SA - LLM Assisted Development

Dominik Castelberg
dominik.castelberg@ost.ch

Linus Flury
linus.flury@ost.ch

Supervisor: Prof. Mitra Purandere
mitra.purandere@ost.ch

I. Abstract

A. Introduction

In recent years, Large Language Models (LLMs) have become valuable tools for developers with their ability to quickly scaffold code and act as an impactful accelerator for development teams around the world. With code being heavily standardized on a global scale and an abundance of training data freely available on platforms such as GitHub, it is easy to intuit possible reasons for their performance. While these assumptions hold true for popular languages such as Java, Python or C#, there has been little research in the usage of LLMs as development tools for less commonly used languages such as Haskell. With this project we aim to explore LLM based development support for Haskell and develop an environment, in which such research can be conducted in a quick and efficient manner.

B. Approach

Four state-of-the-art models (Llama 2, Code Llama, GPT-3.5 and GPT-4) were evaluated based on their performance on tasks typically faced by an automated development support tool. The tasks were scoped and classified into three major categories: Code Generation, Debugging and Testing. For each of these categories, quantifiable metrics for the evaluation of the model performance were defined. These criteria must be interpretable as quantifiable metrics to allow a comparative analysis between models. These metrics were then weighted according to their importance, based on the insight of experts in the field of Haskell development. Each task was executed with three sorting algorithms of varying cognitive complexity in sample implementations. Cognitive complexity was selected as the complexity measure after careful evaluation, to ensure that the ordering of the algorithms is based on complexity of interpretability. Utilizing cognitive complexity inspired us to frame LLMs as entities whose performance can be analysed through the lens of cognitive load theory. This enabled the differentiation between errors caused by the complexity of a provided algorithm (intrinsic load) and errors caused by unclear instructions (extraneous load). To accelerate the evaluation processes, a development environment, enabling both the automated testing of generated Haskell code using Jupyter notebooks and the utilization of cloud hosted models with modern LLM tooling like LangChain, was created.

C. Conclusion

Our work has shown that there are large gaps in output quality between each model. We have encountered outliers, but we are confident that these outliers were not caused by the intrinsic complexity of the algorithms provided and can be explained with extraneous complexity that lead to the model not understanding the task. This problem can be resolved with further prompt engineering and fine-tuning, which is why we are optimistic about the viability of models such as GPT-4 or Code Llama as supporting tools for Haskell development. Using Chain of Thought Prompting, which leads models to break down given tasks into sequential subtasks, tends to increase the outputquality in general. However the sequential nature of it lead to inconsistencies in the compositions of Haskell's higher-order functions. It lead the models to neglect critical nuances in function composition, resulting in erroneous code generation.

II. Executive Summary

Introduction In recent years, Large Language Models (LLMs) stopped being an academic curiosity and have become value generators across multiple industries and markets. Their ability to very quickly scaffold texts ranging from offerings, documentation up to executable software code have increased productivity for companies across the globe. They have established an entire industry, with companies like OpenAI, Meta and Microsoft leading the charge into a future in which generative AI will be omnipresent. We believe that taking advantage of the booming market for LLMs and investing into research about optimal LLM usage puts any investing company in an advantageous position when generative AI becomes more mature and excellence in ML and prompt engineering become demanded skill sets across all industries.

Project Goals Our goal is to verify the viability of LLMs as development support tools for Haskell developers. As their abilities when working with more commonly used languages has already been established, we focused on a technology stack that fulfils a more specific niche. It being a language used for functional programming, Haskell poses unique challenges to a neural network that has primarily been trained with programming languages used for object-oriented programming. We also aim to develop processes and tooling that streamline research tasks including LLMs.

Methodology In order to minimize risks during our evaluation phase, we conducted extensive research at the start of our project to identify and remove any major roadblocks during early development. In this phase we developed a workflow template that ensures that each step is built upon rigorous examination of multiple layers of requirements. We then conducted initial research into prompt engineering, algorithm analysis and cognitive load theory to establish a foundation on which we can base our evaluation criteria used to assess the viability of each model we are testing. We consulted established academic and industrial experts in the fields of Haskell development and data science to receive guidance in their respective specialties. We decided on a set of LLMs to analyse and tested their capabilities with numerous use cases, sample algorithms of varying complexity and evaluated their performance with our developed evaluation criteria.

Results Our experimentation has shown that modern LLMs are easily capable of producing and analyzing working Haskell code for algorithms of low to intermediate complexity. The analysis of higher complexity algorithms has shown itself to be a challenge that can be surmounted with skilled prompt engineering. Time constraints and inconsistencies in output and prompt interpretation have prevented us from ascertaining the viability of fully automated development support tooling vor Haskell developers. We are confident that further prompt engineering and experimentation will improve those results drastically, as our analysis suggests the issues to be mostly related to instructional shortcomings. A deeper analysis of LLM supported software testing was also cut due to time constraints.

Recommendations To take advantage of current trends, we recommend further research into prompt engineering to ensure consistent LLM inference. Our developed processes and frameworks, now that their development has been completed, have shown themselves to be valuable aids to accelerate research progress. As of writing, we discourage the unsupervised usage of LLMs to generate, analyse and validate code. Usage when supervised by a professional is recommended and has proven itself as an accelerator in the industry.

Table of Contents

III. Research: Initial Evaluation

A. Initial Evaluation

In the initial evaluation, the aim was to assess the feasibility of the project task, which involved evaluating the capabilities of large language models (LLMs) for software development. To achieve this, the focus was set on determining whether popular, openly available LLMs could generate code from an open task and write tests for it as well, for different target languages. The selected programming languages were:

- C#
- Java
- Python
- Haskell

The languages were chosen as these are the languages where both of the team members had previous coding experience and where there are experts available at OST.

1. *Algorithms:*

Generating sorting algorithms was selected to be the test case for several reasons:

- Simplicity: Sorting algorithms are relatively simple to test, making them an ideal starting point for evaluation.
- Versatility: They allow for a wide range of implementations, which can be easily adapted to focus on different aspects of the algorithms, leading to a diverse set of potential solutions.

2. *Models:* The following LLMs were selected for their ease of use without prior setup:

- Chat-GPT 3.5 by OpenAI[1]
- Llama2 70b by Meta, Model deployed by Replicate[2]

3. *Task:* The LLMs were given the following task:

```
Write LANGUAGE code. It should contain functions which take a generic collection. The output of each is
the same collection in a sorted order. The comparison algorithm is provided to the functions as well.
Using the provided "sort" function of the collection is not allowed. Write functions each of the following
objectives:

Minimize the number of comparisons
Minimize the lines of code
Minimize time complexity
Maximize code readability and simplicity
Then provide a test class to test those functions
```

This prompt was the first step at providing prompts and was not very optimized. It is a bit overloaded, with too many tasks in a single prompt and it reached the output token limits. In those cases, the task was split up into two separate tasks. Other than this split, only one try was allowed for each model and language. The prompt would have needed to be adjusted for all models and languages if it was changed. No automated prompting was implemented yet, so small adjustments would have led to a large increase in time invested to do this initial evaluation. The evaluation of the resulting code showed the following result:

[1]https://chat.openai.com/
[2]https://llama2.ai

Model	Language	Passed assignment	Passed tests
GPT 3.5	Python	✔	✔
GPT 3.5	Java	✔	✔
GPT 3.5	C#	✔	✔
GPT 3.5	Haskell	✔	✔
Llama 2 70b	Python	Failed	Failed
Llama 2 70b	Java	Failed	Failed
Llama 2 70b	C#	Failed	✔
Llama 2 70b	Haskell	Failed	Failed

Figure 1: Result of the initial test

4. *Conclusion:*

Despite Llama 2′s apparent failure in the binary yes/no evaluation for nearly all cases, the actual generated code was found to be nearly functional. The issues that arose, such as ignoring parts of the prompt and having a single wrongly typed function call, seemed as if they could be worked out with improvement of the prompts. The potential shown by Llama 2 warranted its inclusion in the further project.

Haskell was selected as the focus language for this study, as it is less widely used than C#, Java, and Python, and there has been relatively less work done in this area. The composition of code plays a significant role in a functional programming language like Haskell, which is not as central to object-oriented, imperative languages. This led to the additional question how well a sequential thinking LLM, which focuses on probabilities of next tokens in a sequential fashion, can handle compositions of functions through higher order functions. This step would benefit from an approach of having a bigger picture which is then broken down into logical parts and how they interconnect. Finding out whether the LLMs are capable of this, was another point of interest which led to the selection of Haskell.

IV. Analysis: Cognitive Load Theory

A. Introduction to Cognitive Load Theory

Cognitive Load Theory (CLT) is a framework in educational psychology and neuropsychology that allows us to better understand the processes that allow a human being to learn. While it was developed for humans, we can nonetheless borrow aspects of it to analyse LLMs and their performance.

Cognitive Load Theory claims that cognitive load during learning processes is divided into three major categories.

Extraneous load describes how complex a task is to parse. Communication overhead can impact the execution of a task. An intuitive example of this concept is additional extraneous load for non-native speaking students in a learning environment when compared to native speakers.

Intrinsic load describes how difficult a task is to execute. Task difficulty can be influenced by multiple factors, such as the complexity of a task and how familiar an individual already is with adjacent concepts.

Germane load describes how hard it is to build a so-called schema for a task. During learning, humans build schemata that encapsulate cognitive processes. They function as building blocks to be used when executing more complex tasks, allowing humans to abstract away behaviour stored in schemata. This process is called schema automation.

Cognitive Load Theory also provides a model for short-term memory and long-term memory and the interactions in between. While a person can replicate tasks while instructions are still present in the short-term memory, they will be unable to do so in the future unless these instructions have been put into long-term memory as schemata.

"The Evolution of Cognitive Load Theory and the Measurement of Its Intrinsic, Extraneous and Germane Loads: A Review" [1] provides an overview over the intricacies and recent developments in research.

B. Cognitive Load Theory and Deep Learning Models

Applying Cognitive Load Theory to deep learning models is not intended by psychology, is not broadly supported by research, and is mostly based on conjecture on our part. It nonetheless allowed us to frame our analysis and results in an easy-to-understand manner.

Extraneous load in the context of LLMs can be interpreted as the quality of the prompts used to interact with the model. A prompt that contains clear instructions and is well formed with unambiguous phrasing will be easier to parse for an LLM and therefore reduces the extraneous load.

To frame ***intrinsic load*** in the context of LLMs, we have multiple factors contributing to the intrinsic load of a task. Generating a piece of code tends to be easier than finding bugs in existing code. Bubble Sort is easier to generate and analyse than more complex algorithms such as Radix Sort or Introspective sort. Intrinsic load should also increase if the model has encountered few similar data points during training, as the reasoning required to solve a task becomes more complex.

Germane load in the context of LLMs can be framed as the processes of training and fine-tuning as building schemata for later use. This conjecture is supported by the parallels between task specific brain activity [2] and LLMs building internal knowledge graphs whose sub-trees mirror brain regions dedicated to specific tasks. Our project primarily concerns itself with inference, which is why Germane load is not relevant to our work.

Analyzing the load caused by tasks executed by a large language model with this framework allows us to isolate performance issues based on suboptimal prompting (extraneous) from issues caused by the inherent complexity of a task (intrinsic).

Problems caused by extraneous causes typically manifest as the model not understanding the task properly and providing a correct solution for a different task. An example of this phenomenon would be to be tasked with generating an implementation of the introspective sort algorithm and then implementing a correctly working bubble sort algorithm. Another example would be to ignore instructions such as "Only respond with code, do not provide explanations."

Problems caused by intrinsic causes lead to the model generating code that implements algorithms but does so in a faulty or incomplete manner. An issue we encountered often was that a model tasked with the implementation of an introspective sort only used two of the three algorithms contained within the algorithm. The model did understand the task and named the algorithm appropriately but was unable to generate working code.

C. *Applying Cognitive Load Theory in our Project*

Our project aims to analyse the feasibility of LLM usage in the context of Haskell developer support. It must be noted that this includes a human element with the developer, which allows us to apply Cognitive Load Theory to analyse the needs of the developer with the same framework as the capabilities of a model.

We can define the total cognitive load l caused by a task t executed by actor a to be

$$l_t(a) = l_{\text{extraneous}_t}(a) + l_{\text{intrinsic}_t}(a) + l_{\text{germane}_t}(a) \tag{1}$$

$l_{\text{extraneous}_t}(a)$ varies between actors such as humans and language models based on their existing abilities. A student natively speaking the language in which a question is asked will experience less extraneous load when answering than a non-native speaker. The same applies to a large language model that is prompted in a language it was not trained for.

$l_{\text{intrinsic}_t}(a)$ varies between tasks and actors. Tasks inherently have varying difficulty and depending on their pre-existing knowledge required for the execution of task t, actor a will require more complex reasoning in order to execute the task.

$l_{\text{germane}_t}(a)$ varies between tasks and actors. It is the most complex to model and is not relevant to our current project, as the LLM tasked by the developer will merely infer a solution and not create and memorize any schemata. The developer is not in a learning context and is experiencing negligible germane load while attempting to infer a solution to their task.

A use case scenario for an LLM powered development support tool emerges, once the quantifiable form of $l_t(a_1)$ for developer a_1 has reached a threshold specific to each individual developer, which causes developer a_1 to consult LLM a_2 for help. In order for LLM a_2 to be able to assist developer a_1, it must be able to bear the total load of

$$l_t(a_2) = l_{\text{extraneous}_t}(a_2) + l_{\text{intrinsic}_t}(a_2) + l_{\text{germane}_t}(a_2) \tag{2}$$

whereby we can assume $l_{\text{germane}_t}(a_2)$ to be 0.

It must be considered that $l_{\text{extraneous}_t}(a_2)$ not only differs in terms of actor a, but likely also in how the task is communicated. We can assume a loss of information between the instruction received by a_1 and the formulation of an instruction (prompt) for a_2 by a_1. We therefore must redefine $l_t(a_2)$ as

$$l_t(a_2) = l_{\text{extraneous}_{t'}}(a_2) + l_{\text{intrinsic}_t}(a_2) + l_{\text{germane}_t}(a_2) \tag{3}$$

whereby t' is defined as the re-formulated instruction given by a_1 to a_2.

Based on this analysis we work with the following assumptions and limitations:

1. $l_{\text{extraneous}_t}(a)$ will be different between for each language model executing a task, even if we standardize our prompts
2. $l_{\text{extraneous}_t}(a)$ is impossible for us to measure in the time frame provided and must be regarded as a random variable for this project
3. Due to their maturity, we expect all models to have encountered training data that allow them to solve the task we pose them, turning $l_{\text{intrinsic}_t}(a)$ into a de facto constant reflecting the inherent complexity of t
4. $l_{\text{germane}_t}(a)$ can be neglected for the purpose of this project

We can therefore conclude that, in the context of this project, our only plausible approach to adjusting task difficulty in order to evaluate the load bearing capacity of a model should be to evaluate a model with multiple tasks of varying intrinsic complexity.

This analysis is formed based on the general assumption that cognitive load is linear, which has been called into question in recent years [1]. We still present this analysis as is and encourage further research into this topic.

V. Analysis: Algorithmic Complexity

A. Introduction

In order to evaluate the viability of a model as a Development Support Tool, we task the models with a number of use cases involving analysis and generation of algorithm implementations in Haskell. As outlined in Section IV.C ("Applying Cognitive Load Theory in our Project"), we will adjust the difficulty of such a task by changing the algorithm to be worked with.

This allows us to approximate a level of algorithmic complexity that a model is able to reason about. In an attempt to minimize the pseudo-randomness introduced by extraneous load, we standardize our prompts across all models.

There are numerous complexity measures that could be used to identify the complexity of an algorithm. In this section, we will compare two of those measures (Cyclomatic Complexity and Cognitive Complexity) and evaluate their viability as a selection criteria for our project.

B. Complexity Measures

Cyclomatic complexity [3], developed by T.J. McCabe in 1976, is based in graph theory and one of the most used complexity metrics for software. It measures the numbers of linearly independent paths within a piece of code and is used as a proxy variable describing its maintainability and testability.

An example application of cyclomatic complexity on a C# switch statement:

```
Cyclomatic Complexity
switch(model) {
  case "GPT-3.5": return "I am GPT 3.5!";       // +1
  case "GPT-4": return "I am GPT 4!";           // +1
  case "Llama-2": return "I am Llama 2!";       // +1
  default: return "I have no idea what I am!";  // +1
}
                                              // Total: 4
```

Cognitive complexity [4] was developed by SonarSource and aims to focus on code understandability over code testability.

To contrast cognitive complexity with cyclomatic complexity we provide this application of cognitive complexity on the same switch statement:

```
Cognitive Complexity
switch(model) {                               // +1
  case "GPT-3.5": return "I am GPT 3.5!";
  case "GPT-4": return "I am GPT 4!";
  case "Llama-2": return "I am Llama 2!";
  default: return "I have no idea what I am!";
}
                                              // Total: 1
```

Among other factors, cognitive complexity also puts more emphasis on the avoidance of nesting. Consider the following applications of cyclomatic and cognitive complexity, respectively:

```
Cyclomatic Complexity
var numbers = new List<int>();
for(int i = 0; i < 99; i++) {     // +1
  for(int j = 0; j < 99; j++) {   // +1
    if(i % j == 0) {              // +1
      numbers.Add(i*j);
    }

  }
}
                              // Total: 3

Cognitive Complexity
var numbers = new List<int>();
for(int i = 0; i < 99; i++) {     // +1
```

```
  for(int j = 0; j < 99; j++) {   // +1+1
    if(i % j == 0) {              // +1+1+1
      numbers.Add(i*j);
    }

  }
}
                                // Total: 6
```

While not overly complex, most people would consider this code snippet to be harder to understand than the first code snippet containing a switch statement, even though cyclomatic complexity gave it a lower score. Cognitive complexity punished the algorithm for its nested nature and greatly increased the score.

For a full list of all increment generators the official whitepaper can be consulted.

Research into which of these values provide a closer approximation to the perceived understandability of an algorithm is sparse, but initial studies [5] suggest that cognitive complexity matches the subjective impression of the involved developers more closely.

C. Applications in our Project

When considering our goal, we have to focus on what might cause a developer to consult a development support tool to begin with. Selecting and ranking the algorithms for the models to work with should be based on the perceived difficulty to understand the algorithm, as a developer will be able to quickly implement an easy-to-understand algorithm on their own.

While cyclomatic complexity serves as a proxy value for this property, cognitive complexity explicitly focuses on the selection for this property and appears to outperform cyclomatic complexity. We therefore have chosen cognitive complexity as our selection metric for the sorting algorithms to be used in the tasks executed by the models we plan to evaluate. We additionally are working with the following assumptions:

1. Code which is harder to understand is also harder to write.
2. This applies to both humans and LLMs.

   We therefore not only select for the likelihood of a developer consulting a development support tool when facing a specific algorithm, we can also adjust the difficulty of the tasks for the LLMs during the evaluation stage with cognitive complexity as the deciding factor.

