as there is more than one way to summarize a method.

Surprisingly, Qwen2 1.5B scored relatively high, which we assume is partly accounted for because it primarily provided shorter answers.

Moreover, the lower score of Claude 3.5 can, in part, be explained by the longer comment lengths, which lead to a lower precision score of the ROUGE-1 F1 score, thereby reducing the overall score. The same rationale applies to Phi3 3.8B, as can be seen in Table 6.6. Moreover, Phi3 also seems to tend to output nonsensical texts that have no apparent connection to the prompt.

The worst performing model was Qwen2 0.5B, which achieved a low score due to the tendency to not adhere to the prompt and to output the code from the one-shot example.

Based on the results, we conclude that all models except Phi3 and Qwen2 0.5B are generally suited for method comment generation. However, Claude 3.5 Sonnet seems to tend to synthesize lengthy text, which we would consider a slight disadvantage. Moreover, while we did not exhaustively check the comments as this is a quantitative evaluation, the comments from the best-performing models we actually looked at were good. However, based on that we can not generally say whether all the comments make sense and are helpful.

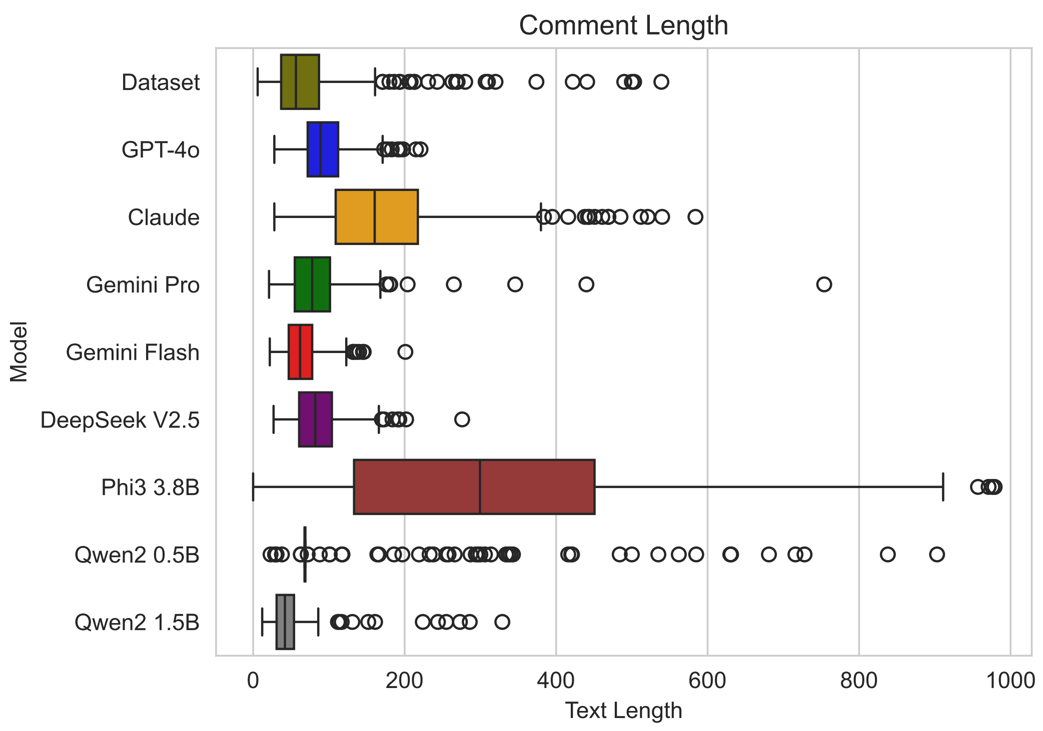


Figure 6.13.: **Method Comment Generation Comment Length** for all eight models and the dataset. Outliers with a length of >1000 characters were removed. These are 10 for Phi3 and 12 for Qwen2 0.5B.

Answer for RQ2.4: How do the eight models compare on their ability to synthesize method comments?

The larger models and Qwen2 1.5B scored the best in our evaluation. However, Claude 3.5 scored lower than the others on the F1 score. This was primarily due to longer answers. Phi3 and Qwen2 1.5B scored very low and seem unsuitable for practical application.

Answer for RQ3.4: Where in the method comment generation benchmark do the highest-scoring models perform well and where is room for improvement?

The six highest-scoring models primarily performed well due to the comparatively high token overlap (i.e. high precision and recall scores) combined with producing text of roughly similar length to the dataset. Claude 3.5 Sonnet tended to create lengthier comments, leading to a lower score. Except for Claude, there is no clear room for improvement for the six best-performing models, as the generated comments we looked at generally seemed good.

6.5. Automated Program Repair

Automatic Program Repair is the task of automatically correcting defective code. It is one approach to fixing bugs, and we selected it because several benchmarks are readily available for this sub-aspect of software correction[12]. Moreover, this aspect is straightforward to evaluate. In the following, we elaborate on the approach we used to assess and compare the capabilities of the models. Furthermore, we show how the models performed and what learnings we could draw from them.

6.5.1. Method

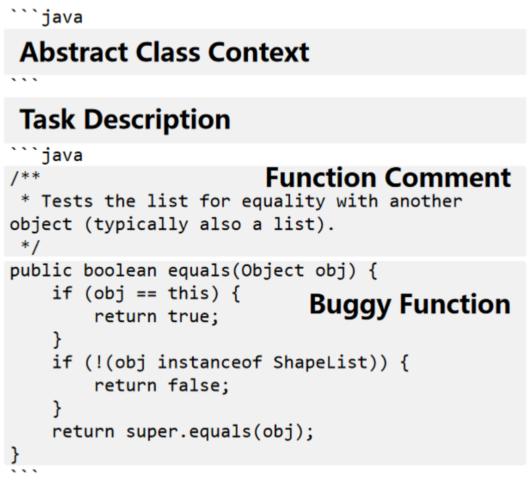
To evaluate the models on their ability to fix code, we utilized the **codeujbrepair** benchmark from CoderUJB (Zeng et al., 2024). The task in this benchmark is to repair methods that contain bugs. The available information for each task is the method itself, the method comment, and class context information. The individual tasks in this benchmark stem from the "single-function defects" in Defects4j. These were extracted by Xia et al. (2023) and subsequently used in this benchmark. A "single-function defect" is defined as a defect that can be corrected by only editing code withing a single method. Additionally, Defects4j was used to execute and evaluate the fixes.

The benchmark comprises 470 tasks, which are evaluated in four stages. These are the same for code generation (see Section 6.1) and unit test generation (see Figure 6.8). However, both "pass_trigger" and "pass_all" are relevant for this benchmark. Here, the former indicates whether the previously failing test cases now passed, and the latter indicates whether all tests passed. We once again ran the benchmark in accordance with the instructions provided by the benchmark creators. This involved setting the temperature of the LLM to 0.2 and the maximum

[12]https://program-repair.org/benchmarks.html (Accessed: 21.11.2024)

number of newly generated tokens to 1024. Moreover, we generate one sample per task.

The performance of the models is reported as **pass@1**, computed as the average of the pass@1 scores of each task. As we only generated one candidate per task, the pass@1 score of an individual task is a binary measure. Due to the nature of the results, they can also be interpreted as passing percentages.



~~~
```java
Abstract Class Context
```
Task Description
```java
/**                                    Function Comment
 * Tests the list for equality with another
object (typically also a list).
 */
public boolean equals(Object obj) {
    if (obj == this) {
        return true;                   Buggy Function
    }
    if (!(obj instanceof ShapeList)) {
        return false;
    }
    return super.equals(obj);
}
```
~~~

Figure 6.14.: **APR Prompt**. A prompt example for APR with CoderUJB. Taken from Zeng et al. (2024).

The prompt generation strategy is similar to code generation, differing only in the task description and by including the entire method instead of the method header (see Figure 6.14).

As for code and test case generation, we categorized and reported the compilation errors based on the information in Table 6.1.

The rationale for choosing this benchmark is similar to code generation and test case generation. Another factor is that this benchmark uses Defects4j, a well-explored benchmark for APR (Xia et al., 2023) .

6.5.2. Results

The performance of each model on the APR benchmark is given as pass@1 and visualized in Figure 6.15. Moreover, the exact scores are reported in Table 6.7. The results are computed as the mean of the binary pass@1 values from the 470 individual tasks. As the pass@1 scores of the individual tasks are binary, the resulting combined values can also be interpreted as percentage values, which we did in the following.

The results show that the five larger models outperformed the smaller models, with Claude performing the best in the ”pass_all” category (25.1%). The other larger models managed to pass between 18.1% (GPT-4o) and 15.7% (Gemini 1.5 Flash) of tasks. All smaller models scored below 1%.

Moving back to ”pass_syntax,” the first stage, all larger models passed between 88.3% and 79.1% of tasks and were closely followed by Qwen2 1.5B with a pass rate of 77.2%. The other smaller models were further off, with the smaller Qwen2 models scoring 64.5% and Phi3 reaching 17.2%.

In ”pass_compile,” Gemini Flash synthesized passing code in 79.6% of cases, followed by Claude, GPT-4o, and Gemini Pro, which all scored between 72.1% and 69.1%. DeepSeek V2.5 scored the lowest of the large models, with 64.3%. The Qwen models managed to pass this stage 54.9% (1.5B) and 47.4% (0.5B) of times, respectively, with Phi3 3.8B scoring the lowest (3.8%) by a sizable margin.

For ”pass_trigger,” the stage that executes the initially failing test case(s), the performance dropped notably, with Claude, the best-performing model, now managing to pass 30% of tasks. It is followed by DeepSeek V2.5, which scored 25.7%, and the other three larger models, which achieved between 21.9% and 20.2%. The smaller models all scored below 3%.

Model	**pass_syntax [pass@1]**	**pass_compile [pass@1]**	**pass_trigger [pass@1]**	**pass_all [pass@1]**
Claude	88.3	72.1	30.0	25.1
GPT-4o	82.6	69.1	21.9	18.1
DeepSeek V2.5	79.1	64.3	25.7	17.4
Gemini Pro	83.2	70.9	20.9	16.6
Gemini Flash	87.9	79.6	20.2	15.7
Qwen2 1.5B	77.2	54.9	2.8	0.6
Phi3 3.8B	17.2	3.8	1.1	0.4
Qwen2 0.5B	64.5	47.4	1.3	0.2

Table 6.7.: The numeric results for the APR benchmark with a precision of 1.

The compilation errors that occurred during the evaluation of this benchmark are visualized in Figure 6.16. The most common error categories were **identifier**, **computation**, and **version**. Moreover, for Phi 3, the compilation failed 6 times due to syntactical errors initially missed by the parser in the first stage. Both Qwen2 models had multiple instances of **missing return statement**s. All other error categories did not occur more than three times per model.

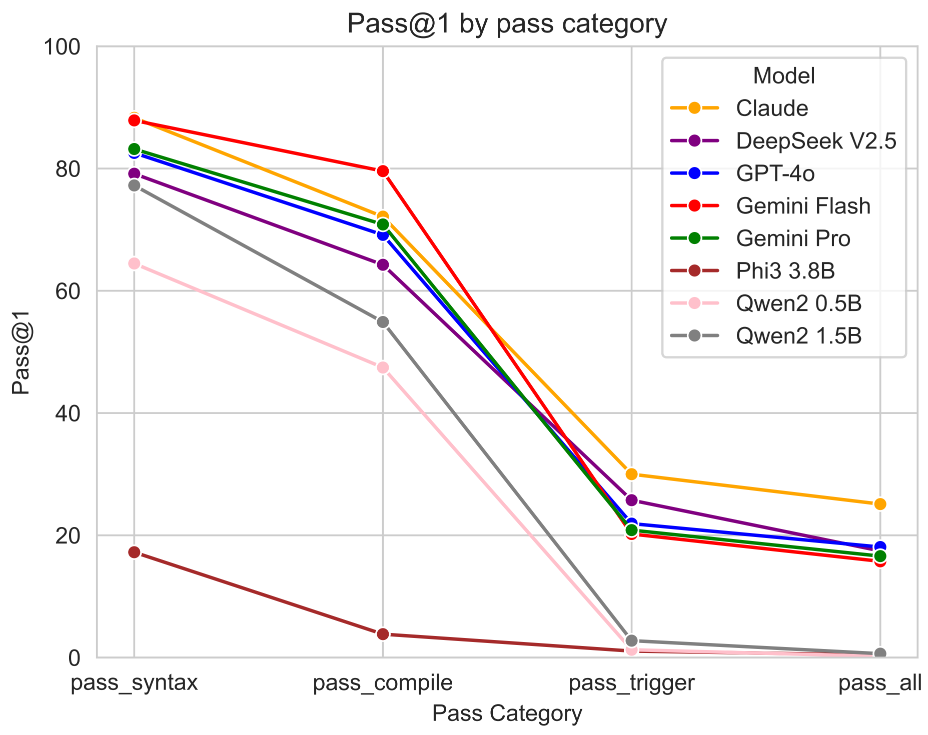
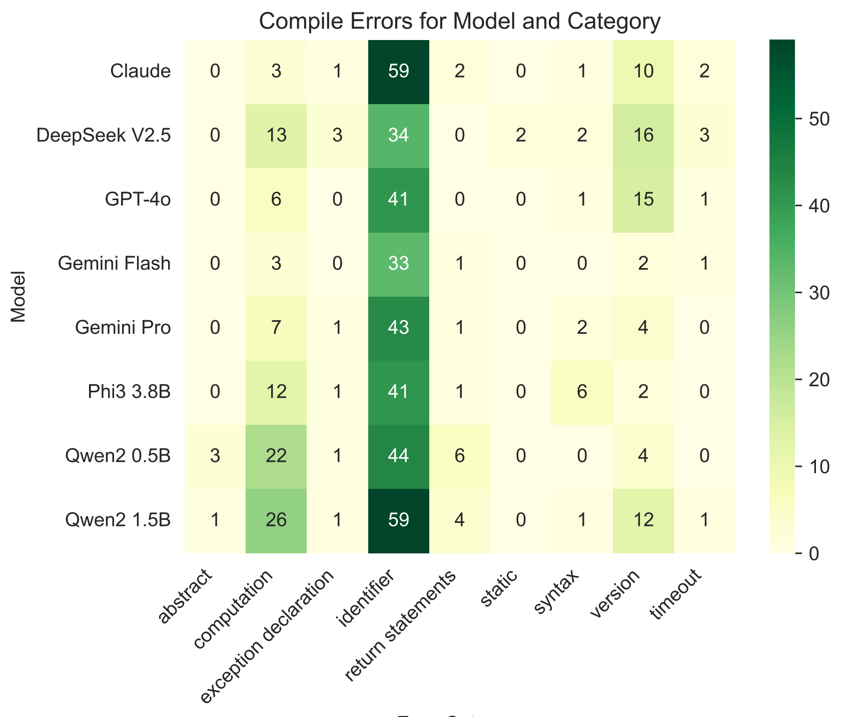


Figure 6.15.: **APR Results** for the codeujbrepair benchmark. Results are reported as pass@k-1, and *pass_all* is the relevant measure.

6.5.3. Discussion

In the evaluation, we observed that the larger models outperformed the smaller ones at every stage, with Claude 3.5 Sonnet ultimately achieving the best score with a pass@1 of 25.1%. Although, the two Qwen2 models were not to far from the larger models for the first two parts, Phi3 scored only 17.2% on ”pass_syntax.” After that, the performance differences became apparent. We generally observed a sizable drop for every model when evaluating the previously failing test cases. This was followed by a small decrease in performance when extending the evaluation scope to all test cases. Furthermore, the most common compiler errors were once again **identifier**, **computation** and **version**.

Looking at the pass@1 values at the different stages, we noted that, surprisingly, the smaller Gemini model achieved the second-highest pass@1 score for ”pass_syntax.” Moreover, the Flash variant achieved the highest score for



Model	abstract	computation	exception declaration	identifier	return statements	static	syntax	version	timeout
Claude	0	3	1	59	2	0	1	10	2
DeepSeek V2.5	0	13	3	34	0	2	2	16	3
GPT-4o	0	6	0	41	0	0	1	15	1
Gemini Flash	0	3	0	33	1	0	0	2	1
Gemini Pro	0	7	1	43	1	0	2	4	0
Phi3 3.8B	0	12	1	41	1	0	6	2	0
Qwen2 0.5B	3	22	1	44	6	0	0	4	0
Qwen2 1.5B	1	26	1	59	4	0	1	12	1

Figure 6.16.: **APR Detail Results** are given in absolute number of occurrences per category and model.

"pass_compile" with a margin of 7.5% to Claude, the second-best performing model. However, Gemini 1.5 Flash ultimately achieved last place among the larger models.

Additionally, we found that apart from Claude, the best-performing model, all other large models were within 2.4% in the final score. There, we also observed that while DeepSeek V2.5 scored better for "pass_trigger," it was ultimately overtaken by GPT-4o in "pass_all." This could indicate that DeepSeek V2.5 is better at fixing bugs than GPT-4o, but at the same time, introduces more new bugs.

Again, we observed that the most common compilation error categories were identifier, computation, and version. However, the expressiveness of the errors is limited by the fact that version errors cannot be mitigated without fundamentally changing the benchmark and that only the first error was reported, hiding further potentially existing ones.

In general, APR is the first benchmark from CoderUJB for which the syntax and compilation errors seem less problematic, and functional correctness becomes more relevant. However, the fact that all tasks for this benchmark stem from the ”single-function defects” part of Defects4j could allow models to perform better more easily, as we assume context plays a minor role under those circumstances. Moreover, with the highest passing percentage being 25.1% and the other larger models scoring in the range of 15-20%, it remains to be seen how the practical experience with APR will be.

Answer for RQ2.5: How do the eight models compare regarding their APR abilities?

The larger models performed better than the smaller ones, but even the best-performing model only managed to achieve a pass rate of 25.1%. Moreover, all other larger models achieved notably lower rates between 18.1% and 15.7%. The smaller models achieved pass rates below 1% and are not suited for the task. Syntax and compilation errors played a reduced role, likely because the models were less dependent on the provided context, as the fixes required for the tasks were always within a single method.

Answer for RQ3.5: Where in the APR benchmark do the highest-scoring models perform well and where is room for improvement?

The highest-scoring models Claude 3.5 Sonnet, GPT-4o, and DeepSeek V2.5 all had some problems with syntax and with the compilation step. However, all models had their largest drop due to failing test cases.
Claude 3.5 Sonnet passed more tasks than DeepSeek V2.5 and GPT-4o. Moreover, DeepSeek V2.5 seemed better at fixing bugs than GPT-4o but ultimately appeared to introduce more new bugs, leading to a lower pass@1 score. However, all three models introduced new bugs in some cases.

6.6. Model Ranking

In this part, we detail how we combined the results from the five evaluations to arrive at a single model that we then used in the practical exploration.

6.6.1. Method

To determine the best-performing model, we had to find a way to combine the results from the five benchmarks. For this, we needed to consider two main caveats: Firstly, the resulting values measure different things (e.g., pass rates versus n-gram overlap) and are, as such, not directly comparable. Secondly, while all results are, per definition, in a value range of $[0, 1]$, the achieved value ranges differed between the five comparisons (see Table 6.8). If we had combined the scores by computing the sum or the arithmetic mean, an evaluation with higher scores would have been overrepresented. In comparison, one with lower scores would have been underrepresented.

Therefore, we opted to use a ranking approach. To do this, we computed ranks for each benchmark, with each model receiving a rank from 1 (first) to 8 (last). If two models achieved precisely the same score, they were awarded the average rank of their places. This means that, for example, if the last two models achieved the same score, they were both given a rank of 7.5. The final rank was computed using the arithmetic mean of the ranks for each model.

Additionally, we wanted to evaluate if there were groups of models that differed significantly in their performance. For this, we aimed to use the Nemenyi posthoc test[13] which requires a statistical test that shows a significant difference as a precondition. We opted to us a Friedman test on the model scores in Table 6.8. Moreover, we used the following hypothesis and a p-value of 0.05:

H_0: *The ranks that the models achieved are the same across all aspects*

H_A: *The ranks that the models achieved differ across all aspects.*

We ran the Friedman test with an assumed Chi^2 distribution from the "scipy.stats"[14] package to evaluate this. Please note that it is not entirely clear whether using a Chi^2 assumption under these circumstances is acceptable. While the scipy package description would have required us to have more than 10 models **and** more than 6 aspects that we evaluated, Wikipedia[15] lists a requirement of more than 15 models **or** more than 4 aspects. Since we could not determine which restrictions were correct, we took the more beneficial one and assumed that having more than 4 aspects was enough.

Furthermore, we wanted to explore how the average rank and the model size are related. For this, we created a scatter plot to visualize the relationship. Since we could not verify the actual size for most commercial models, we had to fall back to assumptions we found online. For the Gemini 1.5 models, we assumed 32B (Flash)

[13]https://scikit-posthocs.readthedocs.io/en/latest/generated/scikit_posthocs.posthoc_nemenyi_friedman.html (Accessed: 14.12.2024)

[14]https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.friedmanchisquare.html (Accessed: 14.12.2024)

[15]https://en.wikipedia.org/wiki/Friedman_test (Accessed: 14.12.2024)

and 120B (Pro) parameters[16] respectively. For GPT-4o[17] and Claude 3.5 Sonnet[18], we assumed a parameter count of more than 175B. We plotted both as having a size of 175B. For DeepSeek V2.5, we do know the precise model size: 236B[19], and for all smaller models, the size is already in the name.

6.6.2. Results

Table 6.8 provides the relevant scores for each of the five evaluations. It can be seen that GPT-4o and Claude 3.5 Sonnet each achieved the highest score in two of five evaluations. Furthermore, Gemini 1.5 Flash ranked first for method comment generation.

When considering the value ranges, it can be observed that for four of the five benchmarks, the scores all stayed below 0.4. In contrast to that, in code completion, all five larger models scored above 0.58. This illustrates the need to choose a ranking approach that avoids over- and underrepresentation.

Based on these results, we computed the ranks for each evaluation and combined them to a final rank using the arithmetic mean. Table 6.9 shows the results. Based on this ranking process, we determined GPT-4o to be the best-performing model with a rank of 2.2, closely followed by Claude 3.5 Sonnet (2.5) and DeepSeek V2.5 (2.9). Moreover, the smaller Gemini 1.5 Flash (3.8) model ranked better than the Pro counterpart (4.0).

The small models ended up in the last places, with Qwen2 1.5B receiving a rank of 5.6, followed by Phi3 3.8B (7.4) and Qwen2 0.5B (7.6).

In addition to computing the combined rank for all models, we also computed the Friedman test on the model *scores* to evaluate whether they differed significantly. The test resulted in a p-value of **0.107**, which is larger than 0.05, leading us not to reject the null hypothesis. Therefore, we did not do a Nemenyi post-hoc test or provide a critical distance plot.

Figure 6.17 provides the average model scores with their (assumed) model sizes. Since not all models have publicly available parameter counts, we took estimates from various sources that we considered reasonable. The Figure shows that the rank generally improved with model size.

[16] https://www.reddit.com/r/LocalLLaMA/comments/1cxlsa9/comment/l53gga5/ (Accessed: 14.12.2024)
[17] https://aimlapi.com/comparisons/claude-sonnet-3-5-vs-chatgpt-4o (Accessed: 14.12.2024)
[18] https://felloai.com/2024/08/claude-ai-everything-you-need-to-know/ (Accessed: 14.12.2024)
[19] https://huggingface.co/deepseek-ai/DeepSeek-V2.5 (Accessed: 14.12.2024)

Model	Code Generation	Code Completion	Weighted Unit Test Generation	Method Comment Generation	APR
GPT-4o	0.3025	0.7917	0.2435	0.2855	0.3025
Claude	0.3950	0.7619	0.2284	0.2561	0.3950
Gemini Pro	0.2311	0.5833	0.1890	0.3085	0.2311
Gemini Flash	0.1975	0.6905	0.2012	0.3101	0.1975
DeepSeek V2.5	0.2899	0.7619	0.2090	0.3007	0.2899
Phi3 3.8B	0.0000	0.0119	0.0120	0.0897	0.0000
Qwen2 0.5B	0.0000	0.0000	0.0451	0.0290	0.0000
Qwen2 1.5B	0.0336	0.0179	0.1268	0.2878	0.0336

Table 6.8.: The relevant scores for each of the five evaluations. All scores are rounded to the 4th decimal place.

Model	Code Generation	Code Completion	Weighted Unit Test Generation	Method Comment Generation	APR	Rank
GPT-4o	2.0	1.0	1.0	5.0	2.0	2.2
Claude	1.0	2.5	2.0	6.0	1.0	2.5
DeepSeek V2.5	3.0	2.5	3.0	3.0	3.0	2.9
Gemini Flash	5.0	4.0	4.0	1.0	5.0	3.8
Gemini Pro	4.0	5.0	5.0	2.0	4.0	4.0
Qwen2 1.5B	6.0	6.0	6.0	4.0	6.0	5.6
Phi3 3.8B	7.5	7.0	8.0	7.0	7.5	7.4
Qwen2 0.5B	7.5	8.0	7.0	8.0	7.5	7.6

Table 6.9.: The rankings for each evaluation and the overall ranking.

6.6.3. Discussion

In this chapter, we compared the eight models and their performance in five aspects of Java SD and subsequently combined the results to determine the best-performing model. This model is **GPT-4o**, ranking first in code completion and unit test generation, second in code generation and APR, and fifth in method comment generation.

Moreover, the final rank unveiled three groups: The **best-performing models** with ranks from 2.2 to 2.9, followed by the two **Gemini models** (4.0-5.6) and the **small models** (5.6-7.6). It must be noted, however, that Qwen2 1.5B has a notably better rank (5.6) than the other small models (7.4-7.6).

Another noteworthy observation we made is that based on the rankings, DeepSeek V2.5 appears to be the most consistent model of the larger ones, reaching

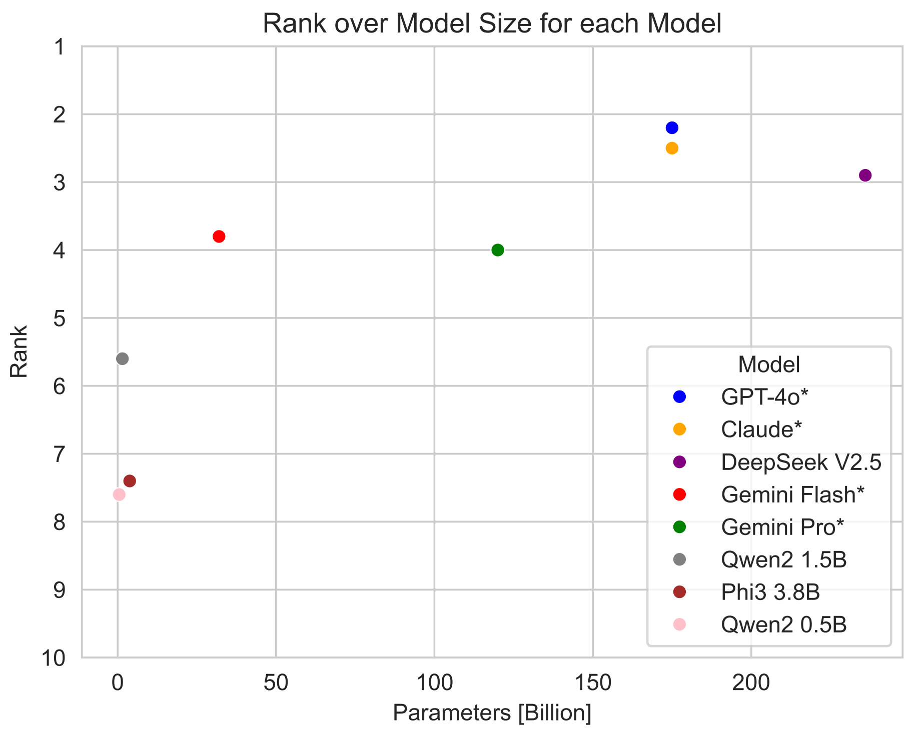


Figure 6.17.: **Rank over Model Size** for all models. Unverified model sizes are denoted with a star.

third place in all but one evaluation. However, even there, it shares second/third place with Claude 3.5 Sonnet.

Furthermore, we found the Gemini models to be the worst-performing larger models, with method comment generation being the only exception. Interestingly, the smaller Gemini 1.5 Flash scored better than the Gemini 1.5 Pro. While both models always were within one place of each other, Gemini Flash ranked above Gemini Pro in 3 of 5 benchmarks. One potentially contributing factor to the performance differences is their architecture. While Gemini 1.5 Pro is a ”sparse mixture-of-expert (MoE)” model, Gemini 1.5 Flash is a dense decoder model that is ”online distilled”[20] from the Pro variant (Team et al., 2024b).

[20]The following article explains the process in details: https://medium.com/google-developer-experts/online-knowledge-distillation-advancing-llms-like-gemma-2-through-dynamic-learning-e61c39280693 (Accessed: 23.01.2025)

The smaller models performed consistently worse than the larger ones, with the only exception being Qwen2 1.5B, which ranked fourth in method comment generation.

Given the fact that the models ranked rather consistently in all aspects but method comment generation, we deem it likely that this difference might be due to the evaluation differences. While the other aspects were primarily evaluated using a pass-fail measure (pass@1), method comment generation was evaluated with a similarity measure that compares how close the text is to a reference comment.

When considering whether models are open-weight or proprietary, it is interesting to see that DeepSeek V2.5, an open-weight model, scored in the top three consistently. This also means that it outperformed both Gemini models in 4 out of 5 aspects. Moreover, all other three open-weight models ranked in the bottom three places. However, this can likely not be attributed to open-weighted-ness but should arguably attributed to the comparatively small model sizes (under 4B parameters).