   A limitation of this approach is that cognitive complexity must be measured on a sample implementation. Implemented algorithms in the form of code have their own degree of optimization. Two implementations of the same algorithm can have differing cognitive complexity scores. We are accepting the risk that our sample algorithms will not be fully optimized and therefore can have slightly distorted cognitive complexity scores.

VI. Evaluation: Model Performance

A. Model Selection

In the process of model selection, input was gathered from colleagues, as well as from online platforms such as the HumanEval competition on Papers with Code[3].

The model selection criteria included a requirement for the models to achieve relatively high scores, specifically being among the top 10 models in the human evaluation, and the availability of the models through an API.

After considering these criteria, several options emerged, including GPT 3.5/4, PaLM 2, and CodeLlama. Ultimately, GPT 3.5/4 was chosen based on its performance, while CodeLlama was preferred over PaLM 2 due to the latter being announced to be superseded by Gemini, which was released too late in the project for evaluation.

Other options that were initially considered, such as StarCoder[4] and Mistral[5], were not included in the project due to the limited time available to evaluate additional models. This decision was made to prioritize the thorough evaluation of the selected models in accordance with the project's timeline and resources.

Additionally, Llama 2 was selected to compare its performance with CodeLlama, as both models are available through the same channel and in a similar manner, making the comparative analysis cost-effective.

B. Use Cases

While there are multiple definitions of the software development steps, the following was selected as a guideline to select specific use cases for the usage of LLMs in software development:

- Planning & Requirement Analysis
- Design, Architecture
- Coding, Implementation, Documentation
- Testing
- Integration
- Maintenance

Planning & Requirement: While resource planning could be an interesting usage for LLMs a lot of existing data would be needed to finetune a model to this specific use case. This would be outside of the scope of this project. Requirement engineering is very specific for each individual project. Finding all the relevant data would be a project by itself, this would not fit the scope of this project as well.

Design, Architecture: There are interesting potential use cases for LLMs like creating Bill of Materials for a given setup, or suggesting architectures or suggesting specific elements for a given problem. However, there is only limited knowledge on the project team on this topic, so it was not selected.

Coding, Implementation, Documentation: Several use cases arise from this step, which can be evaluated directly, without the need of a surrounding project.

Coding: LLMs can be used to generate code snippets or even entire programs based on descriptions provided by users. ***Benefit:*** Could save time and effort for programmers, could make new programming environments easier to access.

Implementation: LLMs can be used to guide the implementation process by providing recommendations on how to implement specific features or solve specific problems. ***Benefit:*** Could help to make better decisions and avoid pitfalls while coding.

Documentation: LLMs can be used to generate documentation for code, including user guides, API documentation, and commentary within the code itself. ***Benefit:*** Reduces the time and effort required for creating documentation, allowing developers to focus on more complex tasks, could improve the overall quality of the documentation, making it more useful and accessible to users and other developers.

---

[3]https://paperswithcode.com/sota/code-generation-on-humaneval

[4]https://huggingface.co/blog/starcoder

[5]https://docs.mistral.ai/

Debugging: LLMs can be used to analyse code for potential bugs and to provide possible fixes for those or other known bugs. ***Benefit:*** Early detection of software bugs makes them less costly to fix, could save time and effort for code reviews and Bug Fixing.

Testing: LLMs can be used to generate test cases and test data for software, ensuring that it meets the requirements and is free of bugs. They could pinpoint reason behind the failing tests suggest fixes. Benefit: This can help developers improve the quality of their software and reduce the time and effort required for testing.

Integration: LLMs could be used to generate testcases for integrating software components to ensure them working together properly, to integrate data from different sources to design and optimize workflows and more. For our case those cases would require multiple two+-party systems for the LLMs to be evaluated on. This would increase the complexity of the project beyond a point which could be achieved with the available time.

Maintenance: LLMs could be used during implementation to make code more readable and easier to understand, maintain and update. They could also be used to help managing and maintaining different versions of code or to automate the integration and deployment process. However, for these use cases the potential gain over existing conventional solutions seemed not as big.

1. *Selected use cases:* The following use cases were selected:
   - Code Generation
   - Documentation Generation
   - Debugging: Identification
   - Bug Fixing
   - Test Generation

Originally the use case Test Evaluation was included. Due to the development of the custom testing suite being more complex than initially estimated, the automated testing suite was only available after the evaluation phase has passed already. For that reason, Test Evaluation was cut from the scope and Test Generation was implemented but not evaluated in the final evaluation.

Test Evaluation, Debugging: Identification and Bug Fixing were deliberately kept separate from each other as combining use cases would create more possible chain links where a small change in one part could lead to a big change in another part. This would increase the evaluation complexity enormously when quantitative measures are used of it would reduce the meaning of the evaluation if a binary yes / no approach was chosen, because the number of runs without a part of the chain having a defect would diminish. For that reason, each use case was kept to a single functionality.

C. Evaluation Criteria

For each selected use case an initial list of possible metrics was collected form coworkers and Prof. Dr. Mehta as a Haskell expert Section XIII.G. His feedback pointed out the difficulty of assessing the quality of Haskell code. A difference in quality assessment to regular imperative code is, that the code should be written in a way that makes it easy to proof to be correct. The quality of code composition and the chaining of functions through higher level functions are key in assessing Haskell code.

After an initial review the decision was made to forgo qualitative criteria and instead to use a list of binary yes/no evaluations, based on the qualitative criteria. The criteria were then weighted against each other on a scale from 1-5.

The criteria were grouped based on their functionality: Generation, Debugging, Testing.

The weights were then normalized to a total of 100 for each group, so the groups get an identical amount of the overall final score. Finally, the used criteria evaluation weights were divided by 3, as this would make it easier to get a final percentage of the score, with a maximum of 100 over all groups.

This led to the following criteria and scores:

Proposed Score	Usecase		Score	Subtotals	Group tota	Grand total	Normalized we	Normalized overall
	Generation							
	Use Case 1: Assist developers in generating Haskell code for specific tasks or components.							
3,47		i. Syntactically correct haskell?	5				10,42	3,47
3,47		ii. Compiles?	5				10,42	3,47
3,47		iii. Correct functionality?	5				10,42	3,47
1,39		iv. No Code smell? (see below)	2				4,17	1,39
0,69		v. Appropriately commented?--	1				2,08	0,69
2,78		vii. Readable (fly through-> understand)?	4				8,33	2,78
				22				
0,00		a. Uses Adjustement: Use e. instead					0,00	0,00
2,08		b. Uses Adjustment: Appropriately chained (not all in one and not everything sperated)	3				6,25	2,08
2,78		e. Uses type system to express and enforce constraints?	4				8,33	2,78
							0,00	0,00
				7				
	Use Case 2: Automatic Documentation Generation: Automatically generate code documentation, including function and module descriptions, based on the source code.							
2,78		a. Documentation is generated for all functions / modules / components?	4				8,33	2,78
2,78		b. Documentation covers all relevant details (specified in prompt?) for all components?	4				8,33	2,78
2,78		c. Documentation is correct, reflects actual behavior and usage?	4				8,33	2,78
2,08		d. Consistent style?	3				6,25	2,08
2,78		f. Readable (fly through->understand)?	4				8,33	2,78
				19				
					48		100,00	33,33
	Debugging							
	Use Case 1: Bug Identification: Assist developers in identifying and categorizing bugs.							
4,50		a. Detects all potential bugs?	5				13,51	4,50
1,80		b. Can categorize into different appropriate types?	2				5,41	1,80
2,70		c. Can categorize into different severity levels?	3				8,11	2,70
2,70		d. Appropriate severity levels?	3				8,11	2,70
3,60		e. Produces not a significant amount of false positives (threshold to be defined in accord to	4				10,81	3,60
3,60		f. Points out correct location?	4				10,81	3,60
3,60		g. Provides appropriate descriptions?	4				10,81	3,60
				25				
	Use Case 2: Debugging Assistance: Provide developers with debugging hints, suggestions, and potential fixes for identified issues.							
4,50		a. Suggestion fixes issues?	5				13,51	4,50
2,70		b. Are the corrections appropriate? (e.g. Length, accuracy)	3				8,11	2,70
3,60		c. Explicit code? (vs. Higher level sugestion)	4				10,81	3,60
				12				
					37		100	33,33

Figure 2: Evaluation criteria with their normalized score.

Originally it was intended to include Testing as a group, those criteria were also defined. As mentioned in Section VI.B.1: Selected Use cases we could not reach the stable state for automated testing in time. This is why they are not shown here. Removing that group from the evaluation table did not alter the outcome. It only moved the maximum overall achievable score to 66.67 instead of 100. By calculating the final score through percentages, adjustments like this do not affect the final scores.

The proposed scores were once more discussed with the Haskell expert Prof. Mehta. He pointed out that the evaluation method made sense overall but again mentioned that the quality of code composition is key in Haskell. This evaluation would not be feasible in an automated fashion and would require a lot of experience on the domain. Because of this, he recommended holding a presentation in front of "Zürich Friends of Haskell" the organizers of ZuriHack[6] which he is part of. Maybe some more ideas could be gathered about evaluating code quality. While the presentation led to an interesting further discussion, the main takeaways were the following thoughts:

- Should non compiling code be evaluated at all? Automated testing will fail either way.
- Would it make sense to evaluate the output of a Haskell language server when presented with the output code to assess quality?
- Further reassurance that evaluating Haskell code on a compositional level is complex.

Upon further discussion in the project team, the following decisions were made:

- Non compiling code will still be evaluated, albeit manually. Running the prompts multiple times and measure the frequency of compiling, functionally correct code would be another possible measure. However, the single evaluation run was chosen, where each model gets each use case with 3 levels of complexity and only gets one try each. The evaluation will then have the same structure over all use cases.

-Measuring the number of warnings or errors from a Haskell language server would be an interesting metric, however it does not represent the actual code quality. Especially in Haskell it is possible that the programmer intends to write in a subpar way

[6]https://zfoh.ch/zurihac2023/

from the language server's point of view but with the intention to improve the overall code composition and, for example, make it easier to proof. As this metric is not strongly tied to code quality, the decision was made to not evaluate it.

-Additional criteria of the use case Code Generation were removed as assessing the Haskell specifics other than the appropriate chaining and using strong typing were too complex for this project. 1. *Haskell Clean Code:* In order to define code smells, online research was conducted but as no collection of smells was found, a set of relevant smells was collected from discussion boards such as discourse.haskell.org[7]

If generated code contains one or more of these code smells they will not pass that criterion.

- Monolithic Functions: Writing overly long and complex functions instead of breaking them down into smaller, more manageable pieces. This makes the code harder to understand and maintain.
- Excessive Type Signatures: Writing overly verbose type signatures when type inference can handle them more succinctly.
- Monads Everywhere: Overusing monads when simpler abstractions or functions could be used.

-Complex Pattern Matching: Excessive nested pattern matching, especially when more concise solutions are possible.

-Unused Variables or Bindings: Leaving unused variables, function parameters, or import statements in the code.

-Inconsistent Style and Formatting: Inconsistent code style, formatting, or naming conventions within a codebase.

-Excessive Dependencies: Including unnecessary or redundant external dependencies in your project, which can increase complexity and maintenance overhead.

-Side Effects

-No Separation of Concerns

D. Sample Algorithm Selection

As we have established in Section IV.C ("Applying Cognitive Load Theory in our Project"), we are adjusting the intrinsic load caused by a task by changing the sorting algorithm the model has to implement or analyse. In Section V.C ("Applications in our Project") we concluded that cognitive complexity, which selects for code understandability, is a better fit for our work than cyclometric complexity.

To measure the cognitive complexity of an algorithm, we are calculating the cognitive complexity of a selection of sorting algorithm implementations. We then pick three algorithms to be used during testing.

We classify these algorithms as either "Simple", "Intermediate" and "Complex".

To reduce inconsistencies in implementation quality, we ensured that the recursion depth of no implementation exceeds 3.

Consider the following sorting algorithms, implemented in C#, and corresponding cognitive complexity ratings.

```
                                                                    // Cogn.
public int[] BubbleSort(ref int[] array) {                          // +1
    for (int i = 0; i < array.Length - 1; i++) {                    // +1
        for (int j = 0; j < array.Length - i - 1; j++) {            // +1+1
            if (array[j] > array[j + 1]) {                          // +1+1+1
                var tempVar = array[j];
                array[j] = array[j + 1];
                array[j + 1] = tempVar;
            }
        }
    }
}
                                                                // Total: 7

                                                                    // Cogn.
public static void ShellSort(ref int[] array) {                     // +1
    int n = array.Length;
```

[7]https://discourse.haskell.org/t/important-things-to-know-about-writing-good-haskell-code/7302/2

```
    for (int split = n / 2; split > 0; split /= 2) {                                // +1
        for (int i = split; i < n; i += 1) {                                        // +1+1
            int temp = array[i];

            int j;
            for (j = i; j >= split && array[j - split] > temp; j -= split) {        // +1+1+1+1
                array[j] = array[j - split];
            }
            array[j] = temp;
        }
    }
}
                                                                                    // Total: 8

                                                                                    // Cogn.
public static void PancakeSort(ref int[] array, int n) {                            // +1
    for (int curr_size = n; curr_size > 1; --curr_size) {                           // +1
        int mi = FindMax(array, curr_size);
        if (mi != curr_size - 1) {                                                  // +1+1
            Flip(array, mi);
            Flip(array, curr_size - 1);
        }
    }
}

public static void Flip(int[] arr, int i) {                                         // +1
    int temp, start = 0;
    while (start < i) {                                                             // +1
        temp = arr[start];
        arr[start] = arr[i];
        arr[i] = temp;
        start++;
        i--;
    }
}

public static int FindMax(int[] arr, int n) {                                       // +1
    int max, i;
    for (max = 0, i = 0; i < n; ++i) {                                              // +1
        if (arr[i] > arr[max]) {                                                    // +1+1
            max = i;
        }
    }

    return max;
}
                                                                                    // Total: 10

                                                                                    // Cogn.
public static int[] RadixSort(ref int[] array) {
    int max = array[0];
    for (int i = 1; i < array.Length; i++) {                                        // +1
        if (array[i] > max) {                                                       // +1+1
            max = array[i];
        }
    }

    for (int digit = 1; max / digit > 0; digit *= 10) {                             // +1
        int[] buckets = new int[10];

        for (int i = 0; i < array.Length; i++) {                                    // +1+1
```

```
            int bucketIndex = array[i] / digit % 10;
            buckets[bucketIndex]++;
        }

        int index = 0;
        for (int bucket = 0; bucket < buckets.Length; bucket++) {                 // +1+1
            for (int i = 0; i < buckets[bucket]; i++) {                           // +1+1+1
                array[index++] = bucket * digit;
            }
        }
    }
}
                                                                                  // Total: 11
                                                                                      // Cogn.
public static void IntroSort(ref int[] data) {                                    // +1
    int partitionSize = GetPartitionSize(ref data, 0, data.Length - 1);
    // Arbitrary threshold set to 32
    if (partitionSize < 32) {                                                     // +1
        InsertionSort(ref data);
    }
    else if (partitionSize > (2 * Math.Log(data.Length))) {                       // +1
        HeapSort(ref data);
    }
    else {
        QuickSortRecursive(ref data, 0, data.Length - 1);
    }
}

private static void InsertionSort(ref int[] data) {                               // +1
  for (int i = 1; i < data.Length; ++i) {                                         // +1
    int j = i;

    while ((j > 0)) {                                                             // +1+1
      if (data[j - 1] > data[j]) {                                                // +1+1+1
        data[j - 1] ^= data[j];
        data[j] ^= data[j - 1];
        data[j - 1] ^= data[j];

        --j;
      }
      else {
        break;
      }
    }
  }
}

private static void HeapSort(ref int[] data) {                                    // +1
  int heapSize = data.Length;

  for (int p = (heapSize - 1) / 2; p >= 0; --p) {                                 // +1
    MaxHeapify(ref data, heapSize, p);
  }

  for (int i = data.Length - 1; i > 0; --i) {                                     // +1
    int temp = data[i];
    data[i] = data[0];
    data[0] = temp;
```

```
    --heapSize;
    MaxHeapify(ref data, heapSize, 0);
  }
}

private static void MaxHeapify(ref int[] data, int heapSize, int index) {         // +1
  int left = (index + 1) * 2 - 1;
  int right = (index + 1) * 2;
  int largest = 0;

  if (left < heapSize && data[left] > data[index]) {                              // +1+1
    largest = left;
  }
  else {
    largest = index;
  }

  if (right < heapSize && data[right] > data[largest]) {                          // +1+1
    largest = right;
  }

  if (largest != index) {                                                         // +1
    int temp = data[index];
    data[index] = data[largest];
    data[largest] = temp;

    MaxHeapify(ref data, heapSize, largest);
  }
}

private static void QuickSortRecursive(ref int[] data, int left, int right) {     // +1
  if (left < right) {                                                             // +1
    int q = GetPartitionSize(ref data, left, right);
    QuickSortRecursive(ref data, left, q - 1);
    QuickSortRecursive(ref data, q + 1, right);
  }
}

private static int GetPartitionSize(ref int[] data, int left, int right) {        // +1
  int pivot = data[right];
  int temp;
  int i = left;

  for (int j = left; j < right; ++j) {                                            // +1
    if (data[j] <= pivot) {                                                       // +1
      temp = data[j];
      data[j] = data[i];
      data[i] = temp;
      i++;
    }
  }

  data[right] = data[i];
  data[i] = pivot;

  return i;
}
                                                                                  // Total: 24
```

As one would expect, introspective sort turned out to be the by far most complex algorithm we evaluated. This is not surprising, as it is a combination of multiple sorting algorithms that have been combined by conditional execution logic. Its comparatively high cognitive complexity score led to it being selected as our “Complex” algorithm.

For our “Simple” algorithm, we chose the lowest scoring algorithm, which turned out to be bubble sort.

With introspective sort scoring more than twice as high as the second highest scoring algorithm, radix sort, we decided to use radix sort as our algorithm of “Intermediate” difficulty.

E. Prompting

Large Language models (LLM) are a fundamental component in natural language processing (NLP) that provides a foundation for various applications such as machine translation, text generation, and speech recognition. A key aspect of LLM’s functionality lies in its ability to generate output based on prompts, which essentially guides the model in producing the desired results.

In LLM, prompting refers to the input text given to the model to generate a particular output. The prompt is the model’s initial cue, and its quality directly influences the output. A well-formed prompt can guide the model to generate relevant, comprehensive, and contextually accurate responses, ensuring high-quality output. Prompting is valuable in LLM because it:

- Drives Contextual Understanding: Good prompts help the model understand the context of the task, thereby generating more accurate and relevant responses.
- Enhances Coherence: The quality of prompts impacts the coherence and logical consistency of the LLM’s output. A well-structured prompt can guide the model to produce a coherent narrative.
- Ensures Relevance: The precision of the prompt determines the relevance of the output. Specific, clear, and concise prompts can help the LLM produce highly relevant and focused content.
- Influences Creativity: The nature of the prompt can stimulate the LLM’s creativity. An open-ended or challenging prompt can encourage the model to generate innovative and diverse solutions.