Another aspect under which to view the results is training data. While information on the code contained in training datasets is generally sparse, both Phi3 3.8B and DeepSeek V2.5 provide more detailed information. The authors of Phi3 3.8B mention[21] that code in the training dataset primarily consisted of basic Python code, which we consider as the likely cause for being outperformed by Qwen2 1.5B in all aspects. Furthermore, it is interesting to see that *DeepSeek-Coder-V2-Base*, the base model of DeepSeek V2.5, was trained on a dataset consisting of 60% code in 338 languages (DeepSeek-AI et al., 2024b). This could indicate that the model might also perform well on other languages.

Furthermore, knowledge cutoff dates (see Section 5.2) do not seem to be relevant to performance. This becomes clear when seeing that all known cutoff dates[22] lie between October 2023 and May 2024 (8 months). Moreover, the time range only reduces by one month (down to 7) when looking at just the three best-performing models.

While GPT-4o ranked the highest, we also deem Claude 3.5 Sonnet and DeepSeek V2.5 valuable software development tools as they achieved a rank close to GPT-4o. The small models, however, would likely be less suited to aid with software development. This becomes especially apparent when looking at the scores in Table 6.8. Moreover, we expect the experience with the Gemini models to be more involved than the other large models, although they would most likely still fare better than the small ones.

[21] `https://huggingface.co/microsoft/Phi-3.5-mini-instruct#responsible-ai-considerations` (Accessed: 22.01.2024)

[22] The cutoff dates for the two Qwen2 models are unknown.

Moreover, we could not to verify that the average scores differed significantly between the different aspects. Therefore, we could not do a Nemenyi test. Furthermore, we showed that the rank generally improved with larger model sizes. However, it must be noted that 4 of the 8 model sizes were assumptions taken from the internet, limiting the expressiveness of this plot.

Another important limitation that needs to be mentioned is that even though we evaluated five aspects of Java SD, these only cover some aspects of software development. Moreover, while providing a convenient way to evaluate and compare models, benchmarks, and datasets can only cover parts of the aspects they aim to address.

Therefore, to partially address this shortcoming and further understand of how the LLMs fared in practice, we evaluated GPT-4o in a software development scenario in the next chapter.

Overall answer for RQ2: How do the eight models compare on their benchmark performances?

The large models performed better than the smaller ones, with GPT-4o ranking the highest, closely followed by Claude 3.5 Sonnet and DeepSeek V2.5. The two Gemini models performed the worst out of the larger models, only surpassing the other three in method comment generation. Gemini Flash ranked higher than the larger Pro counterpart. The smaller models performed the worst in all benchmarks, except for Qwen2 1.5B, which ranked fourth in method comment generation.

Knowledge cutoff dates do not appear to impact the results of the comparison, given that all known cutoff dates (known for all models except Qwen2) are within 8 months of each other and the spanned date range only reduces to 7 months when considering the top three.

Moreover, it is worth mentioning that an open-weight model (DeepSeek V2.5) is under the three best-performing models, outperforming the proprietary Gemini 1.5 models. In part, this can likely be attributed to the fact that it's base model was trained on a dataset containing 60% code from 338 programming languages.

The minor performance differences between Gemini 1.5 Pro and Gemini 1.5 Flash might be partially explained by the fact that their architectures differ and that the Flash (dense encoder) variant is trained with the help of the Pro (Mixture of Experts) variant.

The low scores and partially nonsensical outputs for Phi3 3.8B across all five aspects can likely be attributed to the fact that primary programming language in the training dataset was Python.

Overall answer for RQ3: In which parts of the benchmarks do the highest-scoring models perform well and where is room for improvement?

In the **code generation** benchmark the highest-scoring models were capable of synthesizing syntactically correct code in almost all cases. Moreover, the models mostly failed due to compilation errors and test cases. For compilation errors the most relevant categories were **identifier**, **version** and **computation**.
For the **code completion** benchmark the highest-scoring models performed quite well with overall scores in the range of 75-80%. The models achieved high scores for control-flow completion and API function call completion and struggled most with block completion.
In the **unit test generation** benchmark, the highest-scoring models also performed well in regard to syntactical correctness. Furthermore, about half of all tasks failed due to compilation errors, with the most relevant error types being **identifier**, **version** and **computation**. Moreover, test case failures accounted for another 15-30% reduction in pass rate for the best-performing models.
For **method comment generation** all larger models and Qwen2 1.5B performed similarly leading to an overall F1-score between 0.256 and 0.310. Moreover, with the exception of Claude 3.5 Sonnet, which synthesized longer comments, the other well-performing models synthesized comments of similar length to the original comment. Overall, this benchmark does not lend itself well to this question as it does not provide detailed insights.
In the **APR** benchmark, the three best-performing models had some problems with all stages, failing up to about 20% of test cases due to syntax errors. However, the most pronounced drop was due to test cases failures, indicating that the ”corrected” code could be compiled but did not pass functional requirements. Compilation issues posed a less pronounced challenge, only accounting for up to 16,2% of errors among the highest-scoring models. Moreover, all of the best-performing models introduced new errors in at least some tasks.
Generally for code generation, test case generation and APR there is **room for improvement** both in creating compile-able code, specifically in the identifier and computation categories, as well as in creating functionally correct code. For code completion, the functional correctness could be improved (i.e. less errors in the ”Wrong Answer” category) and for method comment generation there is no clear room for improvement other than shorter comments for Claude 3.5 Sonnet.

Given the learnings from this chapter we expect that the IDE will likely be helpful in the practical exploration as it has the potential to detect compilation errors, like **identifier** and **computation** early, allowing for a swift mitigation. Furthermore, we conclude that we should extensively use tests, especially seeing as failing to meet functional requirements was also a common cause for failing tasks. Even though GPT-4o was the best-performing model, given the numeric results, we expect Claude 3.5 Sonnet and DeepSeek V2.5 to perform similar in practice.

7. Practical Scenario - Exploration

In this chapter, we detail our implementation following the assumed practical scenario where we used GitHub Copilot, an AI tool for SD powered by a GPT-4 version, to implement a small software project. This chapter differs from the others in this thesis as it primarily includes subjective experiences from the exploration. The first section (7.1) details how we conducted our exploration. This is followed by our experience, observations, and learnings, which are given in Sections 7.2 and 7.3. Section 7.2 details our experiences and learnings in a structure similar to Chapter 6. Section 7.3 illustrates how our prompting approach evolved and for which reasons. The final section (7.4) summarizes our key findings and takeaways.

7.1. Method

To better understand how well LLMs can aid with actual software development, we decided to implement a part of an expense-tracking application. We used GitHub Copilot for this, as it is built on top of a GPT-4 model. Unfortunately, we were not able to verify that Copilot uses GPT-4o, only that it uses a version of GPT-4[1]. However, given Open AI's pricing structure[2], we consider it to be likely that GPT-4o is used.

The application was already introduced in Chapter 4 and is a web application with a Next.js frontend and Spring Boot backend. The fact that we had prior experience with Spring Boot and Java and no prior experience with Next.js and React[3] allowed us to explore the role that familiarity with the used language plays.

The development for this application was done using IntelliJ IDEA as the IDE, with the Copilot Plugin[4] installed. Additionally, SonarLint[5] was installed to find more code issues than with the IDE-provided tools alone.

[1]According to OpenAI, Copilot is no longer powered by Codex but some non-disclosed version of GPT-4 (Source: support chat with OpenAI - Screenshots available on request)

[2]https://openai.com/api/pricing/ (Accessed: 17.12.2024)

[3]With the exception of having worked before, with Typescript, the underlying language.

[4]https://plugins.jetbrains.com/plugin/17718-github-copilot (Accessed: 14.12.2024)

[5]Now called SonarQube for IDE: https://plugins.jetbrains.com/plugin/7973-sonarqube-for-ide (Accessed: 14.12.2024)

During the evaluation, we primarily recorded our observations using a pen and paper, noting all the relevant observations and learnings we had made. Additionally, we ensured to record the screen and track IDE interactions to allow us to revisit key parts if needed. The screen was recorded using Open Broadcaster Software (OBS)[6], and the IDE interactions were tracked using CodeGRITS[7] (Tang et al., 2024b). For CodeGRITS, we had to adapt the plugin to support a current IDE version[8]. An illustration of our recording approach can be seen in Figure 7.1.

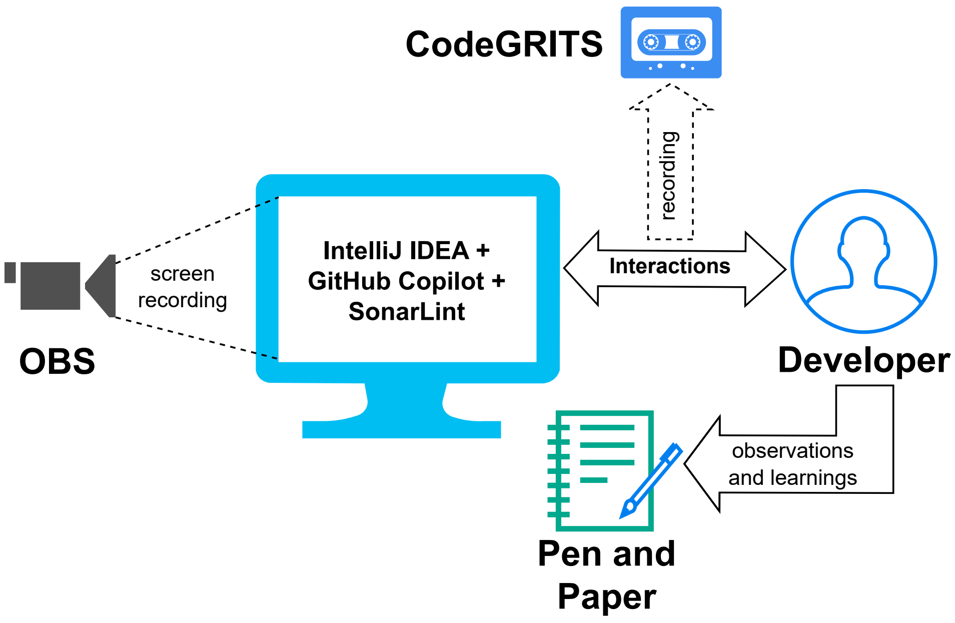


Figure 7.1.: **Recording Approach** we used for the practical exploration.

To conduct the evaluation in an ordered manner, we decided to use user stories, which we had to implement one after the other. A user story could be considered done when all required functionality was present, or we reached a point where we were blocked by a dependency to another, not yet implemented, user story. Additionally, the code had to be tested using test cases, achieving a coverage of at least 80%. We decided against requiring a coverage of 100% to avoid having to create test cases that do not have any meaningful assurances.

[6]`https://obsproject.com/` (Accessed: 28.11.2024)
[7]`https://github.com/codegrits/CodeGRITS` (Accessed: 28.11.2024)
[8]IntelliJ IDEA 2024.2.3

For the implementation, we started with an apparent over-reliance on GitHub Copilot, attempting to do everything using the tool and only falling back to manual editing if we had to. This behavior was subsequently adapted based on our experiences and learnings.

Since we were not sure how long the implementation of the application would take, using Copilot, we decided to restrict the time to be used for this exploration to 80 hours, the equivalent of two 40-hour work weeks. Moreover, if we had finished the project in under 40 hours, we would have had to develop further user stories. However, we ultimately ended up stopping the exploration at around 60 hours, as at that point, we had not made any notable or new observations for multiple hours, and it became clear that we would not be able to finish the application in 80 hours. Figure 7.2 shows a page of the application.

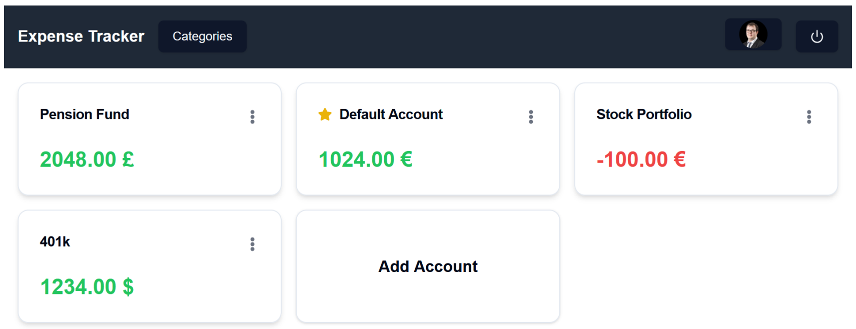


Figure 7.2.: **Screenshot** of the **Accounts Overview Page** of the expense tracking application.

7.2. Observations

In this section we describe our observations, starting with **six key observations**, followed by a more detailed description of five aspects that are similar to the ones in Chapter 6.

7.2.1. Key Observations

During our practical exploration, we made the following key observations when working with GitHub Copilot:

Observation 1: Good Starting Points We generally found GitHub Copilot to produce good starting points, usually in the form of code that was well readable and seemed correct. However, it was rarely the case that the code did exactly fit our needs and diligence was needed as not to overlook certain details or issues. Furthermore, the complexity of the task also played a role, with simpler tasks seemingly leading to better results than more complex ones. When dealing with complex tasks or ones Copilot did not seem to "understand" we found ourselves falling back to manual implementation efforts in multiple instances, due to poor tool performance.

Observation 2: Iterative Improvements When attempting to improve or adapt the code obtained from the tool, we found iterative changes to work well. Using this approach we found it important to (1) make one change after the other, (2) select the code in question (or stating the class) and (3) explicitly state or describe the desired changes. Using this approach, in contrast to changing many parts at the same time, we observed better results, likely due to the reduced complexity. Moreover, these smaller changes could then also be easier verified.

Observation 3: Suboptimal Code and Inconsistencies When using the tool, we found that the outputs sometimes contained suboptimal code and architectural decisions that would lead to errors at runtime or simply not follow best practices. Moreover, the results obtained from Copilot would at times be inconsistent. This included using different testing approaches for a similar task, removing already existing comments or using the wrong libraries.

Observation 4: "Inspiration" for Alignment and Correctness Another key observation we made was that adding or referencing files with similar or related code as an source of "inspiration" to the prompt noticeably improved the results obtained. This held true both in terms of alignment (how the task was solved) as well as correctness. Furthermore, we observed explicitly adding a related file to be more reliable than simply referencing it, likely because this step eliminates a potential source of ambiguity.

Observation 5: Test Case Issues For test case generation we observed the tool to struggle quite a bit. While the test cases that were generated looked quite good and covered the majority of the function in question, they would often lack checks for calls to mocked methods. Moreover, when prompted to create test cases for a specific method, it would frequently miss at least one test condition or branch. Furthermore, the tests for the frontend were notably worse than the ones for the backend, primarily due to issues with reactive state management and UI interactions. Overall, due to these issues, we observed test case generation to be the most time-consuming part of this exploration.

Observation 6: New Language Development Speed Another key observation we made was about using GitHub Copilot for working with React/Next.js, a framework/language, we had no prior experience with[9]. There we found that the tool generally allowed us to get started faster than we likely would have been otherwise. However, this benefit became a liability in cases where the model was not able to fix a problem and we were left with tracking down and solving the issue ourselves. These situations proved to be quite time consuming as we had little experience and knowledge about the code and the intricacies of the underlying concepts.

Having highlighted the six key observations the following parts provide an overview of observations categorized using five aspects similar to the one in Chapter 6. The related key observations are referenced using **(O1)-(O6)**:

7.2.2. Code Generation

We found generated application code look good generally, but to contain flaws in the details **(O1)**. These included, but were not limited to, using recently deprecated methods of libraries, having challenges adding specific libraries correctly, and suggesting suboptimal architecture **(O3)**.

[9]We just had experience with JavaScript and Typescript but not with React or Next.js.

Backend

For the development of the **backend**, it was most notable that the code followed the same overall schema for REST API endpoints and database accesses, only differing in the details. An example of this can be seen in Listing 1. Therefore, somewhat unsurprisingly, the model performed quite well, having usually at least one similar example within the same file **(O4)**. However, we sometimes found the model to omit certain safety checks, requiring additional prompts to add them **(O2, O3)**.

Listing 1 Example of an REST endpoint. This snippet shows all logic needed for the delete operation.

```
/**
 * Endpoint to delete the authenticated user's account.
 *
 * @param userDetails the authenticated user's details
 * @return a response entity with the status of the deletion
 */
@Transactional
@DeleteMapping("/user")
public ResponseEntity<String> deleteUser(@AuthenticationPrincipal UserDetails userDetails) {
    if (userDetails == null) {
        return new ResponseEntity<>(HttpStatus.UNAUTHORIZED);
    }
    User user = userRepository.findByUsername(userDetails.getUsername());
    if (user == null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    userRepository.delete(user);
    return new ResponseEntity<>("User account deleted successfully", HttpStatus.OK);
}
```

Moreover, the model also struggled with correctly setting up the security config **(O3)**. It initially used the now deprecated approach to configure access to the application. While this could partially be fixed by letting the model improve the code in a multi-turn approach, it could not correct all errors **(O1, O2)**. We theorize that this is in part due to the deprecating update being relatively new, with the corresponding blog post being released in February 2022[10] and most existing articles, blog posts, or Stack Overflow questions most likely not being updated.

Furthermore, we observed the model to not follow best practices in some instances, by, for example, creating fields that are annotated with "@Autowired"[11] instead of adding them as a parameter to the class constructor **(O3)**.

Moreover, the model would sometimes not practice separation of concerns by putting the REST call of different entities into the same class **(O3)**. However, we

[10]https://spring.io/blog/2022/02/21/spring-security-without-the-websecurityconfigureradapter (Accessed: 14.12.2024)
[11]https://medium.com/devdomain/spring-boots-autowired-vs-constructor-injection-a-detailed-guide-1b19970d828e (Accessed: 14.12.2024)

only observed this for the first REST endpoint for an entity.

Another issue we found to occur multiple times was that entities that had a relationship (i.e., 1:n) to another entity would sometimes be serialized circularly, leading to an endless loop that only stopped due to a stack overflow. This could have been prevented by the consistent use of data transfer objects (DTOs), which would have given the explicit possibility to specify which fields should be serialized. However, Copilot created and used DTOs inconsistently or naively **(O3)**.

In general, we found Copilot to help speed up development, especially in this scenario, where we already had experience with the tech stack. We often had to fix more complex issues, like serializing the circular dependencies ourselves **(O1)**.

Frontend

For the frontend, we observed that the model had difficulty properly providing the correct commands to install shadcn, the UI library, and jest, the testing library. This prompted us to add shadcn components to the project manually. One reason the model could have had difficulties producing the correct commands is that shadcn is relatively new and follows a different philosophy where the developer inherits the entire code for the UI component using an install command.

Furthermore, we found Copilot to struggle with more complex uses of shadcn components, especially with forms **(O1)**. One way to improve the generated code was to identify existing components, for example an existing form, and to explicitly point the model to it for inspiration (see 7.3). Code generated using this approach was usually notably better and better aligned with our intentions than if we had prompted it from scratch **(O4)**.

Moreover, we found GitHub Copilot helpful for changing details, especially related to styling. There, it saved us time by allowing us to iteratively describe the visual behavior we wanted to achieve **(O2)**. Regarding React Hooks, we had many instances where the conditions for hook to be re-evaluated were incorrectly set, leading to endless re-evaluations **(O3)**.

In addition, Copilot often failed to grasp the context, leading to it using the Next.js server package to implement navigation on a client page or omitting the ”use client” keywords required to execute certain features **(O1,O3)**.

Overall, we found GitHub Copilot to be helpful, especially for quickly starting with a new language. However, our lack of knowledge about the intrinsics of React and the tendency of Copilot to use the wrong libraries, introduce small inconsistencies or omit keywords made SD with Next.JS quite tedious and time-consuming **(O3,O6)**. We believe we would have benefited greatly from prior knowledge and experience with this framework (React/Next.js) **(O6)**.

7.2.3. Code Completion

Code completion is the only one of the five aspects we had ample prior experience with, as we previously used the "Full Line code completion"[12] by JetBrains. Therefore, most of the usage of Copilot's line completion capability was handled intuitively.

Our experience with Copilot line completion was not noticeably better or worse than that of the model by JetBrains. Moreover, we observed it to generally produce good suggestions in cases where the intended action was clear. Furthermore, the line completion suggestions were non-intrusive and could readily be accepted or ignored. We found line completion to work especially well for repetitive tasks like calling multiple setters, completing function calls, or completing boilerplate code.

Code completion is the only aspect where we were able to collect meaningful data using CodeGRITS, which tracked the number of completion suggestions that were accepted or rejected. However, we only have data for around 85% of the evaluation duration due to the IDE crashing multiple times during this implementation phase. However, we were shown 168 suggestions in this time-frame, 160 of which we accepted. This number is not a suitable proxy to gauge the usefulness of suggestions. Firstly, since we started this evaluation with an over-reliance on generating code rather than completing it, the number of suggestions shown is relatively small. Secondly, we observed our tendency to instinctively accept all suggestions that do not seem completely wrong, as we expected to need to make some alterations either way. Therefore, the acceptance rate is likely not indicative of the usefulness of the suggestions.

In general, we found the tool to be unintrusive and a helpful addition to the completions already present in a traditional IDE.

7.2.4. Test Case Generation

For test case generation, our experience with using GitHub Copilot was mixed. While it seemed generally capable of generating test code that looked good on the surface, it often got details wrong or missed essential parts **(O1, O3, O5)**.

Backend

For the backend, the model was usually capable of identifying and synthesizing test cases that captured a considerable amount of the logic contained within a function. Although these test cases appeared great on the surface level, verifying the inputs

[12] https://www.jetbrains.com/help/idea/full-line-code-completion.html (Accessed; 14.12.2024)

and outputs of a function, the same cannot be said of side effects. While the tool correctly employed mocking of related classes, to test a method in isolation, it did not verify whether the correct methods of these classes were actually called **(O5)**. Listing 2 shows an illustrative example of this.

Listing 2 Example of a generated test case. The comment indicates the type of check that would often be missing.

```java
@Test
@WithMockUser(username = "testuser")
void testCreateDefaultAccount_Success() throws Exception {
    User user = new User();
    when(userRepository.findByUsername("testuser")).thenReturn(user);
    when(accountRepository.findByUserAndIsDefault(user, true))
        .thenReturn(null);

    AccountRequest request = new AccountRequest();
    request.setName("Default Account");
    request.setStartingBalance(BigDecimal.valueOf(1000));
    request.setCurrency(Currency.USD);
    request.setDefault(true);

    mockMvc.perform(post("/account/create")
                    .contentType(MediaType.APPLICATION_JSON)
                    .content(new ObjectMapper().writeValueAsString(request))
                    .with(csrf()))
            .andExpect(status().isOk())
            .andExpect(content().string("ommitted for brevity"));

    // Checks, like the one below, would often be missing
    verify(accountRepository, times(1)).save(any(Account.class));
}
```

Moreover, we found that Copilot tends to miss at least one branch in most cases. However, in general, this could be discovered using test coverage and resolved by selecting the not yet covered part of the code and prompting the model to generate a test case for the selection **(O2, O4, O5)**.

Another observation we made was that the tool frequently missed tests for non-explicitly documented requirements **(O5)**. An example of this was the requirement that users may only edit their own categories. While this was reflected in the code by the fact that the repository method did explicitly attempt to retrieve the category for a user, there was usually no test case verifying that no categories from another user were returned. While such a shortcoming can easily be overlooked, it was typically straightforward to rectify once discovered by prompting the model to extend the test suite by a test case for this exact situation **(O2)**.

Moreover, we found GitHub Copilot inconsistent in how it generated test cases **(O3)**. While for some endpoints, it would use MockMVC, a way of testing the entire endpoint including, but not limited to, serialization and de-serialization, in some cases it merely called the method in question.

However, in situations where test cases for similar code already existed in the same file, we found Copilot to perform notably better, leading to better results on the first prompt **(O4)**.

GitHub Copilot also provides a ”/tests” command instructing the model to generate unit tests for the selected code. However, we found ourselves primarily using explicit prompts, which were more in line with the prompting we used in other parts of the exploration.

Overall, Copilot was helpful for quickly identifying and writing required test cases for Java. However, vigilance was needed to ensure the test cases verified all the necessary aspects **(O1, O6)**.

Frontend

For the frontend, we found test generation to be very time-consuming **(O6)**. While the concept of the test cases usually made sense in general, the specific evaluations often contained faults **(O1)**. This was especially prevalent for testing interactions with shadcn UI components and test cases that utilized reactive behavior **(O5)**. Copilot frequently seemed to not ”understand” how to correctly interact with these components using Jest. Common issues encountered were ambiguous UI element selectors, not waiting for state updates, problems with mocking, and incorrectly imported libraries **(O5)**. This led to even ”simple” tests taking quite a long time.

Tracking down some of these issues, especially the state updates and mocks was tedious, time-consuming, and hindered by the fact that we had no prior experience with React or Jest. We were led astray multiple times by Copilot and had to consult the documentation frequently. Although we believe that with more experience and understanding, the benefits would outweigh the drawbacks, this is not the case without prior knowledge **(O5, O6)**.

Moreover, Copilot would almost always miss scenarios when testing, or it would import the wrong libraries **(O3, O5)**.

Summary

We found GitHub Copilot to provide sensible starting points in most cases **(O1)**. However, almost all test suites generated required manual rework or, at the very least, manual verification **(O2)**. While we can see this tool being helpful for test case generation, there is an obvious need for diligence on the side of the programmer to not overlook shortcomings of the generated test cases. Moreover, to get the most utility from Copilot, the user should be experienced with the intricacies of the language and framework as well as testing in that language. Otherwise, the process can be pretty cumbersome **(O6)**.

7.2.5. Comment Generation

While the complexity of the code was not that high due to the utilization of frameworks, we nonetheless wanted to explore Copilot's comment generation abilities.

In the beginning, we did not explicitly prompt GitHub Copilot to create comments for the code it generated. In most cases, this led to code without any comments. Once we adapted the prompting style by adding sentences like **"Provide comments for the generated code."** or **"Comment the code you generate."**, the outputs contained comments. They usually contained either line comments (see Listing 3) or JavaDoc comments (see Figure 4), but rarely both. We generally found the comments to be accurate summarizations of the code. However, we did not require the comments as the code was already easy to read and understand.

Moreover, we found that Copilot sometimes removes already existing comments from the code. This primarily occurred when the conversation did not contain a part that instructed the model to generate comments **(O3)**.

Listing 3 Example of a code snippet with normal comments.

```java
@GetMapping("/{accountId}")
public ResponseEntity<AccountResponse> getAccountById(
    @AuthenticationPrincipal UserDetails userDetails,
    @PathVariable Long accountId) {

    // Check if the user is authenticated
    if (userDetails == null) {
        return new ResponseEntity<>(HttpStatus.UNAUTHORIZED);
    }
    // Find the user by username
    User user = userRepository.findByUsername(userDetails.getUsername());
    if (user == null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    // Find the account by ID
    Optional<Account> optionalAccount = accountRepository
        .findByIdAndUser(accountId, user);
    if (optionalAccount.isEmpty()) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    Account account = optionalAccount.get();
    // Return the account details
    return new ResponseEntity<>(AccountResponse.fromAccount(account),
        HttpStatus.OK);
}
```

Listing 4 Example of a code snippet with a JavaDoc comment.

```java
/**
 * Endpoint to change the password of the authenticated user.
 *
 * @param userDetails the authenticated user's details
 * @param request     the request containing the old and new passwords
 * @return a response entity with the status of the operation
 */
@Transactional
@PutMapping("/user/password")
public ResponseEntity<String> changePassword(
    @AuthenticationPrincipal UserDetails userDetails,
    @RequestBody ChangePasswordRequest request) {

    if (userDetails == null) {
        return new ResponseEntity<>(HttpStatus.UNAUTHORIZED);
    }
    if (request.getNewPassword() == null ||
        request.getNewPassword().length() < 6) {

        return new ResponseEntity<>(
            "New password must be at least 6 characters long",
            HttpStatus.BAD_REQUEST);
    }
    User user = userRepository.findByUsername(userDetails.getUsername());
    if (user == null) {
        return new ResponseEntity<>(HttpStatus.NOT_FOUND);
    }
    if (!passwordEncoder.matches(request.getOldPassword(), user.getPassword())) {
        return new ResponseEntity<>("Old password is incorrect",
            HttpStatus.BAD_REQUEST);
    }
    user.setPassword(passwordEncoder.encode(request.getNewPassword()));
    userRepository.save(user);
    return new ResponseEntity<>("Password changed successfully",
        HttpStatus.OK);
}
```

7.2.6. Automated Program Repair and Code Adaption

Since most of the code produced by GitHub Copilot had at least minor issues or needed to be adapted further to suit our needs, we had ample opportunities to explore the bug-fixing and code-altering capabilities of the tool **(O2)**.

We generally observed bug fixing to work well in low complexity and common scenarios. However, we found it to perform worse for more complex problems where the error message or the error was not clear **(O1)**.

While we found the iterative improvement steps from **(O2)** to work well, we sometimes also employed suggestive questions as special source of ”inspiration” **(O4)**, when the model did not correctly identify the error with the classical approach, and manual editing would have been tedious. This usually led to correct results.

Another approach we sometimes used was the "/fix" command of Copilot, which instructs the model to identify the current error(s) and to fix them. Additionally, a description of the error or fix could be provided. While this worked sometimes, we also had cases where the model would describe correct code as erroneous and would attempt to fix it **(O3)**. While we occasionally observed this behavior using normal prompting, it seemed to occur more often when using the "/fix" command. This led us to primarily narrow down the potential errors with explicit prompting **(O2)**.

Towards the end, we made smaller changes ourselves, as we felt slowed down by waiting for the model to complete an output for the entire class just to change one minor detail. However, we still used it for more complex situations where more code needed to be changed to mitigate an error or adapt the code, as it saved quite some time **(O2, O6)**. Moreover, since the Copilot plugin for IntelliJ does not offer a diff view between the output and the currently open file, we found it tremendously helpful to have files fully committed before starting with changes.