In this project the guidelines to prompt engineering provided by OpenAI[8] were applied where adequate.

1. *Measures:* Measures included:

- “Include details in your query to get more relevant answers”: While adjusting the prompts the tasks were formulated as broadly as possible until at least 3 out of the 4 evaluated models understood the task and produced output that had the correct shape / functionality. This threshold was defined to be the cutoff for prompt engineering to define a specific endpoint.
- “Ask the model to adopt a persona”: LangChain provides functionality to feed prompts from different entities to a model. Firstly a “system” prompt was sent where the persona of the model was defined. Afterwards a “human” prompt was sent as the user input to which the model should produce an output. The system prompt was defined once for each use case.

  An example of the used system prompt for Code Generation was the following:

  ```
  You are a helpful Haskell code generating assistant with a focus on generating compiling, well typed and readable Haskell code that solves the given task. The output should only contain code and no surrounding explanation.
  ```

- “Use delimiters to clearly indicate distinct parts of the input”: For simple Debugging use cases two code variants were used. For that a clear separation of inputs was made using labels for separated blocks.

  ```
  prompt = "Description: " + code_description + "\n\nCode version 1:\n" + code_v1 + "\n\nCode version 2:\n" + code_v2
  ```

  This could have been expanded more for example using a specific delimiters and referencing them in the prompt, but the models were already able to understand the task given this structure.

[8]https://platform.openai.com/docs/guides/prompt-engineering/six-strategies-for-getting-better-results

- “Split complex tasks into simpler subtasks”: The prompts were kept to a single task, as differentiating between the difficulty of tasks was one of the key goals of this project.
- “Give the model time to think”: Prompting the model to use an inner monologue by instructing to solve the task, then checking for specific properties like type correctness, then adjusting if necessary and only then to output the code was tried out with GPT 3.5 and GPT 4.

First, we will implement insertion sort. Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort:

```
insertionSort :: (Ord a) => [a] -> [a]
insertionSort = foldr insert []
where
insert x [] = [x]
insert x ys@(y:ys') = if x < y then x:ys else y:insert x ys'
```

Next, we will implement heapsort. Heapsort is a comparison-based sorting algorithm. Heapsort can be thought of as an improved selection sort: like selection sort, heapsort divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element and moving that to the sorted region.

```
heapSort :: (Ord a) => [a] -> [a]
heapSort = heapsort' . heapify
where
heapify [] = []
heapify xs = foldr siftDown [] xs

heapsort' [] = []
heapsort' xs = x : heapsort' xs'
where (x, xs') = deleteMax xs

deleteMax [] = error "deleteMax: empty heap"
deleteMax h@(Node x xs) = (x, heapify xs)

siftDown x [] = Node x []
siftDown x h@(Node y ys)
| x >= y = Node x (y:ys)
| otherwise = Node y (x:ys)

data Heap a = Node a [a] deriving (Eq, Show)
```

Lastly, we will implement quicksort. Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays.

```
quickSort :: (Ord a) => [a] -> [a]
quickSort [] = []
quickSort (p:xs) = (quickSort lesser) ++ [p] ++ (quickSort greater)
where
lesser = filter (< p) xs
greater = filter (>= p) xs
```

Figure 3: Output of GPT 4 prompted to reflect on the intial solution and adjusting it before showing it, part 1

Now that we have our three sorting algorithms implemented, we can create our introsort function:

```
introsort :: (Ord a) => [a] -> [a]
introsort xs = introsort' (2 * (floor . logBase 2 . fromIntegral $ length xs)) xs

introsort' :: (Ord a) => Int -> [a] -> [a]
introsort' _ xs | length xs < 16 = insertionSort xs
introsort' depthLimit xs
| depthLimit <= 0 = heapSort xs
| otherwise = merge $ introsort' (depthLimit - 1) <$> splitAtPivot xs
where
splitAtPivot xs = let p = medianOfThree xs in partition (<=p) xs
medianOfThree xs = median $ take 3 xs
merge (xs, ys) = quickSort xs ++ quickSort ys
```

Here, `introsort'` is the main driver function for introsort. It first checks if the length of the list is less than 16. If it is, it uses insertion sort. If the recursion depth limit has been reached, it falls back to heapsort. Otherwise, it proceeds with quicksort.

The `medianOfThree` function is a helper function that returns the median of the first, middle, and last elements of the list. This is a common strategy in quicksort implementations to improve performance on partially sorted lists.

The `splitAtPivot` function partitions the list into elements less than or equal to the pivot and elements greater than the pivot.

Finally, the `merge` function combines the two sorted lists produced by the recursive calls to `introsort'`.

Figure 4: Output of GPT 4 prompted to reflect on the intial solution and adjusting it before showing it, part 2

2. *Chain of Thought:* While GPT 4 had no significant changes in code quality in either direction, the code output format of code only was no longer respected and more surrounding commentary was given. GPT 3.5 seemed to over adjust the code output, adjusting code and leaving no longer used imports and functions.

```haskell
import Data.List (sortBy)
import Data.Ord (comparing)

introsort :: Ord a => [a] -> [a]
introsort xs = if length xs < 2 then xs     else introsort' xs

introsort' :: Ord a => [a] -> [a]
introsort' [] = []
introsort' [x] = [x]
introsort' xs = let pivot = head xs
                    smaller = introsort' (filter (< pivot) xs)
                    equal = filter (== pivot) xs
                    larger = introsort' (filter (> pivot) xs)
                in smaller ++ equal ++ larger

sortWithIntrosort :: Ord a => [a] -> [a]
sortWithIntrosort xs = introsort xs
```

Figure 5: Output of GPT 3.5 prompted to reflect on the intial solution and adjusting it before showing it

Prompting the model to output its chain of thought lead to comparable results, with a tendency to write proper functions but creating errors while chaining them together, like chaining functions with wrong types or using or non-defined functions to chain them (see `median` in the Chat GPT 4 example).

This could be an additional hurdle for generating Haskell code, as the sequential oriented predicting of the next token approach of LLM's might struggle a bit more with the compositional style starting from the bigger picture and planning to chain functions together at a later point.

Additionally, another reason for the difficulties with this approach might lie in the amount of training data available. With the lower usage of Haskell in comparison to Python, C# and Java, there is fewer training data available on the web. However, the available code tends to be more rigid, but more correct, because of Haskell's strong type system, which leads to fewer ways to solve a specific problem with Haskell code. That rigidness can cause further problems for LLMs, as small adjustments, which might not seem that important for the LLM, in the chaining of functions through higher order functions can lead to substantial changes in execution logic. An example of that can be seen in the Simple Debugging examples, where the swap from `foldr` to `foldl` (iterate over a list, applies a function to the element and the previous result, starting at either first or last element), which alter the code to either sort in the wrong order or not sort at all, was not detected by any LLM.

VII. Testing: Model Performance

A. Results

1. *Overall Result:*

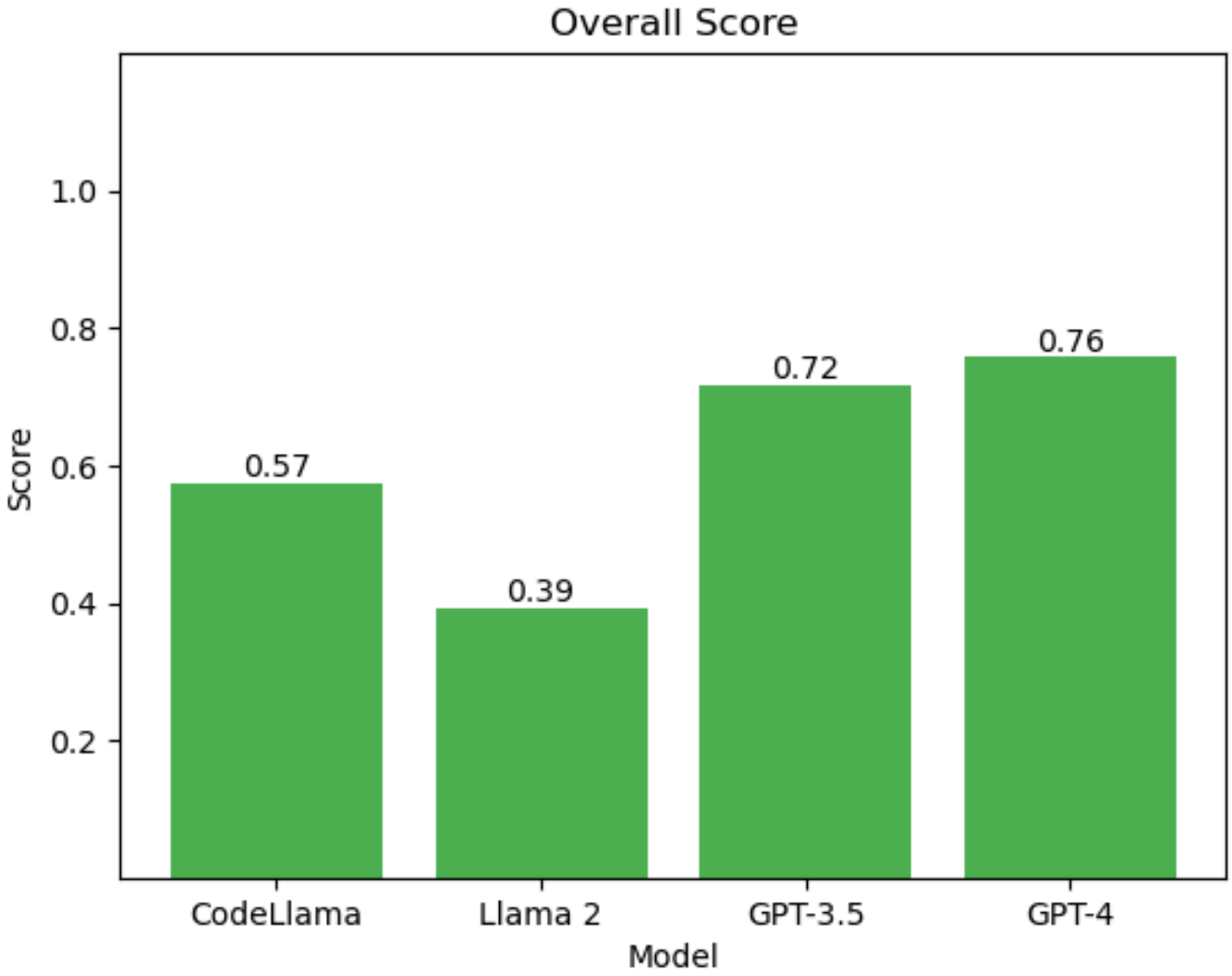
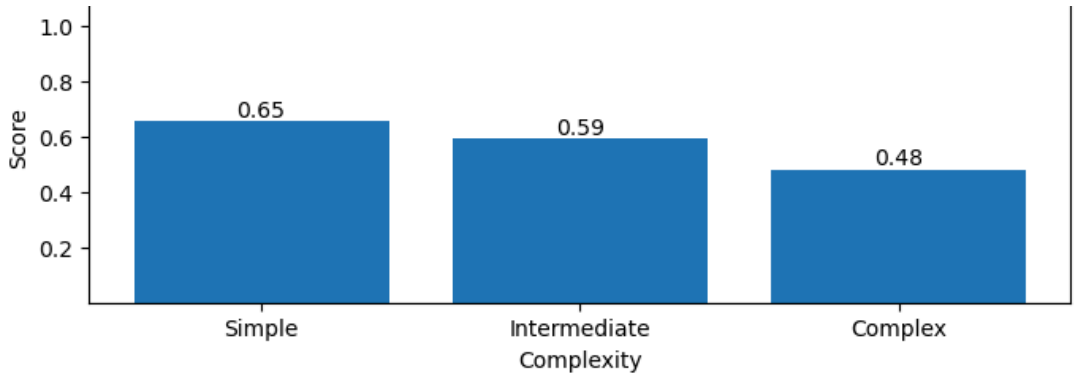
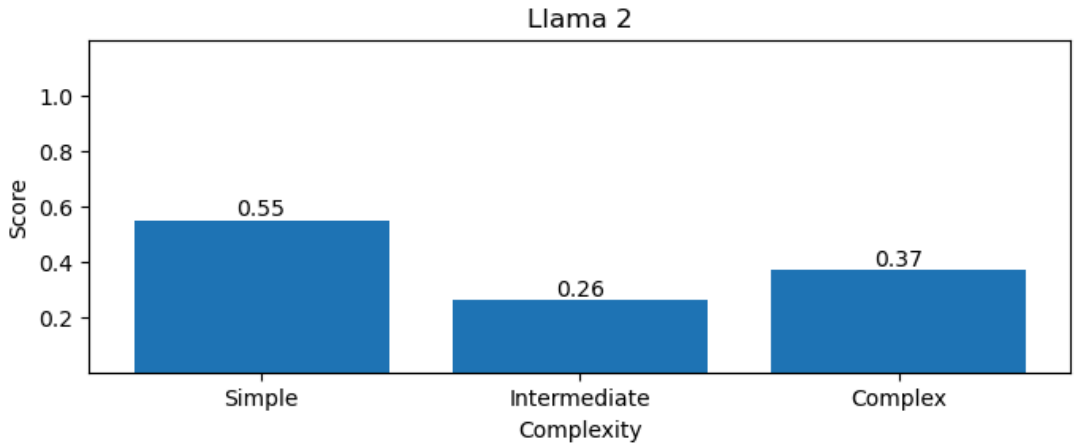
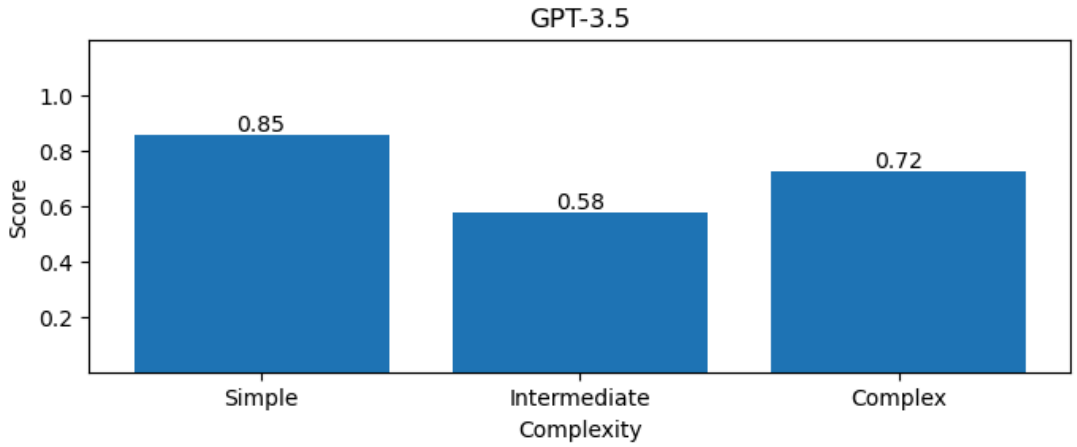


Figure 6: Average score total over all use cases, normalized to [0:1]

2. *Result per complexity for each model:*







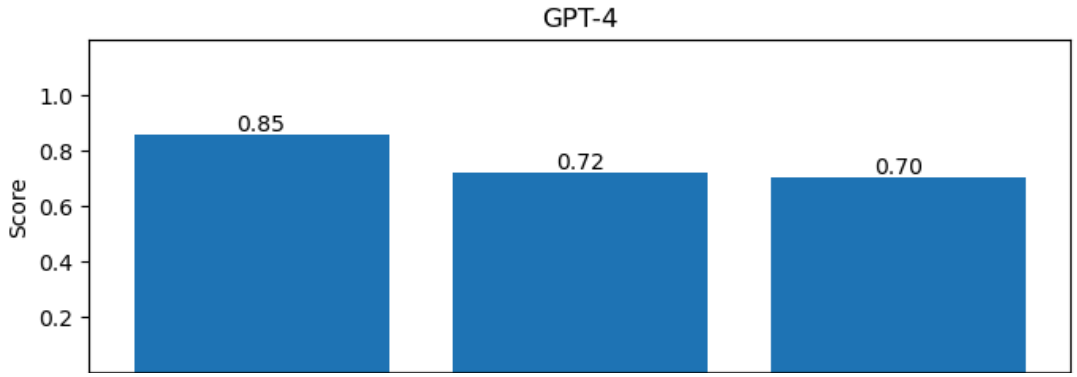
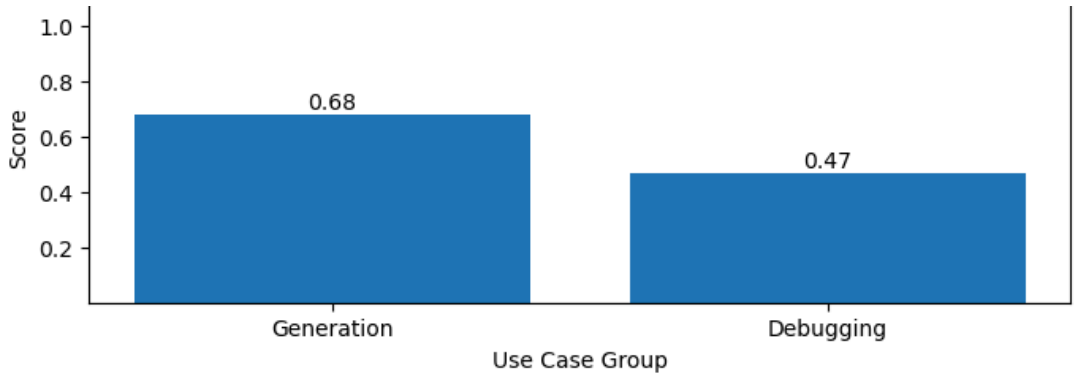
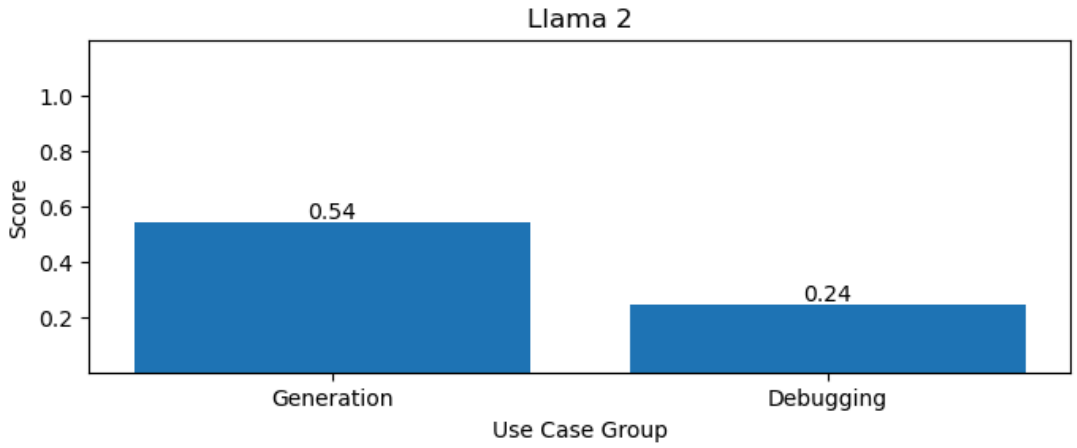
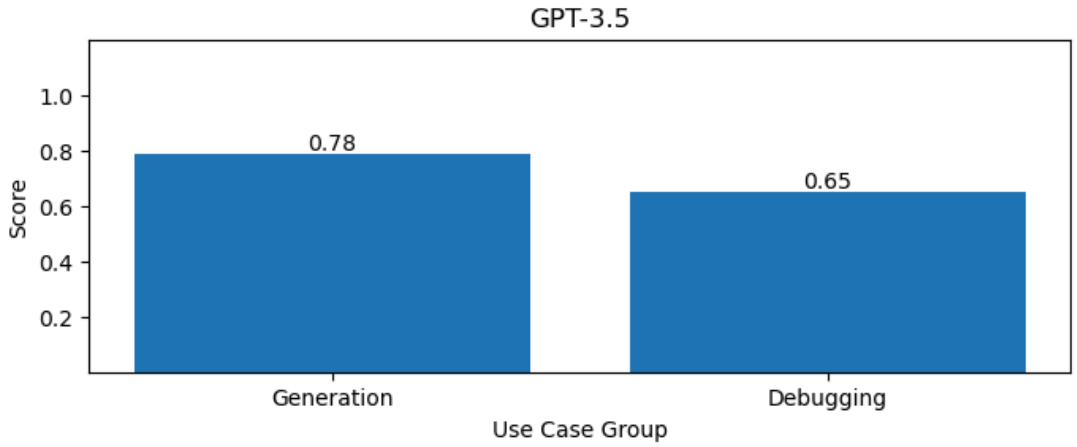