7.3. Prompt evolution

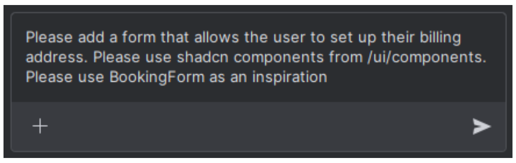
We started our exploration with an over-reliance on the chat functionality of GitHub Copilot. For this, we used a simple initial prompt, which we evolved over time. This section details key evolution steps and learnings. We primarily focused on straightforward prompts here, as this is also what is shown in the examples on the GitHub Copilot webpage[13].

The prompt we started with was a simple and naive approach: **"I need to implement the following user story "As a user I want to be able to delete my account"".** However, this prompting style produced suboptimal results, likely because it was a very high-level description that did not communicate any additional assumptions. As the exploration progressed, the prompting approach got more sophisticated.

The first step in this evolution was to get more specific by detailing the required functionality and assumptions: **"I need to create the functionality to log out of my application. It should be a button that is shown in the Nav Bar to the right of the username. It should log the user out and forward to /".** This prompt improved the alignment of the code with our intentions by eliminating ambiguities through explicitly stating requirements.

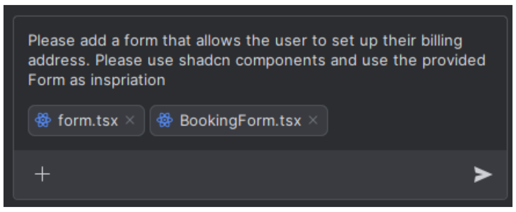
Another adaptation we explored was to prepend the prompt with a text like: **"You are a professional Java/Spring[14] developer"**. However, there was no no-

[13]https://github.com/features/copilot (Accessed: 22.01.2024)
[14]Or: React/Next.js



Please add a form that allows the user to set up their billing address. Please use shadcn components from /ui/components. Please use BookingForm as an inspiration

(a) Implicit



Please add a form that allows the user to set up their billing address. Please use shadcn components and use the provided Form as inspriation

(b) Explicit

Figure 7.3.: **Similar Examples** approach, which points the tool toward similar samples.

ticeable improvement in the responses from the model. Moreover, since the examples given on the Copilot quickstart page[15] do not contain this type of prefix either, we assumed that it is not necessary here to improve performance further.

One strategy we found to work quite well, and therefore ended up using a lot, was pointing the tool to similar existing code snippets. To do this, we used two different strategies: (1) mentioning the name of a class that contains similar code; and (2) explicitly adding the file to the prompt. Illustrative examples of this strategy are given in Figure 7.3. Overall, we found that explicitly adding the files achieved better results than just mentioning the classes.

Another strategy we explored was adding the task of writing comments and testing the code to the initial prompt. This worked reasonably well, and we did not notice differences in the quality compared to creating separate prompts. Moreover, we noticed that Copilot would sometimes remove already existing comments when the initial prompt did not contain an explicit instruction to write comments.

Towards the end of the exploration, a typical initial prompt looked like this: **"Please extend the BookingController to allow the user to retrieve all their bookings in chronological order. You should use pagination. You create exhaustive tests for the code you generate. You comment the code you generate".**

To adapt, improve, and correct code, we usually re-used the already existing conversation and described the desired changes or observed errors in plain language. Furthermore, we sometimes also pasted the error and stack trace we received into the conversation. This usually worked well, likely because the conversation already contained ample context information. Moreover, we experienced that once answers from a conversation started to become inaccurate or, in any other way, clearly suboptimal, it was usually best to start a new conversation instead of attempting to "fix" the current one.

[15]https://docs.github.com/de/copilot/quickstart (Accessed: 15.12.2024)

In general, if we encountered a new situation, we used clear and short prompts to convey all the required information.

7.4. Summary

In this chapter, we explored using GitHub Copilot to implement part of a software project. We found that using Copilot usually worked reasonably well up to a certain complexity, above which the process would get tedious **(O1)**. However, almost all user stories required iterative refinement or correction to achieve the desired results **(O2)**. This not only included functional correctness but also suboptimal code and inconsistencies **(O3)**. While we found this to work well most of the time, sometimes fixing the code manually was faster and more intuitive. Moreover, we observed Copilot as especially good for clear and repetitive tasks. One example of this is to adapt an existing test suite from calling functions normally to using MockMvc, which would have been tedious to do by hand **(O2)**.

For more complex situations, the tool usually provides a good starting point **(O1)**. However, both in general and especially in more complex situations, it was essential to verify the code as it could, among other things, contain (1) suboptimal architectural decisions that did not align well with the application, (2) insecure correctness assumptions (e.g., expecting that a security-relevant part is always configured correctly), or (3) test cases that appear to cover a specific situation but do so incompletely.

Moreover, test cases posed a challenge for the tool **(O5)**. This was especially notable, for mocking, UI interactions and reactive state management. Moreover, the model frequently missed at least one important test case.

For the difference between a known and an unknown framework, we found that for the unknown framework (React), we could start much faster than we likely would have been without using an AI tool **(O6)**. However, this meant that we ran into unexpected issues in situations where the model failed to produce the desired code, and we were left with implementing or fixing the code ourselves. In contrast, we found that our experience with Java and Spring helped us spot potential issues early and were, therefore, able to mitigate them, before becoming a real problem. Based on our experience, we would expect a person with previous experience to benefit more from Copilot than a person without.

Moreover, while code completion played a reduced role in this implementation, we found it easy to apply potentially fitting code and ignore unwanted code.

While we found comment generation to generally create good comments, they were not needed in this project as the code was usually not that complex and well-readable.

For prompting, we found it best to use explicit and concise prompts to achieve a good result. In cases where the model did not produce the desired output, we found it helpful point it to relevant or similar files explicitly **(O4)**. Another approach was to start a fresh conversation once results started to became suboptimal.

Summing up, we found that after some time, we started to get an intuition of what would likely work well and what would not.

Answer to RQ4: To what extent can GitHub Copilot aid in implementing a software project?

Generally, GitHub Copilot can aid in all implementation steps; however, the utility is highly dependent on the complexity. Moreover, we found it crucial to have a solid understanding of the languages and frameworks used to ensure that issues can be detected and remediated early on. Code generated by Copilot often contained smaller issues and occasionally larger problems. If the tool "understood" the code, these could usually be remediated using iterative prompting. However, vigilance is essential when using this LLM to avoid missing errors.

8. Conclusions

In this thesis, we explored how, given a practical SD scenario, suitable LLMs can be selected, how they can be compared to select a single model, and how well the best-performing model can aid in implementing the software project from the scenario. We selected eight models based on key requirements we determined based on the scenario and quantitatively compared and ranked these models based on five aspects of Java SD using benchmarks. These aspects are code generation, code completion, unit test generation, method comment generation and automatic program repair. Using this approach, we found GPT-4o to be the best-performing LLM and, therefore, selected GitHub Copilot, which is based on GPT-4(o), as the tool to aid with implementation efforts. With Copilot, we conducted a qualitative evaluation by implementing part of the expense tracking application until we ceased making new and notable observations. We started the evaluation with an over-reliance on conversations with GitHub Copilot, successively reducing the reliance if we found the tool not helpful for certain tasks.

In the **model comparison step**, we found that for four out of five aspects of Java SD, the top 3 comprised the same three models: GPT-4o, Claude 3.5 Sonnet, and DeepSeek V2.5. The only exception was method comment generation, where only DeepSeek V2.5 scored in the top 3. Moreover, a similar pattern was observed on the lower end of the scores - there the three smaller models consistently ranked last, with method comment generation being the only exception. This suggests that the smaller models might not be well suited for aiding Java SD. For Phi 3 3.8B, the fact that it was not trained on Java or JavaScript code, just basic Python code, might explain part of its low performance.

We theorize that the difference in comment generation stems from a differing evaluation approach. While the other benchmarks were generally evaluated in a pass-fail fashion (using pass@k), the comments were scored on how similar they were to a reference comment. This was done using ROUGE-1.

Moreover, during the model comparison, we found that while syntactical correctness did not appear to be a major problem for the best-performing models, the compilation and functional correctness were. This was observed in all three aspects that explicitly evaluated the steps of syntax, compilation, and functional correctness[1]. For compilation failures, we were able to categorize them and found the most common error categories to be **identifier**, **computation**, and **version**. However, the **version** errors cannot necessarily be attributed to the model. This is due to the fact that some tasks in the benchmarks use old Java versions, that do not support often-used language features. This is a caveat that the LLM is not aware off and unfortunately this cannot be mitigated as the underlying benchmark does not report the used Java versions of each task.

For the **implementation of the software project** in the scenario, in which we used GitHub Copilot as the tool surrounding GPT-4(o)[2] we observed the tool to generally be able to help to some degree in all five aspects of Java SD. However, this helpfulness also depended on the task's complexity and the LLM's capability to "understand" the task at hand. Furthermore, we found that the generated code regularly contained smaller issues and occasionally larger ones. This required vigilance when working with GitHub Copilot. In general, we observed that the result could be notably improved by pointing the model explicitly (by adding files to the prompt) to relevant or similar code. Moreover, we found the line completion in the tool to be good when the completions appeared correct and non-intrusive otherwise. Additionally, we observed test case generation with Copilot to be cumbersome at times due to the tool's problems with mocking, UI interactions and reactive state management. Furthermore, GitHub Copilot tended to miss test cases. In contrast to that, fixing and adapting code usually worked well as long as the model had an accurate "understanding" of the situation. In our scenario, we found comment generation not to be necessary due to the relatively simple and readable code. Nonetheless, the comments the tool generated seemed sensible and were well-readable.

Moreover, writing the frontend using a JavaScript framework (React), which we had no prior experience with, illustrated the importance of having a solid understanding of the technologies one is working with. While Copilot enabled us to start faster than we would likely have been otherwise, it also slowed us down significantly at times when the tool was unable to solve a problem and we did not yet have the knowledge and experience required to swiftly solve the issue.

Considering the entire implementation phase (frontend and backend), we do not feel that using GitHub Copilot made us significantly faster, as especially the issues with frontend tests did cost us a lot of time. However, limiting the scope to the backend part (Java), we would estimate that GitHub Copilot increased our speed by about 20-30%.

[1]In terms of passing test cases

[2]While we cannot confirm that GitHub Copilot uses GPT-4o as the underlying LLM, a support chat with OpenAI confirmed that they use an undisclosed version of GPT-4. Due to the pricing of both GPT-4 and GPT-4o we deem it likely that they use the 4o version.

Moreover, to prompt the model, we found it essential to make implicit assumptions explicit. This was usually done by clearly stating the desired change with some detail rather than describing it more generally. That approach was typically accompanied by explicitly adding and mentioning relevant files to the prompt to ensure the model produced better code, even in more complex scenarios. Moreover, we did not notice any changes in the results when starting the prompt with ”You are a professional <language|framework> developer...” leading us to assume such a prompting approach is either not necessary, or already takes place internally.

Considering the **observations from both the quantitative model comparison and the qualitative exploration**, we found the compilation errors to be less present in the practical exploration. This was likely due to the IDE and SonarLint being able to detect and highlight potential issues in most cases. Furthermore, issues with functional correctness were also observed in the exploration - these ranged from minor issues to more fundamental problems like the incorrect use of libraries. This was especially noticeable when using mocking or UI interactions in test cases. However, the latter might also be attributed to the novelty and the special approach of the UI library we used. There, one inherits the component's code instead of just using it. For code completion, we cannot definitively say if there was a difference between the performance during the model comparison and the practical exploration, as we primarily used chat-based interactions. Furthermore, when we used the completion capability, we did so intuitively as we had exhaustively used JetBrains' line completion before. Moreover, the task of method comment generation does not lend itself well to be compared as it is not a classic pass-fail approach, and we can, therefore, not easily compare the results from the quantitative comparison with the qualitative exploration.

Future work in this field may focus on evaluating models on all languages for a project, potentially even leading to the selection of different models for each programming language. It would furthermore be interesting to conduct an exploration with multiple programmers and models, for example, using the three best-performing models from our evaluation, to see whether the benchmark differences and rankings are indicative of changes in the experience in practice. Moreover, our exploration consisted of a greenfield (new) project, which, based on our experience, can be significantly easier to work with than a brownfield (existing) project. However, exploring the latter could help clarify up to which complexity using a tool like GitHub Copilot makes sense and where boundaries of the model's capabilities start to become a hindrance.

Summing up, while an LLM can be helpful for software development, it is not without pitfalls, especially as minor errors or missing test cases can be easily overlooked. Furthermore, we consider it highly beneficial to have prior knowledge of the language for which one uses the LLM, as it allows oneself to detect problems earlier and makes manual intervention easier when it is eventually needed.

This thesis was revised using Grammarly[3]. However, neither Grammarly's "Generative AI" feature nor any other LLM was used to write the text for this thesis.

[3] https://www.grammarly.com/ (Accessed: 06.01.2025)

Bibliography

M. Abdin, J. Aneja, H. Awadalla, A. Awadallah, A. A. Awan, N. Bach *et al.*, "Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone," Aug. 2024, arXiv:2404.14219 [cs]. [Online]. Available: http://arxiv.org/abs/2404.14219

M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *2013 10th Working Conference on Mining Software Repositories (MSR).* San Francisco, CA, USA: IEEE, May 2013, pp. 207–216. [Online]. Available: http://ieeexplore.ieee.org/document/6624029/

M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A Survey of Machine Learning for Big Code and Naturalness," *ACM Computing Surveys*, vol. 51, no. 4, pp. 1–37, Jul. 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3212695

Anthropic PBC, "Introducing the next generation of Claude," 2024. [Online]. Available: https://www.anthropic.com/news/claude-3-family

O. Asare, M. Nagappan, and N. Asokan, "A User-centered Security Evaluation of Copilot," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering.* Lisbon Portugal: ACM, Apr. 2024, pp. 1–11. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3639154

B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan *et al.*, "Multi-lingual Evaluation of Code Generation Models," Mar. 2023, arXiv:2210.14868 [cs]. [Online]. Available: http://arxiv.org/abs/2210.14868

J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan *et al.*, "Program Synthesis with Large Language Models," Aug. 2021, arXiv:2108.07732 [cs]. [Online]. Available: http://arxiv.org/abs/2108.07732

D. Bahdanau, K. Cho, and Y. Bengio, "Neural Machine Translation by Jointly Learning to Align and Translate," May 2016, arXiv:1409.0473 [cs]. [Online]. Available: http://arxiv.org/abs/1409.0473

S. Barke, M. B. James, and N. Polikarpova, "Grounded Copilot: How Programmers Interact with Code-Generating Models," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 85–111, Apr. 2023. [Online]. Available: https://dl.acm.org/doi/10.1145/3586030

N. Boulanger-Lewandowski, Y. Bengio, and P. Vincent, "Modeling Temporal Dependencies in High-Dimensional Sequences: Application to Polyphonic Music Generation and Transcription," Jun. 2012, arXiv:1206.6392 [cs]. [Online]. Available: http://arxiv.org/abs/1206.6392

T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal *et al.*, "Language Models are Few-Shot Learners," Jul. 2020, arXiv:2005.14165 [cs]. [Online]. Available: http://arxiv.org/abs/2005.14165

M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.* Amsterdam The Netherlands: ACM, Aug. 2009, pp. 213–222. [Online]. Available: https://dl.acm.org/doi/10.1145/1595696.1595728