Figure 7: Average score total over all use cases per complexity

3. *Result per model per usecase:*







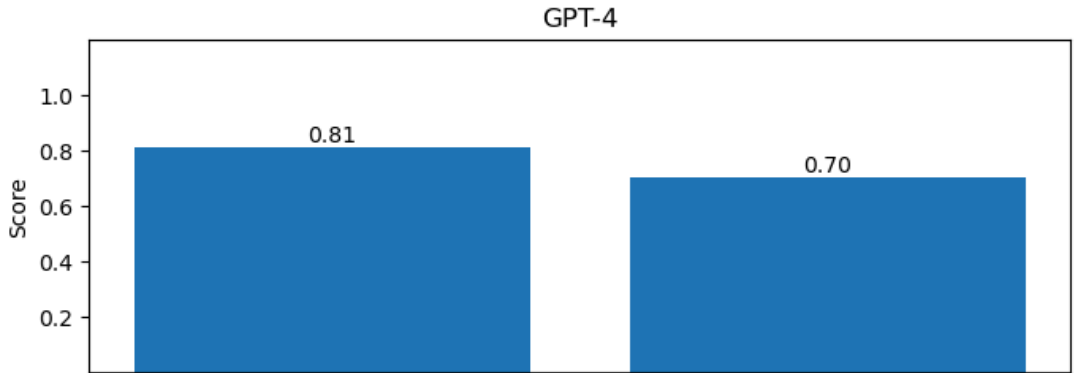


Figure 8: Average score over each use case

4. *Outliers:*

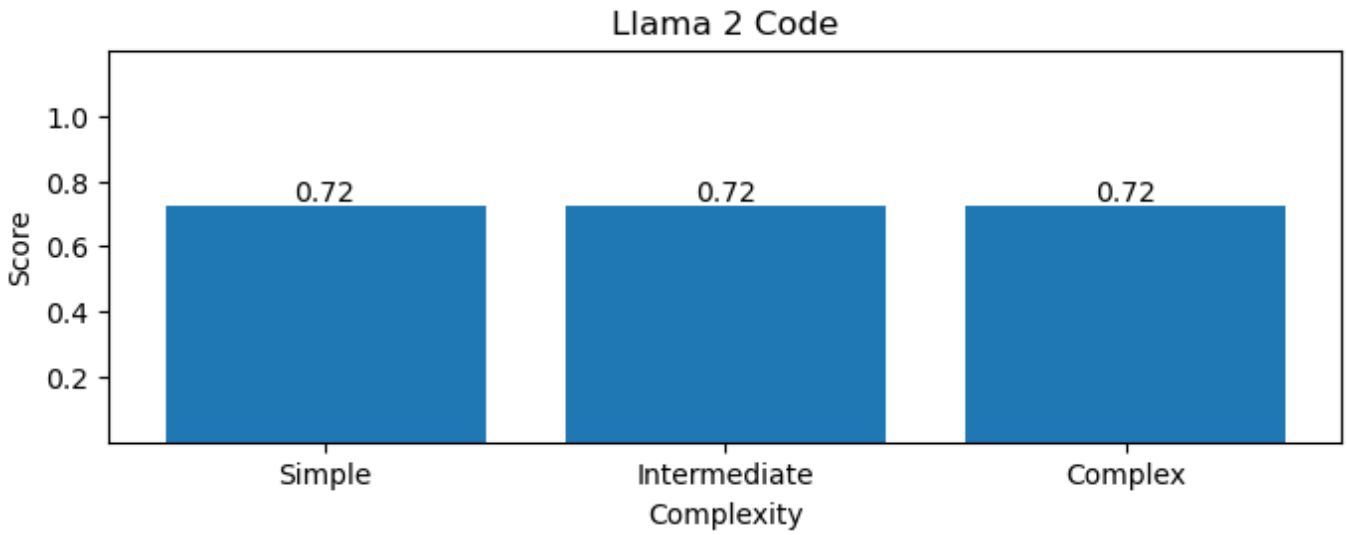


Figure 9: Llama 2 score for Code Generation

Llama 2: scoring 0.72 over each complexity of Code Generation. This was reached by generating compiling code that solves the tests but not with the functionality it was asked to implement (e.g. implementing merge sort instead of introspective sort)

```haskell
import Data.List (sort)

introsort :: (Ord a) => [a] -> [a]
introsort [] = []
introsort [x] = [x]
introsort xs =
    let (ys, zs) = splitAt (length xs `div` 2) xs
    in merge (introsort ys) (introsort zs)



merge :: (Ord a) => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
    | x <= y = x : merge xs (y:ys)
    | otherwise = y : merge (x:xs) ys
```

Figure 10: Output of Llama 2 for Code Generation of introspective sorting

GPT 3.5 was able to create a functionally correct, compiling implementation of the complex use case, introspective sort. During the prompting phase, where the tasks were run repeatedly until the prompts were deemed good enough, this model was not able to create compiling code for this task once. It seems to have overperformed on the evaluation run.

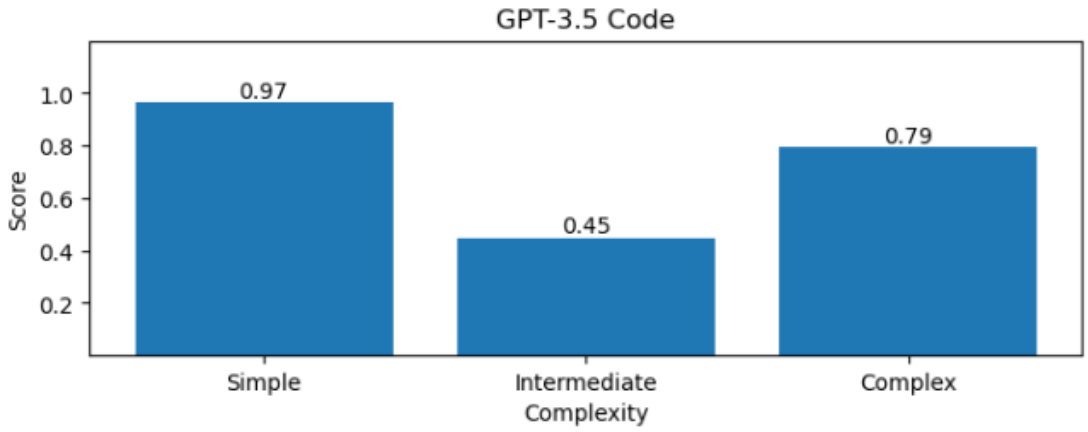


Figure 11: GPT 3.5 score for Code Generation

B. Hypothesis Testing

For the hypothesis that the LLMs can fulfil each task, a threshold was set at a score of 0.75 for each run. While evaluating the prompting runs, the threshold was set to that value, as most of the outputs that were deemed to solve the given task properly scored at that value of higher. The hypothesis could be accepted for the following use cases and models:

Task	Simple	Medium	Complex
Bugfixing	GPT 3.5 GPT 4	GPT 4	
Bug Identification	GPT 4	GPT 4	
Code Generation	CodeLlama GPT 3.5 GPT 4		GPT 3.5 GPT 4
Docu Generation	CodeLlama GPT 3.5 GPT 4	CodeLlama GPT 3.5 GPT 4	CodeLlama GPT 3.5 GPT 4

Figure 12: Table of evaluations reaching at least 0.75

C. Insights

1. *General performance:* Excluding Llama 2, which was mainly used as a comparison to CodeLlama, all models did well in the documentation and explanation of presented Haskell code.

For the Code Generation use case, even though GPT 3.5 and GPT 4 achieved a similar score, GPT 4 was more consistent in understanding and solving more complex tasks. CodeLlama tended to not quite follow the given task to implement. Often this

still led to still functional code but with the wrong implementation by ignoring guidelines set by the task (e.g. implementing merge sort instead of introspective sort). For the Bug Identification use case only GPT 4 managed to be somewhat consistent in finding most of the implemented bugs. All other models overlooked at least one bug and created false positives, like marking to use >= instead of > in a sorting algorithm as a bug, where both operators would lead the identical result. A common occurrence was the tendency to repeat patterns in the answer. Often time nearly identical structures were reused for multiple different answers.

Example: Bug identification CodeLlama simple:

```
The function bubbleSort is not defined correctly. It should take a list of elements as input, but it is defined as a function that takes a single element and a list of elements.

The function swapFill is not defined correctly. It should take two lists of elements as input, but it is defined as a function that takes a single element and a list of elements.

The function swapFill is not defined correctly. It should return a list of elements, but it is defined as a function that returns a single element.
```

2. *Llama 2 vs. CodeLlama:* There was a clear distinction between the output quality of CodeLlama and Llama 2. While CodeLlama and GPT 3.5 behaved in a similar fashion, Llama 2 was sometimes quite uncooperative, reaching from ignoring the task (Bug Fixing intermediate, full answer)

```
Here is your code with known bugs fixed!
```

To just repeating back the provided code (Bug Fixing complex). There seemed to be large issues in regard to code interpretation especially when compared to the finetuned to responding to coding questions CodeLlama instruct model. As both models are originally based on the same base model, this was quite an interesting effect of what difference in quality on a specific topic finetuning is able to achieve.

3. *Overall performance GPT 4:* Chat GPT 4 overall did not only score the highest, but its answers also tend to include additional useful information and tend to give more precise reasoning than the other models. Example: Identical Bug for Bug Identification simple

GPT 4:

```
2. Bug: Syntax error in swapTill definition.
Description: In the function swapTill, a syntax error occurs at 'swapTill max x y xs', the correct usage should be 'swapTill (max x y) xs'. The parentheses are needed in order to correctly parse the function calls.

Category: Syntax error
Severity: High
Location: swapTill function
```

GPT 3.5:

```
2. In the 'swapTill' function, the recursive call 'swapTill max x y xs' is incorrect. The correct syntax should be 'swapTill (max x y) xs'.

Severity: High
Type: Compilation error
Location: swapTill function
```

VIII. Discussion

A. Interpretation

1. *GPT 4 performance:* GPT 4 outperforms or performs as well as all other tested models over all use cases. According to [6], the modelsize is of two level greater magnitude than the other evaluated models, reported to be at around 1.8 trillion parameters. In comparison to that, the evaluated CodeLlama model with 13 billion paramters and GPT 3.5 turbo with 20 billion parameters according to [7], seem to be perfoming quite well. However this implies that the steps to improvement get more expensive the better the result gets. Reaching a user satisfaction of 'great' instead of 'very good' is exponetially harder than getting a 'good' verdict instead of an 'ok'.

2. *Decent performance on code documentation:* Overall GPT 3.5, GPT 4 and CodeLlama performed well in the documentation and explanation of presented haskell code. This was to be expected, as every function that needed to be described was either documented online and as such likely in the training data, or a composition of multiple other functions for which the same argumentation applies. Further tests could be done to test whether the models are able to differentiate small nuances in the code composition, which lead to a different execution logic. They were not executed in this project because of time constraints.

3. *Issues with Bug Fixing:* All models seemed to struggle with suggesting alternative Haskell code when presented with a known bug. One of the root causes seemed that the bugs were once more part of compositions which the LLMs seemed to not quite understand. Finding out how much of the code was affected by a single bug and replacing exactly that part proved to be the most difficult task evaluated for all models. This once again likely goes back to the difference in approaching Haskell code sequentially versus compositionally. One possible path of improving the output quality of LLMs in that regard might be presenting them with an example of working code to reference as stated by OpenAI[9]. This approach is based on having a working solution beforehand, which makes it not that useful for a professional setting, but in a teaching environment this could be a valuable option.

4. *Issues with Code Generation:* The most common issue that arose that could not be handled through more explicit prompting was the improper chaining of functions. Compiler errors like the following were the most common reason why a code output would not compile. A reason for that might be analog to Bug Fixing, the difficulties for sequentially working LLMs trying to grasp the compositional, chained style of functional programming.

Example: CodeLlama Code Generation Intermediate compiler output:

```
main.hs:8:37: error:* Couldn't match expected type Int -> a0′ with actual type Int'
    * In the second argument of on', namely lastDigit'
      In the first argument ofsortBy', namely (compare on lastDigit)'
      In the expression: sortBy (compare on lastDigit) xs
  |
8 |       sorted = sortBy (compare on lastDigit) xs
  |
```

Based on the output code provided by CodeLlama:

```
import Data.List (sortBy)

radixSortLSB :: [Int] -> [Int]
radixSortLSB [] = []
radixSortLSB (x:xs) =
  let lastDigit = x mod 10
      sorted = sortBy (compare on lastDigit) xs
  in x : radixSortLSB sorted
```

5. *Prompt quality:* It could be argued that in use cases where the evaluation showed that no model passed the hypothesis of a specific complexity, the prompt would have needed further improvement. However, with the variance in output quality, getting a consistent output quality requires many evaluations over the same prompt, to confidently say that the prompt is good enough.

[9]https://platform.openai.com/docs/guides/prompt-engineering/strategy-give-models-time-to-think

With the given time restriction, a decision had been made to stop prompt engineering as soon as at least 3 models understand the concept of the task. At that point the prompt was no longer adjusted, and the evaluation run was made with that prompt.

6. *Output variance:* A single try was evaluated instead of a "pass at X" evaluation as each evaluation requires costly manual evaluation. This alongside the before mentioned point of prompt quality caused more variance in results, leading GPT 3.5 being able to solve the complex Code Generation task once during the evaluation, but never before, or the Llama 2 model refusing to try to solve a given task that it might have not understood on that run.

7. *Strengths and weaknesses of binary evaluation:* Weighted binary evaluation criteria allowed the measurement of qualitative metrics. This made more efficient evaluation possible, as creating clear quantifiable metric for qualitative measurements would be a time-consuming task which requires a lot more domain knowledge than was available for the project.

One of the main weaknesses of tallied binary evaluations on the other hand was the lack of distinction between barely failing an evaluation and completely failing it as well as treating an excellent output the same as one barely passing the requirements. This led to code nearly pass the threshold for the Code Generation hypothesis even though that code did not do what it was asked to do at all. As only the functionality criteria would be failed. Also, non-compiling code could still score above 0.5 which increased the average of those models. All in all, those drawbacks had to be accepted to create quantifiable evaluation criteria. These drawbacks can be reduced by creating extra metrics for the frequency of non-compiling or excellent code which are run multiple times and averaged out.

B. Limitations

Cognitive load theory is not designed to be applied to the inner workings of a large language model. This usage is experimental and can still turn out to not be an accurate approach to framing LLM performance.

Utilizing cognitive complexity as the selection criterion for our sample algorithms has introduced some imprecision as it needs to be applied to existing code implementations, which could still be improved upon. C# code has been used as the basis for the complexity evaluation due to pre-existing experience of the project members and it trivializing the evaluation of cognitive complexity. This might introduce further imprecision, as Haskell might lend itself to writing certain algorithms more easily than others.

Due to time constraints our prompts are relatively rudimentary and have potential for optimization. Further prompt engineering is encouraged, as it would alleviate several inconsistencies in our results.

Another factor to this inconsistency was the sample size. While we tested a large variation of use case, model and algorithm criteria, the number of individual runs per combination is low. By increasing this sample size and with the aggregation of the results, more representative results can be achieved.

C. Applications

This project provides tooling and lays a conceptual foundation for future LLM use case evaluation projects. We have shown how one can frame both user and model with cognitive load theory and encourage continuous research and into this usage of cognitive load theory.

Our evaluation results suggest that the usage of LLMs as an interactive tool for Haskell developers is a viable method to increase productivity for tasks of low to medium complexity. An experienced Haskell developer will not be able to gain value from this, as they are skilled enough to implement such algorithms on their own.

The successful execution of complex tasks can be achieved with prompt engineering, but as a Haskell developer one might be better off executing the task yourself.

Purely based on the results of this project, we discourage the fully automated use of LLMs as a code generation and validation tool. But we acknowledge that our insights are gained from a very low sample size, which means that they are likely to not be representative. Using this environment to run further tests would provide clarity in this regard. We also acknowledge that with more advanced prompting techniques and agent setups we might see drastically improved performance and encourage further development of the framework built with this project.

D. Reflection: Dominik Castelberg

I thoroughly enjoyed working on this project. It allowed me to take a deep dive into topics that I am only tangentially familiar with (Large Language Models, Algorithm Analysis) and explore concepts that I was unfamiliar with (Cognitive Load Theory).

Using the Cognitive Load Theory and framing the interactions with an LLM through this lens was something I have not yet heard of before and was something I quickly felt a sense of ownership over, which was a massive motivating factor.

Linus Flury was a fantastic collaborator who hit the ground running and was quick to upskill where necessary. I deeply appreciate his efforts and am looking forward to working with him again in the future.

Prof. Dr. Mitra Purandare's supervision was superb and she was a guiding light in an increasingly stressful project. Her commitment to our sync meetings and genuine interest in our work felt empowering and reassuring.

I still cannot deny that this application of cognitive load theory was (and still is) experimental and requires a considerable amount of conjectures and risk-taking. Especially towards the end of the project I felt like that I spent too much time on something that might not even be received in a positive light due to its experimental nature. Spending more time on validation to solidify my claims was not possible without tanking the rest of the project. It is nonetheless still something I will spend time with in other projects or on my free time, as I believe it to be valuable if only just to model human-model interactions.