M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan *et al.*, "Evaluating Large Language Models Trained on Code," Jul. 2021, arXiv:2107.03374 [cs]. [Online]. Available: http://arxiv.org/abs/2107.03374

W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li *et al.*, "Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference," Mar. 2024, arXiv:2403.04132 [cs]. [Online]. Available: http://arxiv.org/abs/2403.04132

K. Cho, B. Van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk *et al.*, "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP).* Doha, Qatar: Association for Computational Linguistics, 2014, pp. 1724–1734. [Online]. Available: http://aclweb.org/anthology/D14-1179

V. Debroy and W. E. Wong, "Using Mutation to Automatically Suggest Fixes for Faulty Programs," in *2010 Third International Conference on Software Testing, Verification and Validation.* Paris, France: IEEE, 2010, pp. 65–74. [Online]. Available: http://ieeexplore.ieee.org/document/5477098/

DeepSeek-AI, A. Liu, B. Feng, B. Wang, B. Wang, B. Liu *et al.*, "DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model," Jun. 2024, arXiv:2405.04434 [cs]. [Online]. Available: http://arxiv.org/abs/2405.04434

DeepSeek-AI, Q. Zhu, D. Guo, Z. Shao, D. Yang, P. Wang *et al.*, "DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence," Jun. 2024, arXiv:2406.11931 [cs]. [Online]. Available: http://arxiv.org/abs/2406.11931

M. Denkowski and A. Lavie, "Meteor Universal: Language Specific Translation Evaluation for Any Target Language," in *Proceedings of the Ninth Workshop on Statistical Machine Translation.* Baltimore, Maryland, USA: Association for Computational Linguistics, 2014, pp. 376–380. [Online]. Available: http://aclweb.org/anthology/W14-3348

J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *Proceedings of the 2019 Conference of the North.* Minneapolis, Minnesota: Association for Computational Linguistics, 2019, pp. 4171–4186. [Online]. Available: http://aclweb.org/anthology/N19-1423

Y. Ding, Z. Wang, W. U. Ahmad, H. Ding, M. Tan, N. Jain *et al.*, "CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion," Nov. 2023, arXiv:2310.11248 [cs]. [Online]. Available: http://arxiv.org/abs/2310.11248

X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen *et al.*, "ClassEval: A Manually-Crafted Benchmark for Evaluating LLMs on Class-level Code Generation," Aug. 2023, arXiv:2308.01861 [cs]. [Online]. Available: http://arxiv.org/abs/2308.01861

A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman *et al.*, "The llama 3 herd of models," *arXiv preprint arXiv:2407.21783*, 2024.

D. Eck and J. Schmidhuber, "A first look at music composition using LSTM recurrent neural networks," Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale, Tech. Rep., 2002.

A. Eghbali and M. Pradel, "CrystalBLEU: Precisely and Efficiently Measuring the Similarity of Code," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering.* Rochester MI USA: ACM, Oct. 2022, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/3551349.3556903

A. T. Eitan, E. Smolyansky, I. K. Harpaz, and S. Perets, "Connected Papers," 2024. [Online]. Available: https://www.connectedpapers.com/

Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong *et al.*, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," Sep. 2020, arXiv:2002.08155 [cs]. [Online]. Available: http://arxiv.org/abs/2002.08155

G. Fraser and A. Arcuri, "EvoSuite: On the Challenges of Test Case Generation in the Real World," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation.* Luxembourg, Luxembourg: IEEE, Mar. 2013, pp. 362–369. [Online]. Available: http://ieeexplore.ieee.org/document/6569748/

G. Fraser and A. Arcuri, "A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite," *ACM Transactions on Software Engineering and Methodology*, vol. 24, no. 2, pp. 1–42, Dec. 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2685612

D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi *et al.*, "InCoder: A Generative Model for Code Infilling and Synthesis," Apr. 2023, arXiv:2204.05999 [cs]. [Online]. Available: http://arxiv.org/abs/2204.05999

L. Gong, S. Wang, M. Elhoushi, and A. Cheung, "Evaluation of LLMs on Syntax-Aware Code Fill-in-the-Middle Tasks," Jun. 2024, arXiv:2403.04814 [cs]. [Online]. Available: http://arxiv.org/abs/2403.04814

A. Graves, "Generating Sequences With Recurrent Neural Networks," Jun. 2014, arXiv:1308.0850 [cs]. [Online]. Available: http://arxiv.org/abs/1308.0850

D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang *et al.*, "DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence," Jan. 2024, arXiv:2401.14196 [cs]. [Online]. Available: http://arxiv.org/abs/2401.14196

L. B. Heitz, J. Chamas, and C. Scherb, "Evaluation of the Programming Skills of Large Language Models," May 2024, arXiv:2405.14388 [cs]. [Online]. Available: http://arxiv.org/abs/2405.14388

E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of NL-queries for software maintenance and reuse," in *2009 IEEE 31st International Conference on Software Engineering.* Vancouver, BC, Canada: IEEE, 2009, pp. 232–242. [Online]. Available: http://ieeexplore.ieee.org/document/5070524/

A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of the 34th international conference on software engineering,* ser. Icse '12. IEEE Press, 2012, pp. 837–847, place: Zurich, Switzerland Number of pages: 11.

S. Hochreiter, "Long short-term memory," *Neural Computation MIT-Press*, 1997.

S. Hochreiter, Y. Bengio, P. Frasconi, J. Schmidhuber, and others, "Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies," in *A Field Guide to Dynamical Recurrent Networks.* IEEE, 2001.

X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension.* Gothenburg Sweden: ACM, May 2018, pp. 200–210. [Online]. Available: https://dl.acm.org/doi/10.1145/3196321.3196334

X. Hu, G. Li, X. Xia, D. Lo, S. Lu, and Z. Jin, "Summarizing Source Code with Transferred API Knowledge," in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence.* Stockholm, Sweden: International Joint Conferences on Artificial Intelligence Organization, Jul. 2018, pp. 2269–2275. [Online]. Available: https://www.ijcai.org/proceedings/2018/314

H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search," Jun. 2020, arXiv:1909.09436 [cs]. [Online]. Available: http://arxiv.org/abs/1909.09436

S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing Source Code using a Neural Attention Model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* Berlin, Germany: Association for Computational Linguistics, 2016, pp. 2073–2083. [Online]. Available: http://aclweb.org/anthology/P16-1195

S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Mapping Language to Code in Programmatic Context," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing.* Brussels, Belgium: Association for Computational Linguistics, 2018, pp. 1643–1652. [Online]. Available: http://aclweb.org/anthology/D18-1192

S. Jean, K. Cho, R. Memisevic, and Y. Bengio, "On Using Very Large Target Vocabulary for Neural Machine Translation," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers).* Beijing, China: Association for Computational Linguistics, 2015, pp. 1–10. [Online]. Available: http://aclweb.org/anthology/P15-1001

A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas *et al.*, “Mistral 7B,” Oct. 2023, arXiv:2310.06825 [cs]. [Online]. Available: http://arxiv.org/abs/2310.06825

A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford *et al.*, “Mixtral of Experts,” Jan. 2024, arXiv:2401.04088 [cs]. [Online]. Available: http://arxiv.org/abs/2401.04088

J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, “A Survey on Large Language Models for Code Generation,” Nov. 2024, arXiv:2406.00515 [cs]. [Online]. Available: http://arxiv.org/abs/2406.00515

R. Just, D. Jalali, and M. D. Ernst, “Defects4J: a database of existing faults to enable controlled testing studies for Java programs,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. San Jose CA USA: ACM, Jul. 2014, pp. 437–440. [Online]. Available: https://dl.acm.org/doi/10.1145/2610384.2628055

D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, May 2013, pp. 802–811. [Online]. Available: http://ieeexplore.ieee.org/document/6606626/

D. Kocetkov, R. Li, L. B. Allal, J. Li, C. Mou, C. M. Ferrandis *et al.*, “The Stack: 3 TB of permissively licensed source code,” Nov. 2022, arXiv:2211.15533 [cs]. [Online]. Available: http://arxiv.org/abs/2211.15533

S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken *et al.*, “SPoC: Search-based pseudocode to code,” in *Advances in neural information processing systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché Buc, E. Fox, and R. Garnett, Eds., vol. 32. Curran Associates, Inc., 2019. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2019/file/7298332f04ac004a0ca44cc69ecf6f6b-Paper.pdf

C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “GenProg: A Generic Method for Automatic Software Repair,” *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, Jan. 2012. [Online]. Available: http://ieeexplore.ieee.org/document/6035728/

A. LeClair, S. Jiang, and C. McMillan, “A Neural Model for Generating Natural Language Summaries of Program Subroutines,” in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. Montreal, QC, Canada: IEEE, May 2019, pp. 795–806. [Online]. Available: https://ieeexplore.ieee.org/document/8811932/

R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou *et al.*, "StarCoder: may the source be with you!" Dec. 2023, arXiv:2305.06161 [cs]. [Online]. Available: http://arxiv.org/abs/2305.06161

J. T. Liang, C. Yang, and B. A. Myers, "A Large-Scale Survey on the Usability of AI Programming Assistants: Successes and Challenges," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Lisbon Portugal: ACM, Feb. 2024, pp. 1–13. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3608128

Y. Liang and K. Q. Zhu, "Automatic generation of text descriptive comments for code blocks," in *Proceedings of the thirty-second AAAI conference on artificial intelligence and thirtieth innovative applications of artificial intelligence conference and eighth AAAI symposium on educational advances in artificial intelligence*, ser. AAAI'18/IAAI'18/EAAI'18. AAAI Press, 2018, place: New Orleans, Louisiana, USA Number of pages: 8 tex.articleno: 641.

C.-Y. Lin, "ROUGE: a package for automatic evaluation of summaries," in *Text summarization branches out*. Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013/

D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. Vancouver BC Canada: ACM, Oct. 2017, pp. 55–56. [Online]. Available: https://dl.acm.org/doi/10.1145/3135932.3135941

W. Ling, E. Grefenstette, K. M. Hermann, T. Kočiský, A. Senior, F. Wang *et al.*, "Latent Predictor Networks for Code Generation," Jun. 2016, arXiv:1603.06744 [cs]. [Online]. Available: http://arxiv.org/abs/1603.06744

K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Beijing China: ACM, Jul. 2019, pp. 31–42. [Online]. Available: https://dl.acm.org/doi/10.1145/3293882.3330577

A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi *et al.*, "StarCoder 2 and The Stack v2: The Next Generation," Feb. 2024, arXiv:2402.19173 [cs]. [Online]. Available: http://arxiv.org/abs/2402.19173

S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco *et al.*, "CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation," Mar. 2021, arXiv:2102.04664 [cs]. [Online]. Available: http://arxiv.org/abs/2102.04664

T. Luong, H. Pham, and C. D. Manning, "Effective Approaches to Attention-based Neural Machine Translation," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing.* Lisbon, Portugal: Association for Computational Linguistics, 2015, pp. 1412–1421. [Online]. Available: http://aclweb.org/anthology/D15-1166

J. López Espejel, M. S. Yahaya Alassan, E. M. Chouham, W. Dahhane, and E. H. Ettifouri, "A comprehensive review of State-of-The-Art methods for Java code generation from Natural Language Text," *Natural Language Processing Journal*, vol. 3, p. 100013, Jun. 2023. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2949719123000109

F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER).* Hangzhou, China: IEEE, Feb. 2019, pp. 468–478. [Online]. Available: https://ieeexplore.ieee.org/document/8667991/

M. Martinez and M. Monperrus, "ASTOR: a program repair library for Java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis.* Saarbrücken Germany: ACM, Jul. 2016, pp. 441–444. [Online]. Available: https://dl.acm.org/doi/10.1145/2931037.2948705

M. Monteiro, B. C. Branco, S. Silvestre, G. Avelino, and M. T. Valente, "End-to-end software construction using ChatGPT: An experience report," 2023, arXiv: 2310.14843 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2310.14843

H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, "When to Show a Suggestion? Integrating Human Feedback in AI-Assisted Programming," Apr. 2024, arXiv:2306.04930 [cs]. [Online]. Available: http://arxiv.org/abs/2306.04930

N. Muennighoff, Q. Liu, A. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo *et al.*, "OctoPack: Instruction Tuning Code Large Language Models," Feb. 2024, arXiv:2308.07124 [cs]. [Online]. Available: http://arxiv.org/abs/2308.07124

S. O'Neil, "AWS CodeWhisperer creates computer code from natural language," Dec. 2021. [Online]. Available: https://www.amazon.science/latest-news/aws-codewhisperer-creates-computer-code-from-natural-language

OpenAI, "GPT-4o System Card," Dec. 2024. [Online]. Available: https://cdn.openai.com/gpt-4o-system-card.pdf

OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya *et al.*, "GPT-4 Technical Report," Mar. 2024, arXiv:2303.08774 [cs]. [Online]. Available: http://arxiv.org/abs/2303.08774

Oracle Corporation, "Javadoc Tool." [Online]. Available: https://www.oracle.com/java/technologies/javase/javadoc-tool.html

C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for Java," in *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion.* Montreal Quebec Canada: ACM, Oct. 2007, pp. 815–816. [Online]. Available: https://dl.acm.org/doi/10.1145/1297846.1297902

K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: a method for automatic evaluation of machine translation," in *Proceedings of the 40th Annual Meeting on Association for Computational Linguistics - ACL '02.* Philadelphia, Pennsylvania: Association for Computational Linguistics, 2001, p. 311. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1073083.1073135

S. Peng, E. Kalliamvakou, P. Cihon, and M. Demirer, "The Impact of AI on Developer Productivity: Evidence from GitHub Copilot," Feb. 2023, arXiv:2302.06590 [cs]. [Online]. Available: http://arxiv.org/abs/2302.06590

M. Popović, "chrF: character n-gram F-score for automatic MT evaluation," in *Proceedings of the Tenth Workshop on Statistical Machine Translation.* Lisbon, Portugal: Association for Computational Linguistics, 2015, pp. 392–395. [Online]. Available: http://aclweb.org/anthology/W15-3049

M. Popović, "chrF++: words helping character n-grams," in *Proceedings of the Second Conference on Machine Translation.* Copenhagen, Denmark: Association for Computational Linguistics, 2017, pp. 612–618. [Online]. Available: http://aclweb.org/anthology/W17-4770

Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and*

Analysis. Baltimore MD USA: ACM, Jul. 2015, pp. 24–36. [Online]. Available: https://dl.acm.org/doi/10.1145/2771783.2771791

A. Radford and Karthik, "Improving language understanding by generative pre-training," 2018. [Online]. Available: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

S. Rasnayaka, G. Wang, R. Shariffdeen, and G. N. Iyer, "An Empirical Study on Usage and Perceptions of LLMs in a Software Engineering Project," Jan. 2024, arXiv:2401.16186 [cs]. [Online]. Available: http://arxiv.org/abs/2401.16186

S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang *et al.*, "CodeBLEU: a Method for Automatic Evaluation of Code Synthesis," Sep. 2020, arXiv:2009.10297 [cs]. [Online]. Available: http://arxiv.org/abs/2009.10297