Having an extensive network of experts between Prof. Dr. Farhad D. Mehta, his network and Haskell enthusiasts and my colleagues at Zühlke enabled us to seek feedback when necessary and was a big factor for our success.

The testing environment was originally planned to be an accelerator for our testing based use cases, but ended up costing more time than it would have saved. Not only that, but its delayed progress stalled Linus with the testing use cases and kept me busy for far too long. It ended up functioning and can be used as a tool for future endeavors, but it did not help our progress overall.

I feel saddened by the fact that I didn't get to spend another 20 hours on this project due to work related pressure and private issues, as I feel like that we were forced to rush in the end. Especially the documentation was rushed and does, in my opinion, not reflect the quality of our work to its fullest. We started formally documenting our progress too late, which is something we will most definitely correct for our BA.

In general I would argue that all our biggest issues were caused by time management. We spent too much time on some parts of our project which ended up costing us part of the originally planned scope.

E. Reflection: Linus Flury

Overall, the project was a valuable learning experience, and I am grateful for the opportunity to explore a more practical approach to my selected study specialization. The constant evolution of the area, as evidenced by the release of CodeLlama and Google's Gemini, made the project even more interesting and relevant. I particularly enjoyed working on Prompt Engineering, which was unfortunately cut short due to the stopping criterion. However, it sparked a desire in me to further explore the topic, even outside of the university context.

Working with Prof. Purandare was a pleasant experience, as she provided additional ideas, showed us new developments and trends, and offered critical feedback that helped us improve our project. Her approach of letting us work independently, while still providing guidance, helped us realise better what went well and what went wrong with our style of project work. This is valuable information and an important base for the Bachelor thesis.

Collaborating with Dominik was a smooth and efficient process. We had open communication, were available when issues arose, and were able to work independently on our respective parts of the project after laying the groundwork for the project planning. Our respectful and coordinated work dynamic was a key factor in the project's success.

Prof. Mehta's generosity with his time and domain knowledge was invaluable, and I am grateful for his quick responses and feedback. His spontaneous invitation to present in front of Haskell experts was an exciting opportunity that we appreciated.

However, there were some challenges during the project, such as suboptimal time management. Overworking in the middle of the semester led to a lack of free time, which resulted in a reduction of workload for non-immediate tasks. This caused features on the long-term plan to be worked on properly, but the ongoing task of documenting current workitems was postponed. This led to a debt that had to be paid off in the final weeks of the semester, which could have been avoided with better time management.

Additionally, features that overstepped the originally planned deadlines, such as the testing environment, led to resources being used up to finish them, causing delays for following tasks. I believe we should have been more strict and transparent about the origin of the additional resources needed for these workitems. This would have prevented situations like the final parts of the testing stage being finished after the evaluation phase, making it unable to be used for automated evaluation. Instead, manual

evaluation could have been considered, but this is a decision between investing time to gain more knowledge on testing for future projects and obtaining immediate results for the current project. This is something that should have been openly discussed during the evaluation phase.

Another point was the lack of consistency in the created Jupyter notebooks. Initially, Llama 2 and CodeLlama created some trivial issues while integrating them into the Jupyter notebooks. However due to time concerns at that point, it was decided that at that point, getting the additional data is more important at that step, so further conclusions can be drawn from the results. This led to manual passing of the prompts to the locally deployed models. This was represented in the notebooks. However a later adjustment to the notebooks to reflect the correct implementation of the cria API would have led to a need of reevaluating all Llama 2 and CodeLlama notebooks. Out of time concerns this had to be skipped. For GPT a similar issue arose, when a potential swap to Azure was made. We were able to test the functionality and kept it in 1 Notebook, Code Generation Complex GPT 4. Again, adjusting the other notebooks would have required another evaluation run. All in all this left the Jupyter notebooks in an inconsistent state.

In conclusion, the project was a valuable experience, and I am eager to apply the knowledge and skills I gained to future endeavors. To improve our approach in the future, I believe it is essential to include a definition of "done" for workitems that includes the documentation of said workitem. Additionally, being more aware of available resources and justifying the use of resources for workitems that require more investment would be beneficial. Overall, the project was a success, and I am grateful for the opportunity to learn and grow through this experience.

F. Thank You!

We would like to thank our supervisor, Prof. Dr. Mitra Purandare for her continued support and insight throughout the project. Her guidance allowed us to navigate a project that quickly grew in scope and complexity while still letting us explore new concepts and technologies.

We would also like to thank Prof. Dr. Farhad D. Mehta for his consultation and inspiration regarding all things Haskell. He provided the expertise, experience and networking opportunities required to develop our evaluation criteria which were critical for the outcome of our project.

Finally, we would like to thank Dominik's colleagues at Zühlke Engineering AG, which provided advice, inspiration and consultation for anything related to LLM development and prompt engineering.

IX. Infrastructure & Dev Environment

X. Infrastructure

A. Visual Studio Code Setup

We recommend the usage of Visual Studio Code for this project and provide an `extensions.json` file with recommended Visual Studio Code extensions in the attachments Section XIII.C, and in the separately delivered attachment ZIP file. While other IDEs and setups can be used to work in this project, but utilizing the provided resources will ensure a smooth onboarding procedure for new contributors.

B. Conda Environment

A conda environment that contains all relevant Python packages has been created and exported as a YAML file that can be imported during the project setup stage. A copy can be found in the attachments Section XIII.A, and in the separately delivered attachment ZIP file.

This approach allows us to ensure an easy way to onboard new project members and serves as package versioning for the development environment.

C. Cloud Resources

An Azure Resource creation template with parameters can be found in the attachments Section XIII.B, and in the separately delivered attachment ZIP file.

The following models have been deployed to our Azure Tenant:

#	**Name**	**Type**	**Version**	**API Version**
1	OpenAI GPT-3.5 Turbo	Microsoft.CognitiveServices/accounts/deployments	0613	2023-10-01-preview
2	OpenAI GPT-4	Microsoft.CognitiveServices/accounts/deployments	0613	2023-10-01-preview

D. Llama Infrastructure

The models of Llama 2 and CodeLlama were run locally (deploying them onto Azure was not within bidget). Access to the Llama 2 and CodeLlama models was granted by Meta upon request, however in the end the finetuned ggml file from huggingface[10][11] were used instead, as it showed to be easier to get running.

The 13 billion parameter models were selected, as the 70 billion parameter models require more memory than was available privately (64+ GB). The 13 billion models only required 16+ GB of memory.

Cria[12] was used to provide an API which mimics OpenAI's API, so the models can be accessed with LangChain in the same fashion as other models within the Jupyter notebooks. Additionaly, Cria was the reason the GGML files were used instead of the superseding (21.08.2023 onwards) GGUF files.

For the deployment Docker was used with the adjusted docker-compose file.

E. Typst

Typst[13] was used to create the documentation. It is a rust based alternative to Latex currently in beta. It sells itself as a tool which is easier to learn in comparison to Latex, with simpler implementation of common features. It was selected after discussions with other fellow students writing their SA in typst. Initial testing and having the possibility to discuss issues with fellow student gave enough confidence to use it for the documentation creation.

[10] https://huggingface.co/TheBloke/CodeLlama-13B-Instruct-GGML
[11] https://huggingface.co/TheBloke/Llama-2-13B-GGML
[12] https://github.com/AmineDiro/cria
[13] https://typst.app/

XI. Dev Environment

A. *Overview*

The Python project `LLM Assisted Development` serves as the primary research and development environment.

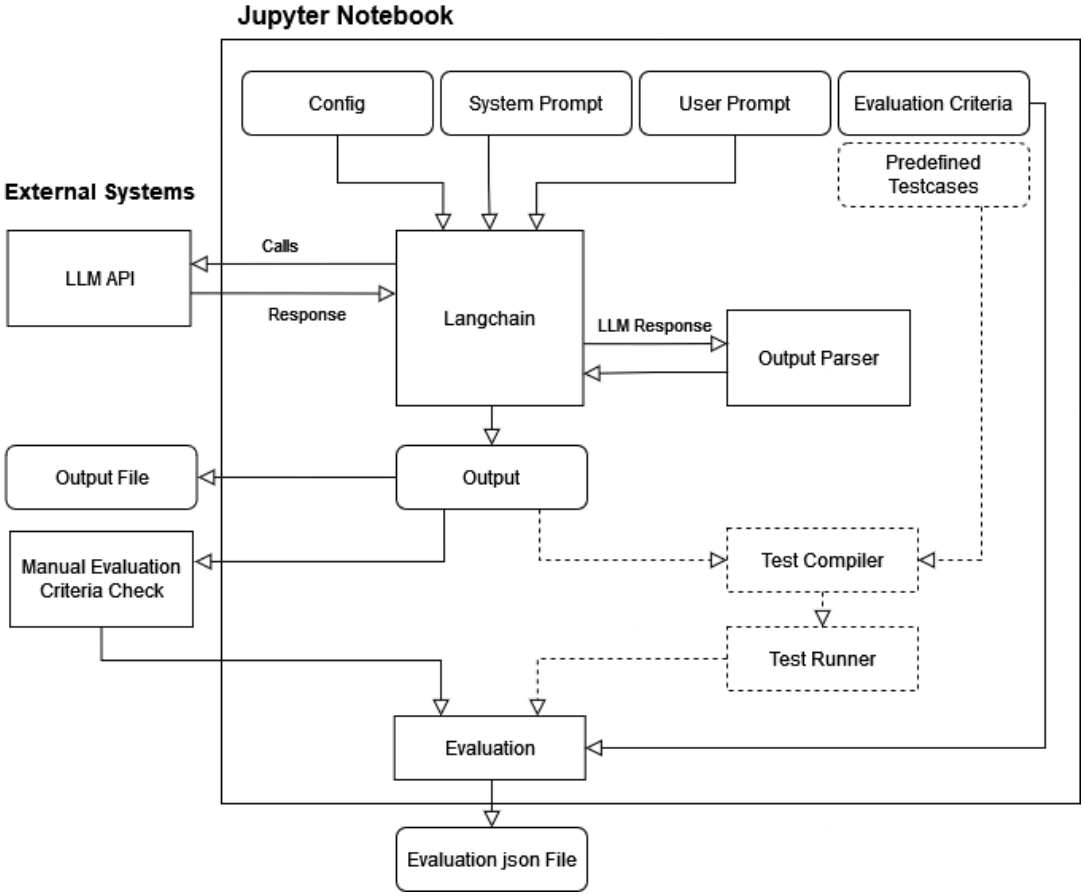
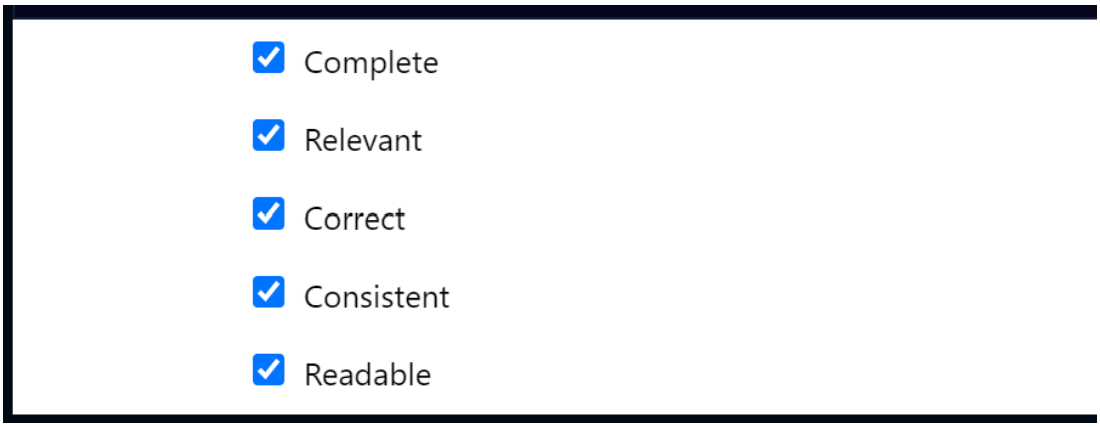


Figure 13: Jupyter Notebook Setup

B. *Jupyter Notebooks*

The configuration of the models and the evaluation criteria available in the environment and accessed by the notebooks. Each notebook is provided with a user and system prompt. LangChain is then used to connect to the LLMs' APIs, delivering the prompts, receiving back an output. LangChain then provides the output to an output parser. That result then gets parsed further by custom parsing (like extracting code blocks for Code Generation). This output is the base for the evaluation. The notebook provides a checklist where criteria can be ticked or unticked.



The content of that checklist is stored as a .json file for each notebook. The criteria then get connected with the scores of the evaluation criteria and the model's performance gets calculated. That score is the baseline for accepting or rejecting the underlying hypothesis that that specific model is able to handle the given task adequately.

For Code Generation and Testing, it was originally planned to have an automated test execution environment, where a test compiler would place the generated or predefined tests in a structured setting and compiles the tests. A test runner would then run those tests on the Code output from the model. In the end however, the test environment was only finished after the evaluation phase. While it is present in the final state of the project, it is not used for automated evaluation.

An example notebook can be found in the attachments: Section XIII.E

C. Test Environment

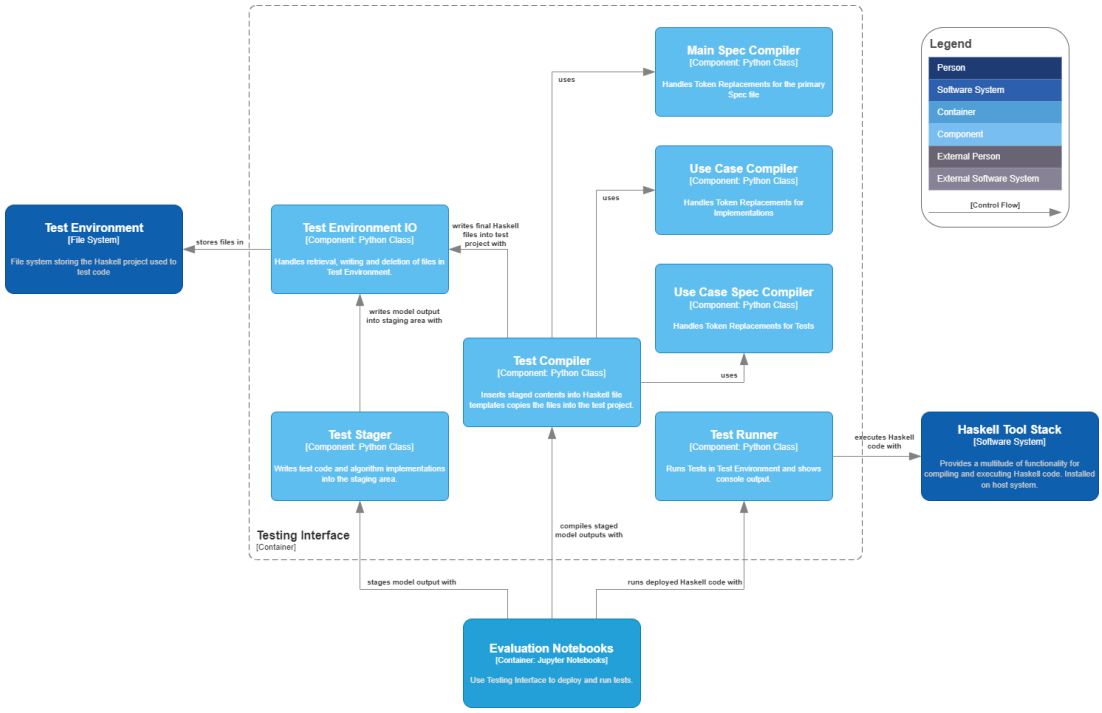


Figure 15: Test Environment Overview

The test environment provides a Haskell project in which algorithm implementations and tests generated by an LLM can be staged, compiled and run from within the Jupyter notebook they were generated in. This saves some manual effort on the side of the developers.

A sample usage would be:

```
# "algorithm" arguments must match -> "BubbleSort" type algorithms will not be tested by "RadixSort" type tests
TestStager.New_Test("BubbleSort", "Demo", sample_test)
TestStager.New_TestCase("BubbleSort", "Demo", sample)
TestCompiler.CompileTests()
TestRunner.RunTests()
```

D. Test Stager

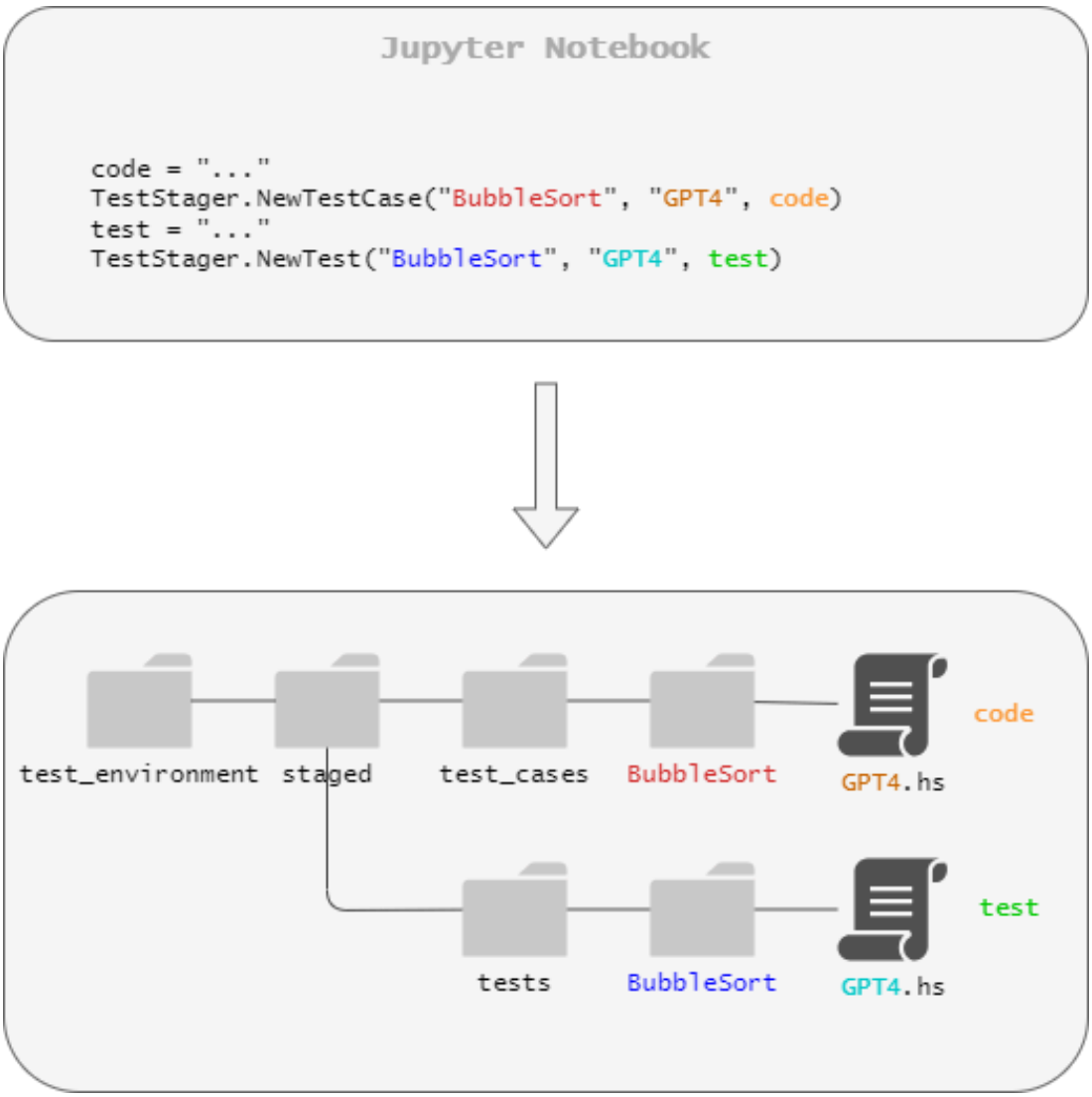


Figure 16: Input/Output of the Test Stager Component

The test stager takes on the code generated by an LLM and stages it for compilation.