Replit, Inc, "Replit's new AI model now available on hugging face," 2023. [Online]. Available: https://blog.replit.com/replit-code-v1_5

T. Ridnik, D. Kredo, and I. Friedman, "Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering," Jan. 2024, arXiv:2401.08500 [cs]. [Online]. Available: http://arxiv.org/abs/2401.08500

S. Robertson and H. Zaragoza, "The Probabilistic Relevance Framework: BM25 and Beyond," *Foundations and Trends® in Information Retrieval*, vol. 3, no. 4, pp. 333–389, 2009. [Online]. Available: http://www.nowpublishers.com/article/Details/INR-019

S. I. Ross, F. Martinez, S. Houde, M. Muller, and J. D. Weisz, "The Programmer's Assistant: Conversational Interaction with a Large Language Model for Software Development," in *Proceedings of the 28th International Conference on Intelligent User Interfaces*. Sydney NSW Australia: ACM, Mar. 2023, pp. 491–514. [Online]. Available: https://dl.acm.org/doi/10.1145/3581641.3584037

B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan *et al.*, "Code Llama: Open Foundation Models for Code," Jan. 2024, arXiv:2308.12950 [cs]. [Online]. Available: http://arxiv.org/abs/2308.12950

N. Saavedra, A. Silva, and M. Monperrus, "GitBug-Actions: Building Reproducible Bug-Fix Benchmarks with GitHub Actions," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. Lisbon Portugal: ACM, Apr. 2024, pp. 1–5. [Online]. Available: https://dl.acm.org/doi/10.1145/3639478.3640023

A. Salinas and F. Morstatter, “The Butterfly Effect of Altering Prompts: How Small Changes and Jailbreaks Affect Large Language Model Performance,” in *Findings of the Association for Computational Linguistics ACL 2024*. Bangkok, Thailand and virtual meeting: Association for Computational Linguistics, 2024, pp. 4629–4651. [Online]. Available: https://aclanthology.org/2024.findings-acl.275

R. M. Schmidt, “Recurrent Neural Networks (RNNs): A gentle Introduction and Overview,” Nov. 2019, arXiv:1912.05911 [cs]. [Online]. Available: http://arxiv.org/abs/1912.05911

A. Semenkin, V. Bibaev, Y. Sokolov, K. Krylov, A. Kalina, A. Khannanova *et al.*, “Full line code completion: Bringing AI to desktop,” *arXiv preprint arXiv:2405.08704*, 2024.

M. L. Siddiq, J. C. S. Santos, R. H. Tanvir, N. Ulfat, F. A. Rifat, and V. C. Lopes, “Using Large Language Models to Generate JUnit Tests: An Empirical Study,” in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, Jun. 2024, pp. 313–322, arXiv:2305.00418 [cs]. [Online]. Available: http://arxiv.org/abs/2305.00418

A. Silva, N. Saavedra, and M. Monperrus, “GitBug-Java: A Reproducible Benchmark of Recent Java Bugs,” in *Proceedings of the 21st International Conference on Mining Software Repositories*, Apr. 2024, pp. 118–122, arXiv:2402.02961 [cs]. [Online]. Available: http://arxiv.org/abs/2402.02961

G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, “Towards automatically generating summary comments for Java methods,” in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. Antwerp Belgium: ACM, Sep. 2010, pp. 43–52. [Online]. Available: https://dl.acm.org/doi/10.1145/1858996.1859006

I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” in *Proceedings of the 28th international conference on neural information processing systems - volume 2*, ser. NIPS’14. Cambridge, MA, USA: MIT Press, 2014, pp. 3104–3112, number of pages: 9 Place: Montreal, Canada.

A. Svyatkovskiy, S. K. Deng, S. Fu, and N. Sundaresan, “IntelliCode compose: code generation using transformer,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Virtual Event USA: ACM, Nov. 2020, pp. 1433–1443. [Online]. Available: https://dl.acm.org/doi/10.1145/3368089.3417058

B. Szalontai, G. Szalay, T. Márton, A. Sike, B. Pintér, and T. Gregorics, "Large Language Models for Code Summarization," May 2024, arXiv:2405.19032 [cs]. [Online]. Available: http://arxiv.org/abs/2405.19032

N. Tang, J. An, M. Chen, A. Bansal, Y. Huang, C. McMillan *et al.*, "CodeGRITS: A Research Toolkit for Developer Behavior and Eye Tracking in IDE," in *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings.* Lisbon Portugal: ACM, Apr. 2024, pp. 119–123. [Online]. Available: https://dl.acm.org/doi/10.1145/3639478.3640037

N. Tang, M. Chen, Z. Ning, A. Bansal, Y. Huang, C. McMillan *et al.*, "A Study on Developer Behaviors for Validating and Repairing LLM-Generated Code Using Eye Tracking and IDE Actions," May 2024, arXiv:2405.16081 [cs]. [Online]. Available: http://arxiv.org/abs/2405.16081

C. Team, H. Zhao, J. Hui, J. Howland, N. Nguyen, S. Zuo *et al.*, "CodeGemma: Open Code Models Based on Gemma," Jun. 2024, arXiv:2406.11409 [cs]. [Online]. Available: http://arxiv.org/abs/2406.11409

G. Team, R. Anil, S. Borgeaud, J.-B. Alayrac, J. Yu, R. Soricut *et al.*, "Gemini: A Family of Highly Capable Multimodal Models," Jun. 2024, arXiv:2312.11805 [cs]. [Online]. Available: http://arxiv.org/abs/2312.11805

G. Team, P. Georgiev, V. I. Lei, R. Burnell, L. Bai, A. Gulati *et al.*, "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context," Dec. 2024, arXiv:2403.05530 [cs]. [Online]. Available: http://arxiv.org/abs/2403.05530

G. Team, T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak *et al.*, "Gemma: Open Models Based on Gemini Research and Technology," Apr. 2024, arXiv:2403.08295 [cs]. [Online]. Available: http://arxiv.org/abs/2403.08295

H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix *et al.*, "LLaMA: Open and Efficient Foundation Language Models," Feb. 2023, arXiv:2302.13971 [cs]. [Online]. Available: http://arxiv.org/abs/2302.13971

H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei *et al.*, "Llama 2: Open Foundation and Fine-Tuned Chat Models," Jul. 2023, arXiv:2307.09288 [cs]. [Online]. Available: http://arxiv.org/abs/2307.09288

M. Tufano, S. K. Deng, N. Sundaresan, and A. Svyatkovskiy, "Methods2Test: a dataset of focal methods mapped to test cases," in *Proceedings of the 19th International Conference on Mining Software Repositories.* Pittsburgh

Pennsylvania: ACM, May 2022, pp. 299–303. [Online]. Available: https://dl.acm.org/doi/10.1145/3524842.3528009

P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models," in *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. New Orleans LA USA: ACM, Apr. 2022, pp. 1–7. [Online]. Available: https://dl.acm.org/doi/10.1145/3491101.3519665

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez *et al.*, "Attention is all you need," in *Advances in neural information processing systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan *et al.*, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

C. Watson, M. Tufano, K. Moran, G. Bavota, and D. Poshyvanyk, "On learning meaningful assert statements for unit test cases," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. Seoul South Korea: ACM, Jun. 2020, pp. 1398–1409. [Online]. Available: https://dl.acm.org/doi/10.1145/3377811.3380429

C. S. Xia, Y. Wei, and L. Zhang, "Automated Program Repair in the Era of Large Pre-trained Language Models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. Melbourne, Australia: IEEE, May 2023, pp. 1482–1494. [Online]. Available: https://ieeexplore.ieee.org/document/10172803/

A. Yang, B. Yang, B. Hui, B. Zheng, B. Yu, C. Zhou *et al.*, "Qwen2 Technical Report," Sep. 2024, arXiv:2407.10671 [cs]. [Online]. Available: http://arxiv.org/abs/2407.10671

H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma *et al.*, "CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Lisbon Portugal: ACM, Feb. 2024, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/3597503.3623316

Z. Yuan, Y. Lou, M. Liu, S. Ding, K. Wang, Y. Chen *et al.*, "No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation," May 2024, arXiv:2305.04207 [cs]. [Online]. Available: http://arxiv.org/abs/2305.04207

Z. Zeng, Y. Wang, R. Xie, W. Ye, and S. Zhang, “CoderUJB: An Executable and Unified Java Benchmark for Practical Programming Scenarios,” Mar. 2024, arXiv:2403.19287 [cs]. [Online]. Available: http://arxiv.org/abs/2403.19287

F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan *et al.*, “RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation,” in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. Singapore: Association for Computational Linguistics, 2023, pp. 2471–2484. [Online]. Available: https://aclanthology.org/2023.emnlp-main.151

Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue *et al.*, “CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X,” in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. Long Beach CA USA: ACM, Aug. 2023, pp. 5673–5684. [Online]. Available: https://dl.acm.org/doi/10.1145/3580305.3599790

S. Zhou, U. Alon, S. Agarwal, and G. Neubig, “CodeBERTScore: Evaluating Code Generation with Pretrained Models of Code,” Oct. 2023, arXiv:2302.05527 [cs]. [Online]. Available: http://arxiv.org/abs/2302.05527

A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister *et al.*, “Productivity assessment of neural code completion,” in *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*. San Diego CA USA: ACM, Jun. 2022, pp. 21–29. [Online]. Available: https://dl.acm.org/doi/10.1145/3520312.3534864

Appendix

A. Practical Scenario

This appendix provides the data model, all user stories, and all UI mockups that are part of the expense tracking application in the practical scenario. All three were defined before the exploration phase of the scenario.

A.1. Data Model

In this section, we present the underlying data model that defines the different entities in the expense tracking application of the practical scenario. The visual representation of the data model can be seen in Figure A.1.

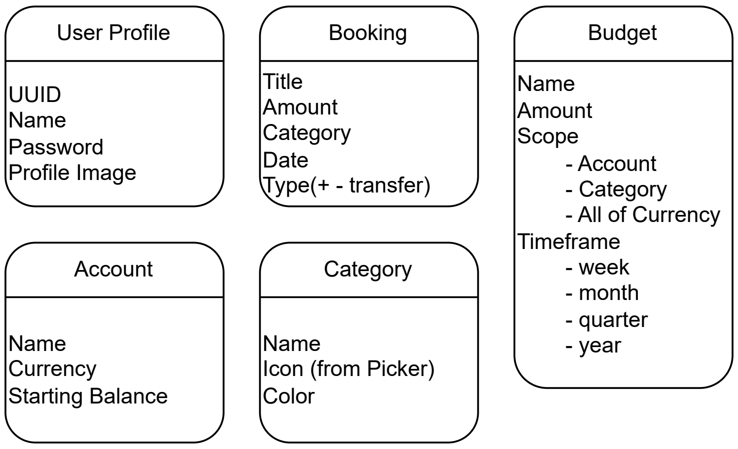


Figure A.1.: **Data Model** for the expense tracking application in the practical scenario.

User Profile An instance of a user profile represents an individual application user and consists of a unique user id, a username, a password, and a profile image. Moreover, the profile image excludes SVGs to avoid having to deal with security issues related to XML parsing.

Account An account instance represents an individual account to which bookings can be made. It consists of a name, a currency, and a starting balance.

Booking A booking represents a real-world transaction like withdrawing money at an ATM. It consists of a title, an amount (in the currency of the account), a category, a booking date, and a booking type. The booking types are *deposit*, *withdrawal*, and *transfer* (between accounts).

Category Categories allow the user to group bookings by, for example, an expense type. It is characterized by a name, an icon, and a color.

Budget A budget allows the user to set spending limits that trigger if exceeded. It is defined by a name, an amount, a timeframe, and a scope. The timeframe defines whether the budget is for a week, a month, a quarter, or a year. The scope can be set to a single account, category, or all accounts with a particular currency.

A.2. User Stories

The simplified user stories in this section are grouped into several tables based on the entities in the data model. The following tables (A.1, A.2, A.3, A.4, A.5) contain the user stories per entity type and are self-contained.

Additionally, two user stories are more open-ended and do not fit particularly well to one of the entities:

- As a user, I want to gain insights into my spending habits, especially regarding month-to-month differences and outliers.
- As a user, I want to be able to automatically scan and add receipt totals as an entry.

A.3. UI Mockups

This section features all UI mockups that were present at the start of the exploration phase. The relevant figures are (A.2, A.3, A.4, A.5, A.4, A.8) and are self-contained.

User Profile	
As a user, I want to be able to...	able to create a profile
	delete my profile after entering the correct password
	log into my profile with my credentials
	log out of my profile
	change my username to a new, unique username after entering the correct password
	change my password after entering the correct current password

Table A.1.: **User Profile** user stories detailing the requirements related managing individual user profiles.

Accounts	
As a user, I want to be able to...	create a default account on my first log in
	always have a default account
	create additional accounts
	change default accounts
	delete all of my accounts except for the default one
	edit all my accounts
	see the current balance of my accounts
	see the account delta for the current month

Table A.2.: **Accounts** user stories detailing requirements regarding accounts.

Bookings	
As a user, I want to be able to...	create bookings for my accounts
	create bookings between all of my accounts
	edit all of my bookings
	delete my bookings
	view the details of my bookings
	view my bookings in chronological order
	filter my bookings by account, budget, time, and category
	sort my bookings by amount, A-Z, and inverse

Table A.3.: **Bookings** user stories. The two user stories marked in blue must be implemented on the server-side.

Categories	
As a user, I want to be able to...	always have a default category
	view all of my categories
	edit my categories
	delete my categories and set all bookings to the default category
	set multiple categories for a booking

Table A.4.: **Categories** user stories.

Budgets	
As a user, I want to be able to...	create budgets
	edit budgets
	delete budgets
	receive a warning if a booking exceeds a budget
	see all of my budgets and their current balance

Table A.5.: **Budgets** user stories.

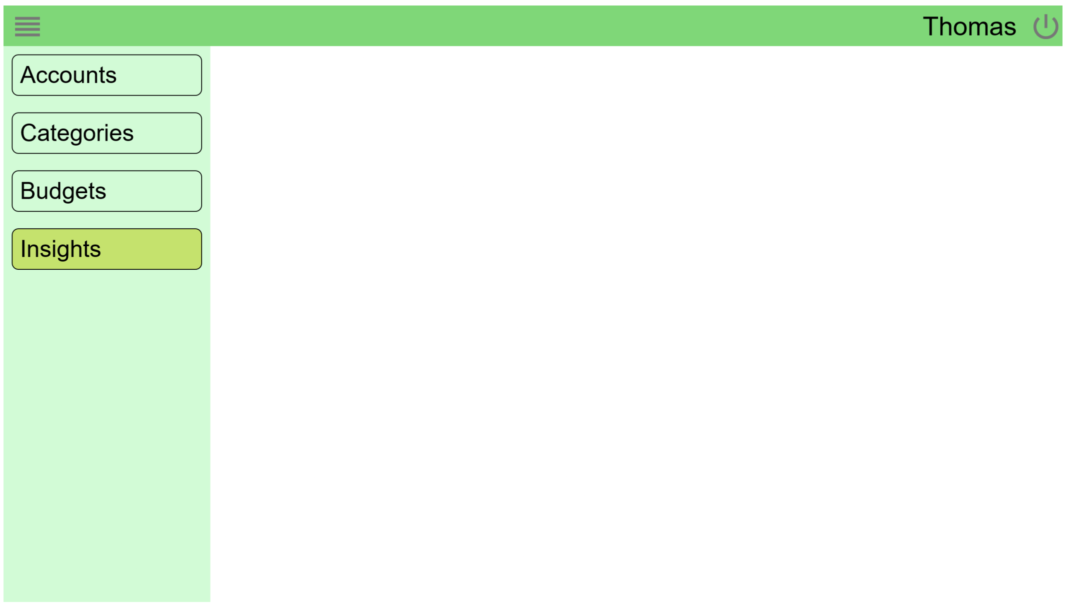


Figure A.2.: **UI Mockup** of the **Basic Application Components**, featuring a collapsible sidebar menu and an indication of the current user, including the possibility to log out.