As an example, `TestStager.New_TestCase("BubbleSort", "GPT-4", generated_implementation)` saves the content of `generated_implementation` in a file called `GPT-4.hs` located in the `test_cases/BubbleSort` subfolder of the staging area.

`TestStager.New_Test("BubbleSort", "GPT-4", generated_test)` saves the content of `generated_test` in a file called `GPT-4.hs` located in the `tests/BubbleSort` subfolder of the staging area.

E. Test Compiler

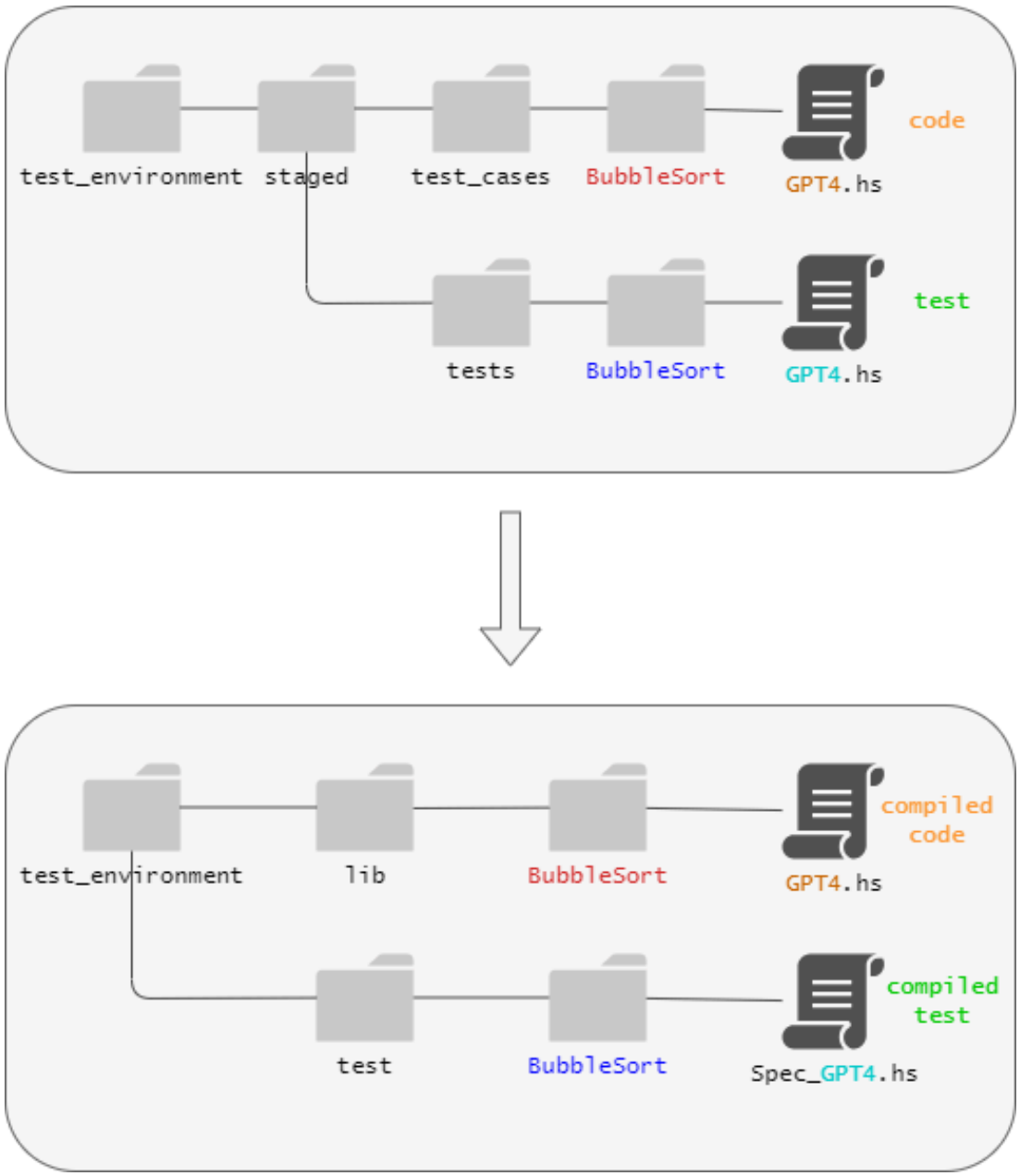


Figure 17: Input/Output of the Test Compiler Component

"Test Compiler" is a bit of a misnomer for this component, as no actual compilation takes place. Instead this component picks up any staged tests and test cases, inserts them into a template and stores them in the corresponding subfolder in the Haskell project.

The nature of this process demands that test cases implement a function with the type definition `sort :: Ord a => [a] -> [a]` and that tests implement a function with the type definition `sort :: Ord a => [a] -> [a] -> Spec`.

F. Test Runner

The test runner class utilizes the Haskell Tool Stack[14] (local installation required) to execute the tests staged and compiled into the test environment. Tests are based on the testing framework HSpec[15]. Output is provided to `stdout`, conveniently providing live updates to the Jupyter notebook user.

[14]https://docs.haskellstack.org/en/stable/
[15]https://hackage.haskell.org/package/hspec

XII. Project Plan

A. Initial Project Idea

The initial project idea was posed by Dominik Castelberg and his supervisor at the consulting firm Zühlke Engineering AG. Its scope was loosely defined, with the goal of allowing Dominik to gain hands-on experience with large language models. It was outlined as research into ways in which large language models can be utilized to support software engineers. He approached Linus Flury as a potential project partner. Linus soon after joined the project team.

Prof. Dr. Mitra Purandare showed interest in the project and expressed her willingness to supervise the project.

During the first weeks of the project the scope of work was refined. Haskell as the primary tech stack to focus on has been established in this phase, so was the set of models (GPT-3.5 Turbo, GPT-4, Llama 2, Code Llama) to evaluate.

B. Project Organization

As this section refers to the common understanding of the project team, it is written in the active form.

Our development process is oriented towards the waterfall methodology, defining time boxed development steps. We hold weekly meetings where we update each other on the current progress and plan the next steps ahead. The planning of the next steps should be orientated towards the completion of the next milestone.

1. *Project Meetings:* Meetings are held on Wednesday 8:00 to 9:00 ahead of the supervisor meetings. We discuss the current progress, what needs to be discussed at the supervisor meetings and what the next steps are.

At the beginning of November, the team decided to meet up Sundays 10:00 at OST until the completion of the project. This was implemented as we realised, we did not put in enough work hours until that point and had to make a commitment to meet in person and work alongside each other in person.

2. *Project Roles:* As a development team of two, we only assign responsibilities to areas of work instead of assigning project roles. This leads to more efficient work on topics and produces less overhead.

3. *Assigned Areas:*

Dominik Castelberg:

- Test Environment
- Research into theoretical background
- Process definition
- Initial code setup
- Defining algorithms to evaluate
- Azure deployment

Linus Flury:

- Initial evaluation
- Evaluation except Testing
- Local model deployment
- Generate result graphs
- Prompting

All other areas like documentation fall under a shared responsibility.

4. *Involved active stakeholders:*

- Dominik Castelberg (Project team)
- Linus Flury (Project team)
- Prof. Dr. Mitra Purandare (Supervisor)
- Zühlke Engineering AG (General Advisory)

5. *Involved parties:*

- Prof. Dr. Farhad Mehta (Haskell Domain Expert)

- Zürich Friends of Haskell (Group of Haskell Domain Experts)

C. Development Cycles

At the beginning of the project, the project members developed a framework of processes to structure their work and plan their tasks in each project phase.

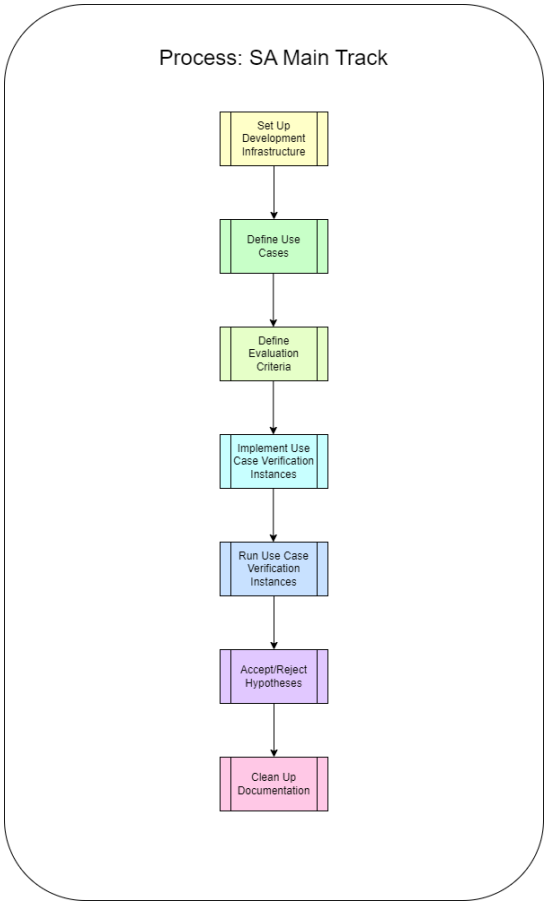


Figure 18: Process: Main Track

The main track process describes the overall project with its individual phases. Each project phase is described in its own process.

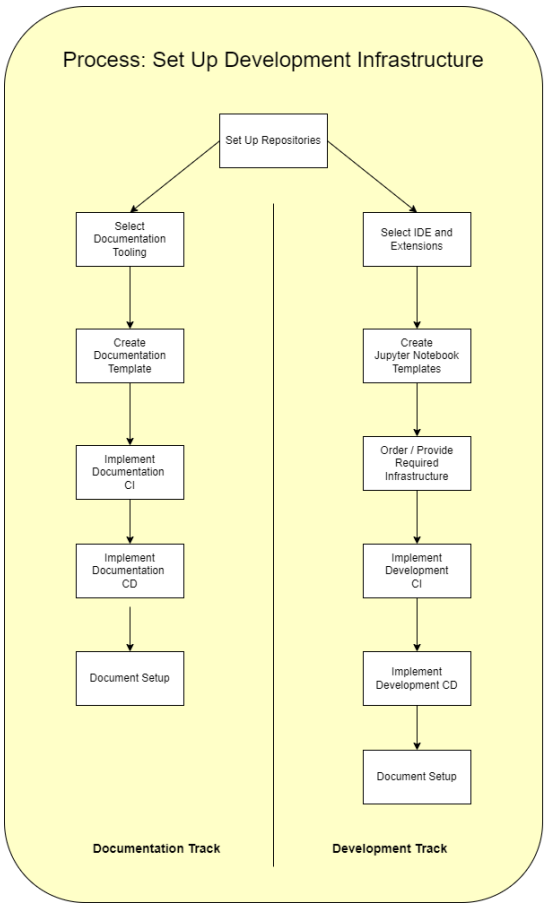


Figure 19: Process: Set Up Development Infrastructure

This process is divided into two tracks: Documentation and Development. This is due to the project setup with two git repositories, one containing the documentation and one containing the development environment.

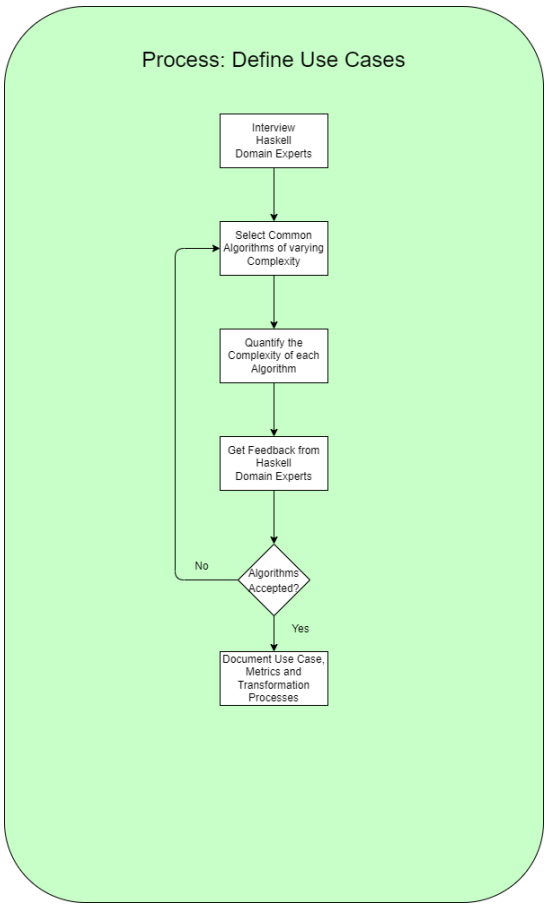


Figure 20: Process: Define Use Cases

This process describes the selection of sample algorithms which serve as the basis of the Use Case Verification Instances discussed in a later process.

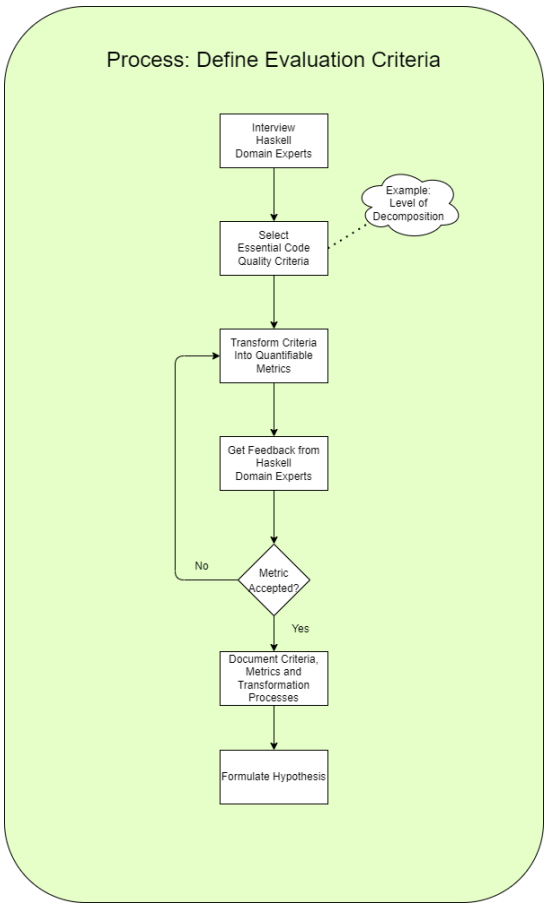


Figure 21: Process: Set Up Development Infrastructure

This process describes the evaluation and documentation of evaluation criteria, which describe the requirements that a Haskell developer has for a development tool.

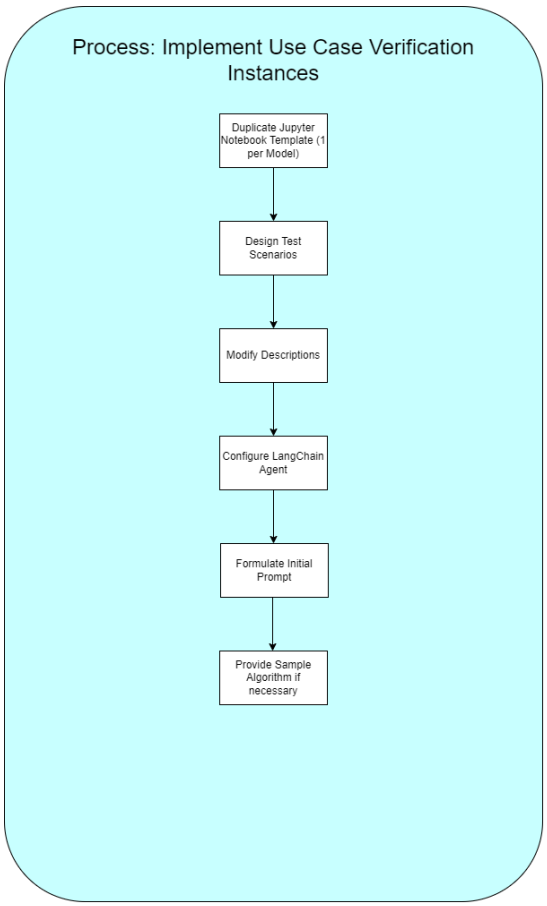


Figure 22: Process: Implement Use Case Verification Instances

This process describes the creation of a use case verification instance. A use case verification instance is a Jupyter notebook Section XI.B containing all necessary steps and information to configure and instantiate an LLM agent, access the test environment Section XI.C and evaluate the results.

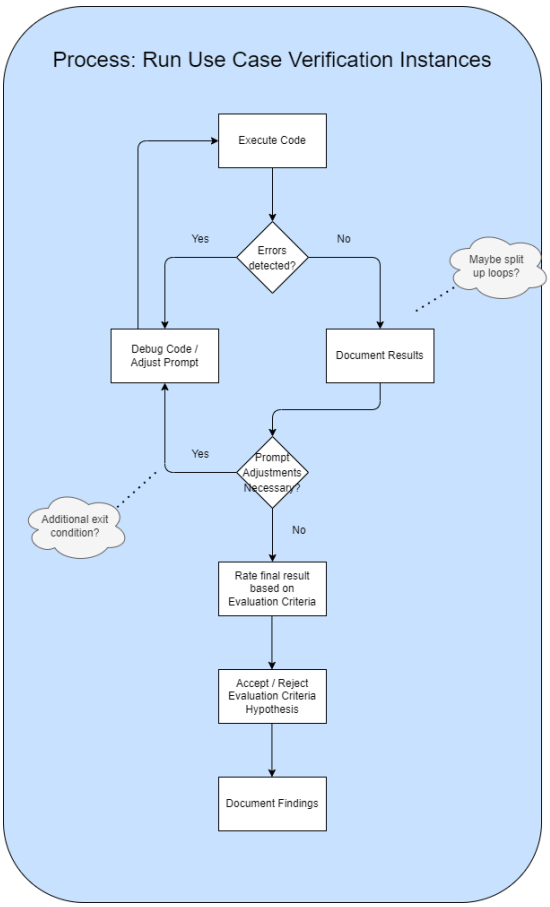


Figure 23: Process: Run Use Case Verification Instances

This process describes how the use case verification instances are supposed to be executed.