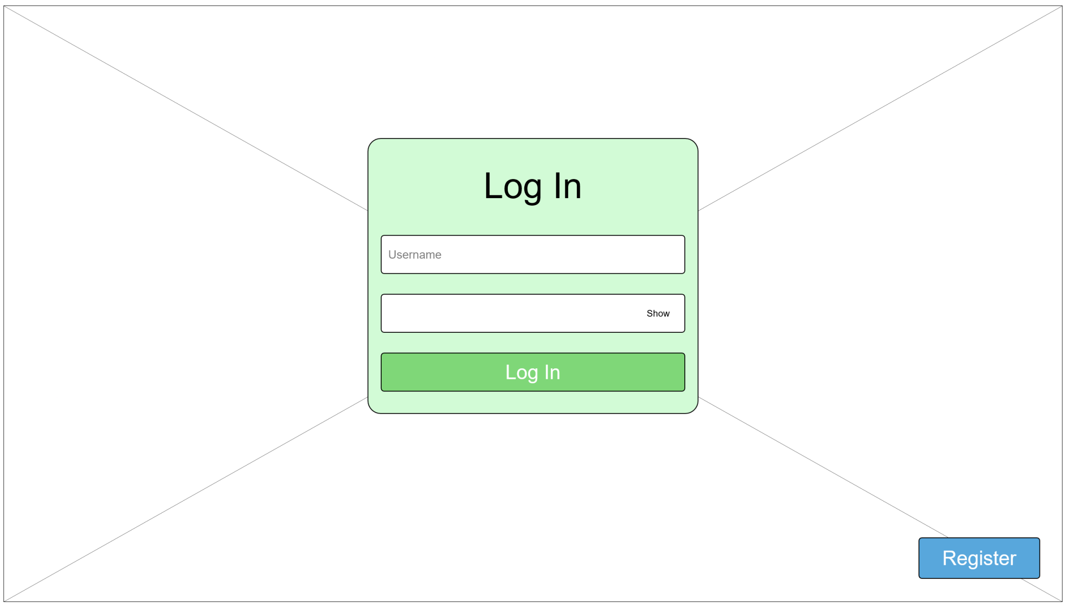


Figure A.3.: **UI Mockup** of **Login Form**, including the possibility of registring a new account.

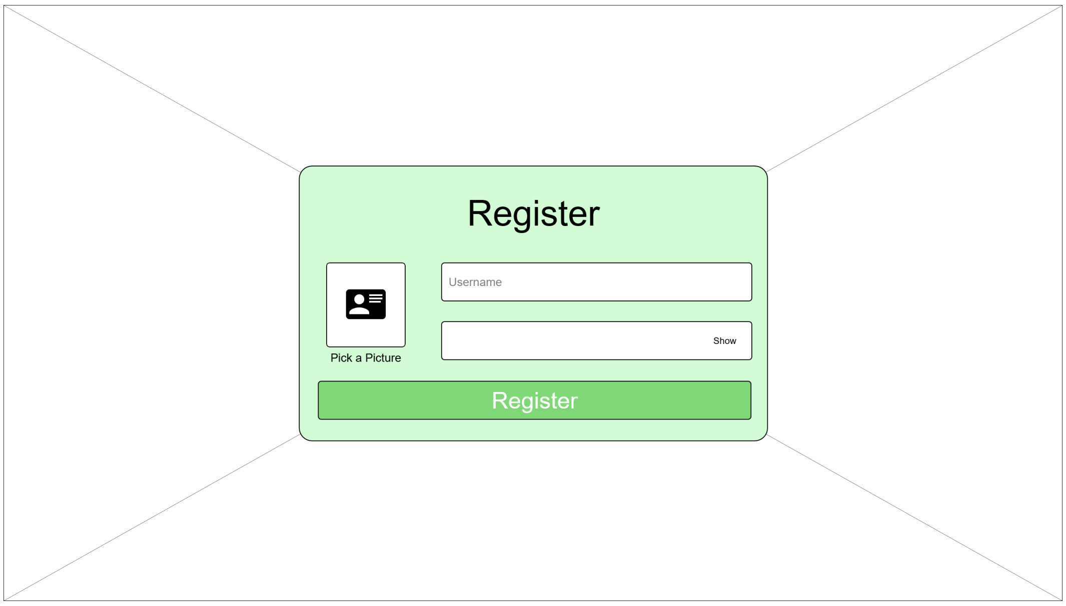


Figure A.4.: **UI Mockup** of the **Account Creation Form**, including the possibility to pick a profile image from the local machine.

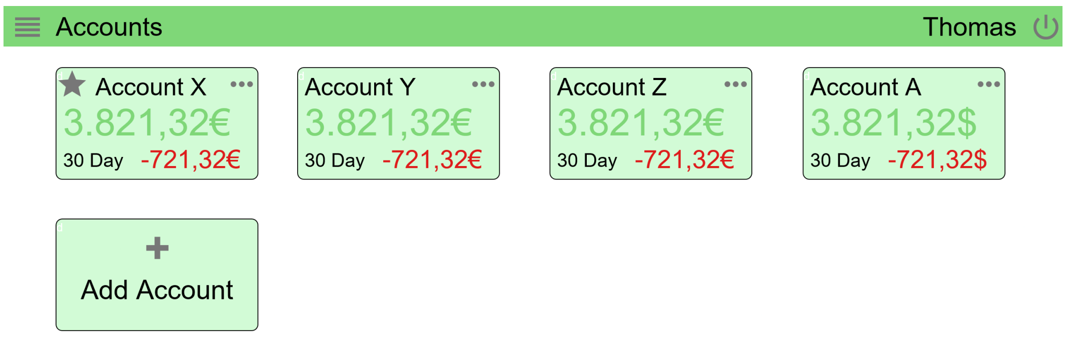


Figure A.5.: **UI Mockup** of **Accounts Overview Page**. It contains multiple accounts, one denoted as the default account using a star.

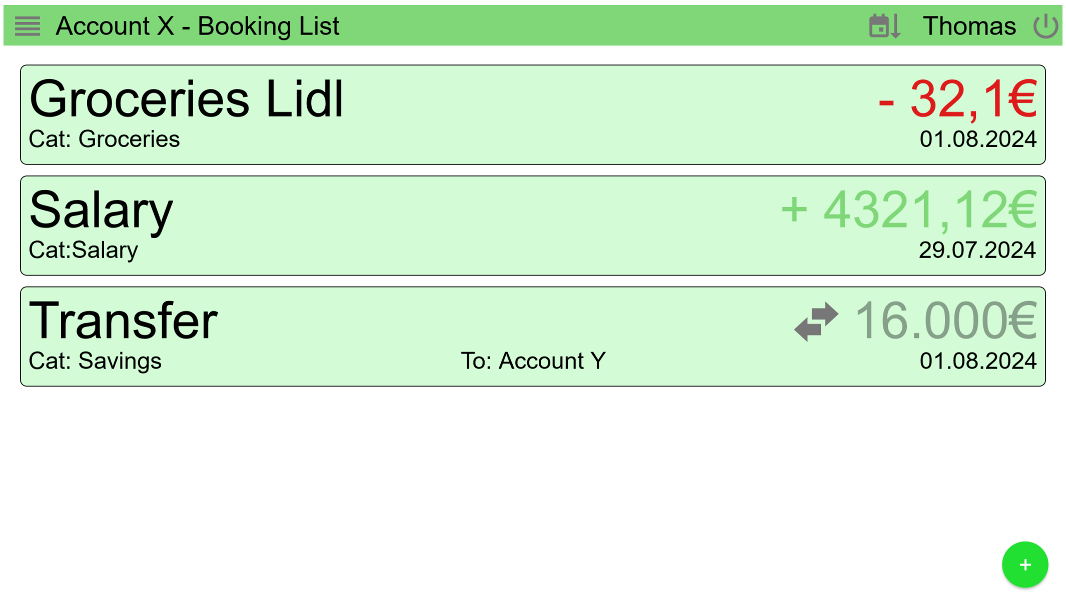


Figure A.6.: **UI Mockup** of a **Booking List** containing all bookings for an account, including the possibility to add an entry using a floating action button. The entries can be sorted using the icon in the header.

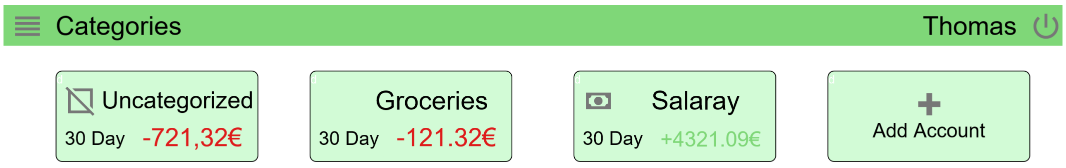


Figure A.7.: **UI Mockup** featuring the **Categories Page**. It includes the default category, two custom categories, and the possibility of creating a new one.

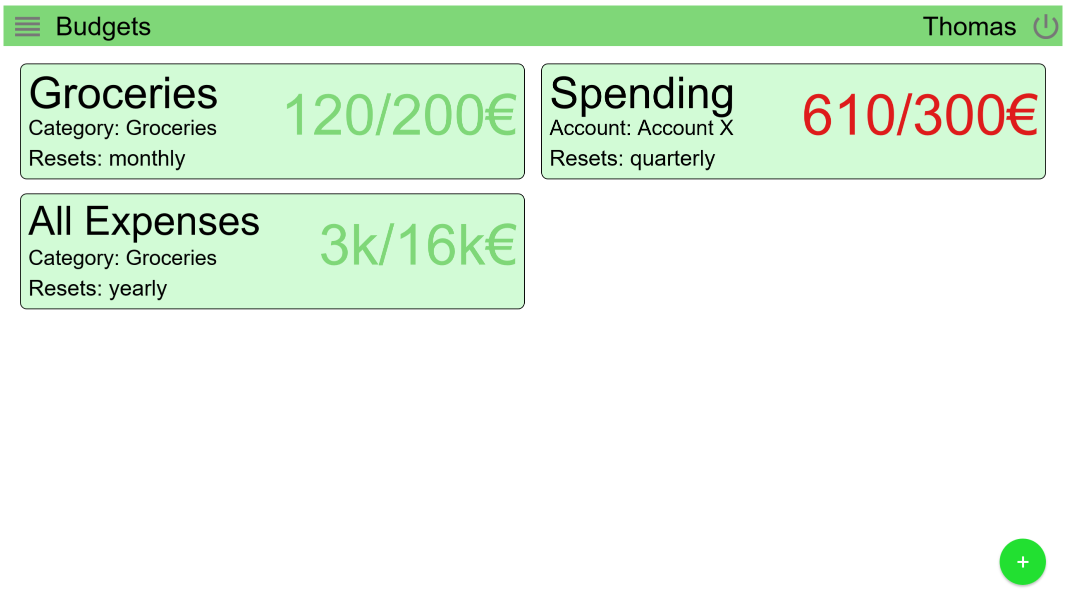


Figure A.8.: **UI Mockup** showing three different **Budgets** for different scopes and timeframes, including an indication of one account being over budget.

B. Code Completion Prompt Examples

Since Figure 6.4 in Section 6.2 might not be ideally suited to illustrate how the prompts would look like in practice, we give examples for both SPM and 1S prompting here. These were taken from the SAFIM benchmark (Gong et al., 2024).

Additional line breaks were automatically inserted, and text was omitted to increase readability.

Listing B.1: Suffix-Prefix-Middle (SPM) prompt example

```
            }
        }
        return bCount > 0 && s.charAt(s.length()-1) == 'B';
    }

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        try {
            int t = s.nextInt();
            for(int i=0;i<t;i++) {
                String inp = s.next();
                System.out.println(isValidString(inp)?"YES
                    ":"NO");
            }
        } catch(Exception e) {
            System.out.println("Exception: " + e);
            e.printStackTrace();
        } finally {
            s.close();
        }
    }
}
```

```
Complete the code in java to solve this programming problem:

Description: <text omitted for brevity>

Input Specification: <text omitted for brevity>

Output Specification: <text omitted for brevity>

Notes: <text omitted for brevity>

Code:

import java.util.*;
public class ILoveAAB {

    public static boolean isValidString(String s) {
        if(s.length() <= 1) return false;
        int bCount = 0, aCount = 0;
        for(int i=0;i<s.length();i++) {
            if(s.charAt(i) == 'A') aCount += 1;
            else bCount += 1;
            if(bCount > aCount) {
```

Listing B.2: One Shot (1S) prompt example

```
Complete the code in java to solve this programming problem:

Description: <text omitted for brevity>

Input Specification: <text omitted for brevity>

Output Specification: <text omitted for brevity>

Code:

import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int t = sc.nextInt();
        for (int i = 0; i < t; i++) {
            /* TODO: Your code here */
        }
        sc.close();
    }
}

Replace the "/* TODO: Your code here */" in the code above
   with the appropriate block. Provide only the replaced
   block.

int a = sc.nextInt();
            int b = sc.nextInt();
            int c = a + b;
            System.out.println(c);


Complete the code in java to solve this programming problem:

Description: <text omitted for brevity>

Input Specification: <text omitted for brevity>

Output Specification: <text omitted for brevity>
```

```
Notes: <text omitted for brevity>

Code:

import java.util.*;
public class ILoveAAB {

    public static boolean isValidString(String s) {
        if(s.length() <= 1) return false;
        int bCount = 0, aCount = 0;
        for(int i=0;i<s.length();i++) {
            if(s.charAt(i) == 'A') aCount += 1;
            else bCount += 1;
            if(bCount > aCount) {
                /* TODO: Your code here */
            }
        }
        return bCount > 0 && s.charAt(s.length()-1) == 'B';
    }

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        try {
            int t = s.nextInt();
            for(int i=0;i<t;i++) {
                String inp = s.next();
                System.out.println(isValidString(inp)?"YES
                   ":"NO");
            }
        } catch(Exception e) {
            System.out.println("Exception: " + e);
            e.printStackTrace();
        } finally {
            s.close();
        }
    }
}


Replace the "/* TODO: Your code here */" in the code above
   with the appropriate block. Provide only the replaced
   block.
```