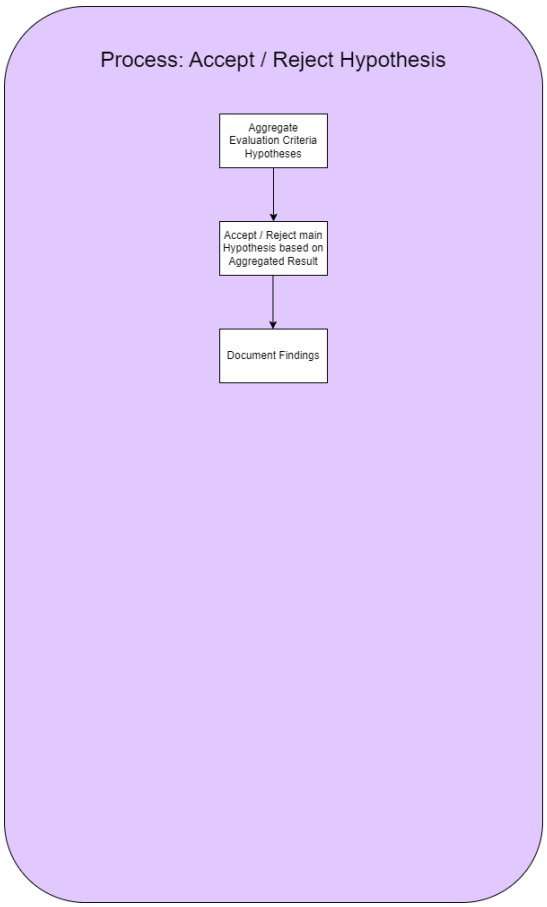


Figure 24: Process: Accept/Reject Hypothesis

This process describes the aggregation and acceptance or rejection of the defined hypotheses.

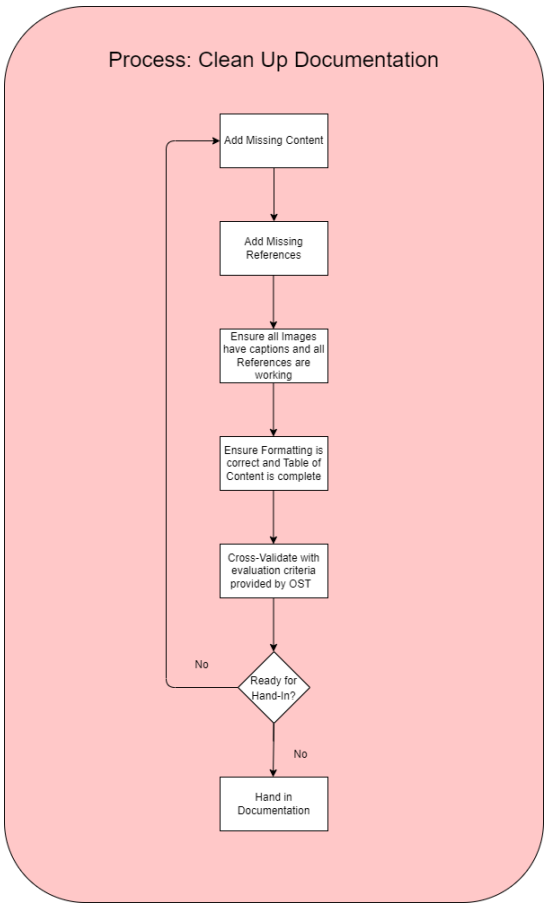


Figure 25: Process: Clean Up Documentation

This process describes how the documentation should receive a final review before it is handed in.

1. *Jira:* Jira was used to plan tasks in a two-week sprint cycle. As only two developers worked on the project, less coordination was needed in comparison to larger teams. Additional tasks could be added mid sprint if extra workload was available.

Sprints were left uncompleted because of previous experiences. Completing sprints made it harder to keep an overview over past work items. The amount of work items was still in a manageable magnitude to keep them on the same page.

Jira was used to keep track of ongoing and future work items and to track time. While Jira offers more functionality and could be used for proper agile development, the additional overhead needed was not deemed worth it.

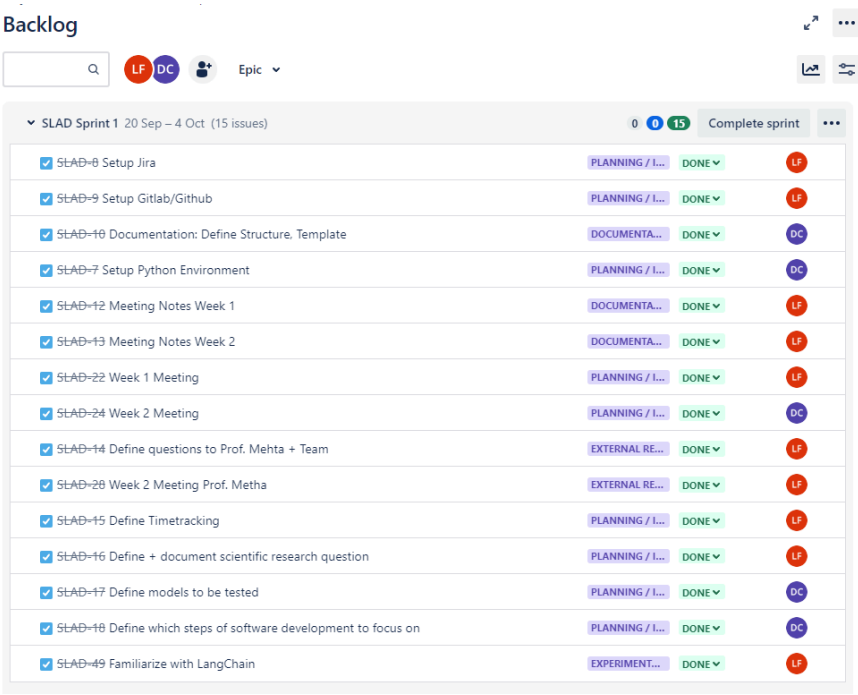


Figure 26: Example Jira sprint

D. Long-Term Plan

The long-term plan was set up with the following milestones:

- M1 Project is set up
- M2 Prompt - Run - Test Pipeline is set up
- M3 Evaluation Criteria are defined
- M4 Define use cases and usage scenarios
- M5 Use Cases are run, and results are documented
- M6 Prove or reject core hypothesis
- M7 Abstract handed in
- M8 Documentation handed in

Figure 27: Long term plan

E. Risk Management

The following project risks have been found and prevention measures have been taken.

#	**Risk**	**Likelihood of Occurence**	**Impact**	**Prevention measures**
1	Inaccurate Effort Estimates	4/5	3/5	1. Try to identify possible problem early (that there is more effort than expected) and communicate. 2. Reallocate resources, change project scope (shrink)
2	Activities not considered in project scope	3/5	2/5	1. Extensive requirements discovery at project start.
3	Communication Overhead	1/5	2/5	1. Streamline communication processes with the establishment of weekly sync meetings
4	Lack of experience	4/5	3/5	1. Utilize meetings with supervisor to ask questions or ask for resources for upskilling. 2. Assign tasks in accordance to skill set of project member.
5	Absence of team member	3/5	1/5	1. Takeover of work item if needed. 2. If prolonged: change project scope (shrink)
6	Conflicts between Team Members	2/5	1/5	1. Members are aware of their duties towards the project. 2. Reach a compromise if incident occurs. 3. In emergency: mediate via supervisor.
7	Problems with interfaces to external sytems	3/5	4/5	1. Initial evaluation serves as a PoC

F. Tools

The following tools were used for this project:

- ***Python 3*** was used as the programming langauge for our evaluation workflows, the test environment and the graphical outputs. It was used as all teammembers were experienced in using it and it is supported by Jupyter Notebooks.

- ***Haskell*** was used as the output target language for models in accordance to the intial evaluation.
- ***Jupyter Notebooks*** were used in order to present the workflow in a intuitive manner, making the workflow easier to understand, follow and debug. They allow easy sharing between the teammembers and allow simple duplication of similar notebooks. Matplotlib plots can be shown within the notebook, allowing for immediate visual feedback in the same notebook. Both teammembers had prior experience in their usage.
- ***LangChain***[16] is a python library that was used to handle the connection to the LLMs' APIs, delivering them prompt and parsing the output. It was used based on prior experience of one of the teammembers.
- ***Cria - Local llama OpenAI-compatible API***[17] was used as an API wrapper for a local deployment of Llama 2 and CodeLlama, imitating the OpenAI API, which allows LangChain to communicate with those models in an identical manner.
- ***Docker*** was used to build the Cria service as recommended by its documentation.
- ***GitHub*** was used as a collaboration platform, allowing for version controlled cooperation on the same repository and development in separate branches for each feature, allowing independant work. It's CI/CD feature was used to continually build the typst documentation on GitHub, as some IDEs like Visual Studio Code had issues linking typst references to other documents, while building the PDF from .typ still worked properly through commands. Both teammembers had prior experience in its usage.
- ***typst***[18] while still being in beta, was used as the tool to write and build the documentation. Another alternative would have been Latex. As other students were also using typst for their SA and after an initial evaluation, it was decided to try it out.
- ***Excel*** was used to dynamically adjust the scores of the evaluation criteria and to present them in a pleasing way. Additionally the project's long term plan was created using it.
- ***Draw.IO***[19] was used to create flow- and sytemdiagrams.
- ***Jira***[20] was used to for workload planning and timetracking.

G. Time Recording

Time recording was done with Jira tasks. In the end we recorded a total of 454 hours on the project. 220 hours were recorded by Dominik Castelberg, 234 hours were recorded by Linus Flury.

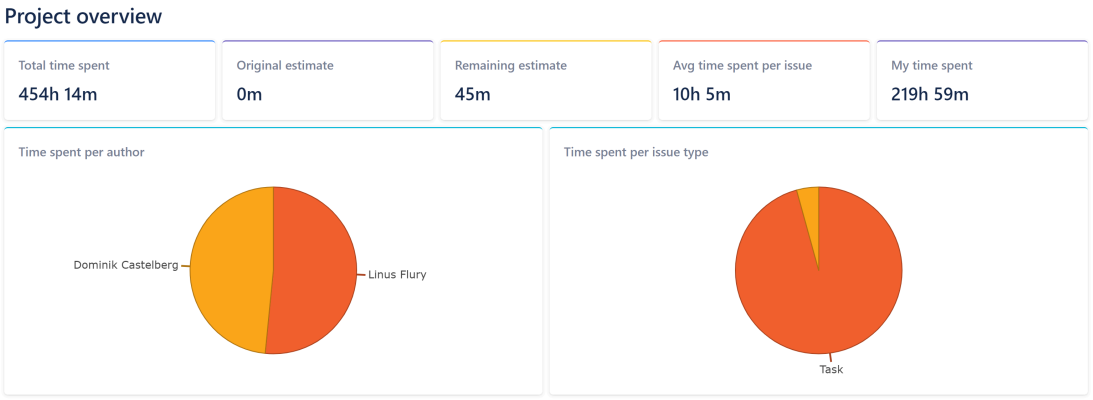


Figure 28: Screenshot: Jira Timesheet

A detailed breakdown grouped by Tasks can be found in the separate document `time_recorded_by_task.attachment.pdf` in the subfolder `attachments`.

[16] https://python.langchain.com
[17] https://github.com/AmineDiro/cria
[18] https://typst.app/
[19] https://www.drawio.com/
[20] https://www.atlassian.com/software/jira

XIII. Attachments

A. Conda Config

conda_env.yaml

```
name: SA_LLM_DevTools
channels:
  - conda-forge
  - https://conda.anaconda.org/conda-forge/
  - defaults
dependencies:
  - aiohttp=3.8.5=py311ha68e1ae_0
  - aiosignal=1.3.1=pyhd8ed1ab_0
  - anyio=4.0.0=pyhd8ed1ab_0
  - appdirs=1.4.4=pyh9f0ad1d_0
  - argon2-cffi=23.1.0=pyhd8ed1ab_0
  - argon2-cffi-bindings=21.2.0=py311ha68e1ae_4
  - arrow=1.2.3=pyhd8ed1ab_0
  - asttokens=2.4.0=pyhd8ed1ab_0
  - async-lru=2.0.4=pyhd8ed1ab_0
  - async-timeout=4.0.3=pyhd8ed1ab_0
  - attrs=23.1.0=pyh71513ae_1
  - babel=2.12.1=pyhd8ed1ab_1
  - backcall=0.2.0=pyh9f0ad1d_0
  - backports=1.0=pyhd8ed1ab_3
  - backports.functools_lru_cache=1.6.5=pyhd8ed1ab_0
  - beautifulsoup4=4.12.2=pyha770c72_0
  - bleach=6.0.0=pyhd8ed1ab_0
  - brotli=1.1.0=hcfcfb64_0
  - brotli-bin=1.1.0=hcfcfb64_0
  - brotli-python=1.1.0=py311h12c1d0e_0
  - bzip2=1.0.8=h8ffe710_4
  - ca-certificates=2023.7.22=h56e8100_0
  - cached-property=1.5.2=hd8ed1ab_1
  - cached_property=1.5.2=pyha770c72_1
  - cachetools=5.3.1=pyhd8ed1ab_0
  - certifi=2023.7.22=pyhd8ed1ab_0
  - cffi=1.15.1=py311ha68e1ae_5
  - charset-normalizer=3.2.0=pyhd8ed1ab_0
  - click=8.1.7=win_pyh7428d3b_0
  - colorama=0.4.6=pyhd8ed1ab_0
  - comm=0.1.4=pyhd8ed1ab_0
  - contourpy=1.1.1=py311h005e61a_1
  - cryptography=41.0.4=py311h28e9c30_0
  - cycler=0.11.0=pyhd8ed1ab_0
  - dataclasses-json=0.5.7=pyhd8ed1ab_0
  - debugpy=1.8.0=py311h12c1d0e_1
  - decorator=5.1.1=pyhd8ed1ab_0
  - defusedxml=0.7.1=pyhd8ed1ab_0
  - docker-pycreds=0.4.0=py_0
  - entrypoints=0.4=pyhd8ed1ab_0
  - et_xmlfile=1.1.0=pyhd8ed1ab_0
  - exceptiongroup=1.1.3=pyhd8ed1ab_0
  - executing=1.2.0=pyhd8ed1ab_0
  - fonttools=4.42.1=py311ha68e1ae_0
  - fqdn=1.5.1=pyhd8ed1ab_0
  - freetype=2.12.1=hdaf720e_2
  - frozenlist=1.4.0=py311ha68e1ae_1
  - gettext=0.21.1=h5728263_0
```

- gitdb=4.0.10=pyhd8ed1ab_0
- gitpython=3.1.37=pyhd8ed1ab_0
- glib=2.78.0=h12be248_0
- glib-tools=2.78.0=h12be248_0
- google-api-core=2.12.0=pyhd8ed1ab_0
- google-auth=2.23.1=pyhca7485f_0
- googleapis-common-protos=1.60.0=pyhd8ed1ab_0
- greenlet=2.0.2=py311h12c1d0e_1
- icu=58.2=ha925a31_3
- idna=3.4=pyhd8ed1ab_0
- importlib-metadata=6.8.0=pyha770c72_0
- importlib_metadata=6.8.0=hd8ed1ab_0
- importlib_resources=6.1.0=pyhd8ed1ab_0
- intel-openmp=2023.2.0=h57928b3_49503
- ipykernel=6.25.2=pyh60829e3_0
- ipython=8.15.0=pyh5737063_0
- ipython_genutils=0.2.0=py_1
- ipywidgets=8.1.1=pyhd8ed1ab_0
- isoduration=20.11.0=pyhd8ed1ab_0
- jedi=0.19.0=pyhd8ed1ab_0
- jinja2=3.1.2=pyhd8ed1ab_1
- joblib=1.3.2=pyhd8ed1ab_0
- jpeg=9e=hcfcfb64_3
- json5=0.9.14=pyhd8ed1ab_0
- jsonpointer=2.4=py311h1ea47a8_3
- jsonschema=4.19.1=pyhd8ed1ab_0
- jsonschema-specifications=2023.7.1=pyhd8ed1ab_0
- jsonschema-with-format-nongpl=4.19.1=pyhd8ed1ab_0
- jupyter=1.0.0=py311h1ea47a8_8
- jupyter-lsp=2.2.0=pyhd8ed1ab_0
- jupyter_client=8.3.1=pyhd8ed1ab_0
- jupyter_console=6.6.3=pyhd8ed1ab_0
- jupyter_core=5.3.1=py311h1ea47a8_1
- jupyter_events=0.7.0=pyhd8ed1ab_2
- jupyter_server=2.7.3=pyhd8ed1ab_1
- jupyter_server_terminals=0.4.4=pyhd8ed1ab_1
- jupyterlab=4.0.6=pyhd8ed1ab_0
- jupyterlab_pygments=0.2.2=pyhd8ed1ab_0
- jupyterlab_server=2.25.0=pyhd8ed1ab_0
- jupyterlab_widgets=3.0.9=pyhd8ed1ab_0
- kiwisolver=1.4.5=py311h005e61a_1
- krb5=1.20.1=heb0366b_0
- langchain=0.0.277=pyhd8ed1ab_0
- langsmith=0.0.41=pyhd8ed1ab_0
- lerc=3.0=hd77b12b_0
- libabseil=20230802.1=cxx17_h63175ca_0
- libblas=3.9.0=18_win64_mkl
- libbrotlicommon=1.1.0=hcfcfb64_0
- libbrotlidec=1.1.0=hcfcfb64_0
- libbrotlienc=1.1.0=hcfcfb64_0
- libcblas=3.9.0=18_win64_mkl
- libclang=14.0.6=default_hb5a9fac_1
- libclang13=14.0.6=default_h8e68704_1
- libdeflate=1.17=h2bbff1b_0
- libexpat=2.5.0=h63175ca_1
- libffi=3.4.2=h8ffe710_5
- libglib=2.78.0=he8f3873_0
- libiconv=1.17=h8ffe710_0
- liblapack=3.9.0=18_win64_mkl
- libpng=1.6.39=h19919ed_0

- libpq=12.15=h906ac69_1
- libprotobuf=4.24.3=hb8276f3_0
- libsodium=1.0.18=h8d14728_1
- libsqlite=3.43.0=hcfcfb64_0
- libtiff=4.5.1=hd77b12b_0
- libwebp=1.3.2=hcfcfb64_0
- libwebp-base=1.3.2=hcfcfb64_0
- libxml2=2.10.4=hc3477c8_0
- libxslt=1.1.37=h0192164_0
- libzlib=1.2.13=hcfcfb64_5
- m2w64-gcc-libgfortran=5.3.0=6
- m2w64-gcc-libs=5.3.0=7
- m2w64-gcc-libs-core=5.3.0=7
- m2w64-gmp=6.1.0=2
- m2w64-libwinpthread-git=5.0.0.4634.697f757=2
- markupsafe=2.1.3=py311ha68e1ae_1
- marshmallow=3.20.1=pyhd8ed1ab_0
- marshmallow-enum=1.5.1=pyh9f0ad1d_3
- matplotlib=3.8.0=py311h1ea47a8_1
- matplotlib-base=3.8.0=py311h6e989c2_1
- matplotlib-inline=0.1.6=pyhd8ed1ab_0
- mistune=3.0.1=pyhd8ed1ab_0
- mkl=2022.1.0=h6a75c08_874
- msys2-conda-epoch=20160418=1
- multidict=6.0.4=py311ha68e1ae_0
- munkres=1.1.4=pyh9f0ad1d_0
- mypy_extensions=1.0.0=pyha770c72_0
- nbclient=0.8.0=pyhd8ed1ab_0
- nbconvert=7.8.0=pyhd8ed1ab_0
- nbconvert-core=7.8.0=pyhd8ed1ab_0
- nbconvert-pandoc=7.8.0=pyhd8ed1ab_0
- nbformat=5.9.2=pyhd8ed1ab_0
- nest-asyncio=1.5.6=pyhd8ed1ab_0
- notebook=7.0.4=pyhd8ed1ab_0
- notebook-shim=0.2.3=pyhd8ed1ab_0
- numexpr=2.8.7=mkl_py311h9a3bfb6_0
- numpy=1.26.0=py311h0b4df5a_0
- openai=0.28.1=pyhd8ed1ab_0
- openapi-schema-pydantic=1.2.4=pyhd8ed1ab_0
- openpyxl=3.1.2=py311ha68e1ae_1
- openssl=3.1.3=hcfcfb64_0
- overrides=7.4.0=pyhd8ed1ab_0
- packaging=23.1=pyhd8ed1ab_0
- pandas=2.1.1=py311hf63dbb6_0
- pandas-stubs=2.0.3.230814=pyhd8ed1ab_0
- pandoc=3.1.3=h57928b3_0
- pandocfilters=1.5.0=pyhd8ed1ab_0
- parso=0.8.3=pyhd8ed1ab_0
- pathtools=0.1.2=py_1
- pcre2=10.40=h17e33f8_0
- pickleshare=0.7.5=py_1003
- pillow=9.4.0=py311hd77b12b_1
- pip=23.2.1=pyhd8ed1ab_0
- pkgutil-resolve-name=1.3.10=pyhd8ed1ab_1
- platformdirs=3.10.0=pyhd8ed1ab_0
- plotly=5.17.0=pyhd8ed1ab_0
- ply=3.11=py_1
- prometheus_client=0.17.1=pyhd8ed1ab_0
- prompt-toolkit=3.0.39=pyha770c72_0
- prompt_toolkit=3.0.39=hd8ed1ab_0

- protobuf=4.24.3=py311h72e314b_0
- psutil=5.9.5=py311ha68e1ae_1
- pure_eval=0.2.2=pyhd8ed1ab_0
- pyasn1=0.5.0=pyhd8ed1ab_0
- pyasn1-modules=0.3.0=pyhd8ed1ab_0
- pycparser=2.21=pyhd8ed1ab_0
- pydantic=1.10.12=py311ha68e1ae_1
- pygments=2.16.1=pyhd8ed1ab_0
- pyopenssl=23.2.0=pyhd8ed1ab_1
- pyparsing=3.1.1=pyhd8ed1ab_0
- pyqt=5.15.7=py311hd77b12b_0
- pyqt5-sip=12.11.0=py311hd77b12b_0
- pysocks=1.7.1=pyh0701188_6
- python=3.11.5=h2628c8c_0_cpython
- python-dateutil=2.8.2=pyhd8ed1ab_0
- python-fastjsonschema=2.18.0=pyhd8ed1ab_0
- python-json-logger=2.0.7=pyhd8ed1ab_0
- python-tzdata=2023.3=pyhd8ed1ab_0
- python_abi=3.11=4_cp311
- pytz=2023.3.post1=pyhd8ed1ab_0
- pyu2f=0.1.5=pyhd8ed1ab_0
- pywin32=306=py311h12c1d0e_1
- pywinpty=2.0.11=py311h12c1d0e_1
- pyyaml=6.0.1=py311ha68e1ae_1
- pyzmq=25.1.1=py311h7b3f143_1
- qt-main=5.15.2=h879a1e9_9
- qt-webengine=5.15.9=h5bd16bc_7
- qtconsole=5.4.4=pyhd8ed1ab_0
- qtconsole-base=5.4.4=pyha770c72_0
- qtpy=2.4.0=pyhd8ed1ab_0
- qtwebkit=5.212=h2bbfb41_5
- referencing=0.30.2=pyhd8ed1ab_0
- requests=2.31.0=pyhd8ed1ab_0
- rfc3339-validator=0.1.4=pyhd8ed1ab_0
- rfc3986-validator=0.1.1=pyh9f0ad1d_0
- rpds-py=0.10.3=py311hc37eb10_0
- rsa=4.9=pyhd8ed1ab_0
- scikit-learn=1.3.1=py311h142b183_0
- scipy=1.11.2=py311h37ff6ca_1
- send2trash=1.8.2=pyh08f2357_0
- sentry-sdk=1.31.0=pyhd8ed1ab_0
- setproctitle=1.3.2=py311ha68e1ae_2
- setuptools=68.2.2=pyhd8ed1ab_0
- sip=6.6.2=py311hd77b12b_0
- six=1.16.0=pyh6c4a22f_0
- smmap=3.0.5=pyh44b312d_0
- sniffio=1.3.0=pyhd8ed1ab_0
- soupsieve=2.5=pyhd8ed1ab_1
- sqlalchemy=2.0.21=py311ha68e1ae_0
- sqlite=3.43.0=hcfcfb64_0
- stack_data=0.6.2=pyhd8ed1ab_0
- stringcase=1.2.0=py_0
- tbb=2021.8.0=h59b6b97_0
- tenacity=8.2.3=pyhd8ed1ab_0
- terminado=0.17.1=py311haa95532_0
- threadpoolctl=3.2.0=pyha21a80b_0
- tinycss2=1.2.1=pyhd8ed1ab_0
- tk=8.6.13=hcfcfb64_0
- toml=0.10.2=pyhd8ed1ab_0
- tomli=2.0.1=pyhd8ed1ab_0

- tornado=6.3.3=py311ha68e1ae_1
- tqdm=4.66.1=pyhd8ed1ab_0
- traitlets=5.10.1=pyhd8ed1ab_0
- types-pytz=2023.3.1.1=pyhd8ed1ab_0
- typing-extensions=4.8.0=hd8ed1ab_0
- typing_extensions=4.8.0=pyha770c72_0
- typing_inspect=0.9.0=pyhd8ed1ab_0
- typing_utils=0.1.0=pyhd8ed1ab_0
- tzdata=2023c=h71feb2d_0
- ucrt=10.0.22621.0=h57928b3_0
- uri-template=1.3.0=pyhd8ed1ab_0
- urllib3=2.0.5=pyhd8ed1ab_0
- vc=14.3=h64f974e_17
- vc14_runtime=14.36.32532=hdcecf7f_17
- vs2015_runtime=14.36.32532=h05e6639_17
- wandb=0.15.11=pyhd8ed1ab_0
- wcwidth=0.2.6=pyhd8ed1ab_0
- webcolors=1.13=pyhd8ed1ab_0
- webencodings=0.5.1=pyhd8ed1ab_2
- websocket-client=1.6.3=pyhd8ed1ab_0
- wheel=0.41.2=pyhd8ed1ab_0
- widgetsnbextension=4.0.9=pyhd8ed1ab_0
- win_inet_pton=1.1.0=pyhd8ed1ab_6
- winpty=0.4.3=4
- xz=5.4.2=h8cc25b3_0
- yaml=0.2.5=h8ffe710_2
- yarl=1.9.2=py311ha68e1ae_0
- zeromq=4.3.4=h0e60522_1
- zipp=3.17.0=pyhd8ed1ab_0
- zlib=1.2.13=hcfcfb64_5
- zstd=1.5.5=h12be248_0

B. Azure Resources

template.json

```
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "accounts_SA_LLM_Azure_OpenAI_name": {
            "defaultValue": "SA-LLM-Azure-OpenAI",
            "type": "String"
        }
    },
    "variables": {},
    "resources": [
        {
            "type": "Microsoft.CognitiveServices/accounts",
            "apiVersion": "2023-10-01-preview",
            "name": "[parameters('accounts_SA_LLM_Azure_OpenAI_name')]",
            "location": "switzerlandnorth",
            "sku": {
                "name": "S0"
            },
            "kind": "OpenAI",
            "properties": {
                "customSubDomainName": "sa-llm-azure-openai",
                "networkAcls": {
                    "defaultAction": "Allow",
                    "virtualNetworkRules": [],
                    "ipRules": []
                },
                "publicNetworkAccess": "Enabled"
            }
        },
        {
            "type": "Microsoft.CognitiveServices/accounts/deployments",
            "apiVersion": "2023-10-01-preview",
            "name": "[concat(parameters('accounts_SA_LLM_Azure_OpenAI_name'), '/SLAD-GPT-35-Turbo')]",
            "dependsOn": [
                "[resourceId('Microsoft.CognitiveServices/accounts',
parameters('accounts_SA_LLM_Azure_OpenAI_name'))]"
            ],
            "sku": {
                "name": "Standard",
                "capacity": 120
            },
            "properties": {
                "model": {
                    "format": "OpenAI",
                    "name": "gpt-35-turbo",
                    "version": "0613"
                },
                "versionUpgradeOption": "OnceCurrentVersionExpired",
                "currentCapacity": 120,
                "raiPolicyName": "Microsoft.Default"
            }
        },
        {
            "type": "Microsoft.CognitiveServices/accounts/deployments",
            "apiVersion": "2023-10-01-preview",
```

```
            "name": "[concat(parameters('accounts_SA_LLM_Azure_OpenAI_name'), '/SLAD-GPT-4')]",
            "dependsOn": [
                "[resourceId('Microsoft.CognitiveServices/accounts',
parameters('accounts_SA_LLM_Azure_OpenAI_name'))]"
            ],
            "sku": {
                "name": "Standard",
                "capacity": 10
            },
            "properties": {
                "model": {
                    "format": "OpenAI",
                    "name": "gpt-4",
                    "version": "0613"
                },
                "versionUpgradeOption": "OnceCurrentVersionExpired",
                "currentCapacity": 10,
                "raiPolicyName": "Microsoft.Default"
            }
        },
        {
            "type": "Microsoft.CognitiveServices/accounts/raiPolicies",
            "apiVersion": "2023-10-01-preview",
            "name": "[concat(parameters('accounts_SA_LLM_Azure_OpenAI_name'), '/Microsoft.Default')]",
            "dependsOn": [
                "[resourceId('Microsoft.CognitiveServices/accounts',
parameters('accounts_SA_LLM_Azure_OpenAI_name'))]"
            ],
            "properties": {
                "mode": "Blocking",
                "contentFilters": [
                    {
                        "allowedContentLevel": "Medium",
                        "blocking": true,
                        "enabled": true,
                        "source": "Prompt"
                    },
                    {
                        "allowedContentLevel": "Medium",
                        "blocking": true,
                        "enabled": true,
                        "source": "Completion"
                    },
                    {
                        "allowedContentLevel": "Medium",
                        "blocking": true,
                        "enabled": true,
                        "source": "Prompt"
                    },
                    {
                        "allowedContentLevel": "Medium",
                        "blocking": true,
                        "enabled": true,
                        "source": "Completion"
                    },
                    {
                        "allowedContentLevel": "Medium",
                        "blocking": true,
                        "enabled": true,
                        "source": "Prompt"
```

```
                    },
                    {
                        "allowedContentLevel": "Medium",
                        "blocking": true,
                        "enabled": true,
                        "source": "Completion"
                    },
                    {
                        "allowedContentLevel": "Medium",
                        "blocking": true,
                        "enabled": true,
                        "source": "Prompt"
                    },
                    {
                        "allowedContentLevel": "Medium",
                        "blocking": true,
                        "enabled": true,
                        "source": "Completion"
                    }
                ]
            }
        }
    ]
}
```

parameters.json

```
{
    "$schema": "https://schema.management.azure.com/schemas/2015-01-01/deploymentParameters.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "accounts_SA_LLM_Azure_OpenAI_name": {
            "value": null
        }
    }
}
```

C. Recommended VS Code Extensions

extensions.json

```
{
  // See https://go.microsoft.com/fwlink/?LinkId=827846 to learn about workspace recommendations.
  // Extension identifier format: ${publisher}.${name}. Example: vscode.csharp

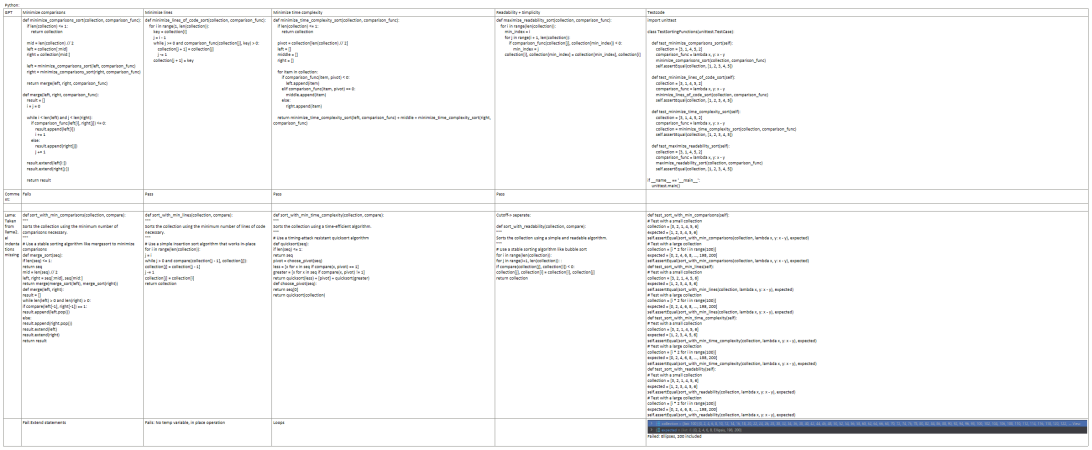
  // List of extensions which should be recommended for users of this workspace.
  "recommendations": [
    "yzhang.markdown-all-in-one",
    "bierner.markdown-footnotes",
    "ms-toolsai.jupyter",
    "ms-toolsai.vscode-jupyter-cell-tags",
    "ms-toolsai.jupyter-keymap",
    "ms-toolsai.vscode-jupyter-slideshow",
    "ms-python.python",
    "tomoki1207.pdf",
    "hediet.vscode-drawio",
    "haskell.haskell"
  ],
  // List of extensions recommended by VS Code that should not be recommended for users of this workspace.
  "unwantedRecommendations": [
    "davidanson.vscode-markdownlint" // Part of markdown-all-in-one
  ]
}
```

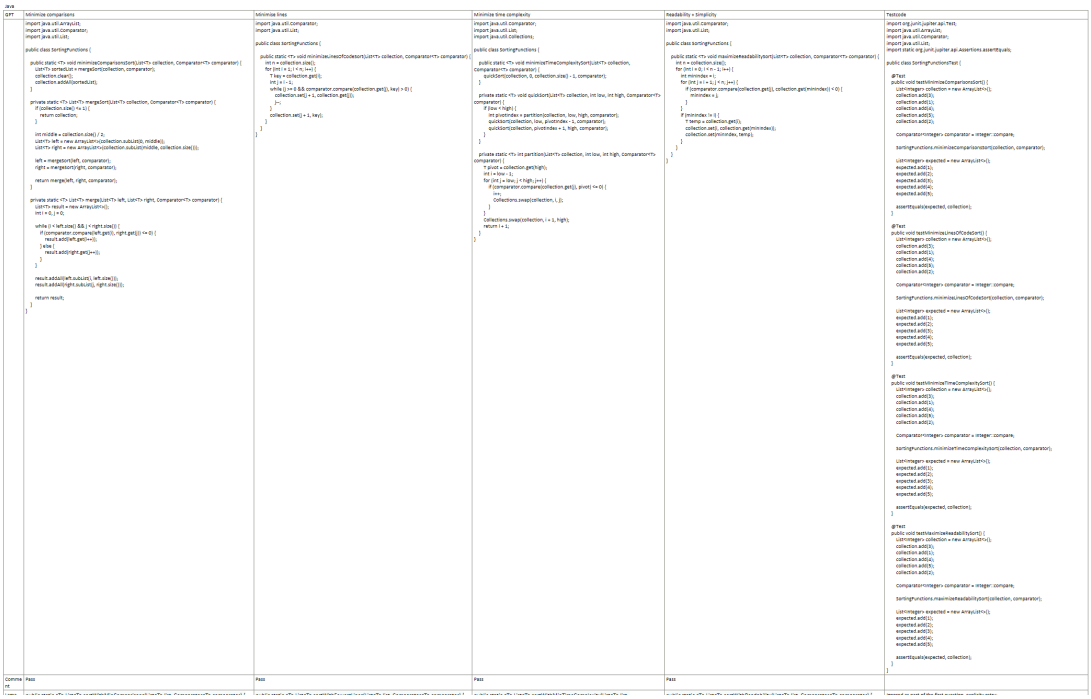
D. *Initial evaluation*

Prompt:

```
Write x code. It should contain functions which take a generic collection. The output of each is the same collection in a sorted order.
The comparison algorithm is provided to the functions aswell.
Using the provided "sort" function of the collection is not allowed.
Write functions each of the following objectives:
1) Minimize the number of comparisons
2) Minimize the lines of code
3) Minimize time complexity
4) Maximize code readbility and simplicity

Then provide a testclass to test those functions
```





Lama (70b, not code specific)	public static <T> List<T> sortWithMinComparisons(List<T> list, Comparator<T> comparator) { int n = list.size(); if (n <= 1) return list; // already sorted or empty T pivot = list.get(0); int i = 1; for (int j = 1; j < n; j++) { if (comparator.compare(list.get(j), pivot) < 0) { i++; swap(list, i - 1, j); } } return list; }	public static <T> List<T> sortWithFewestLines(List<T> list, Comparator<T> comparator) { return list.stream() .sorted(comparator) .collect(Collectors.toList()); }	public static <T> List<T> sortWithMinTimeComplexity(List<T> list, Comparator<T> comparator) { int n = list.size();if (n <= 1) return list; // already sorted or empty T[] aux = new T[n]; for (int i = 0; i < n; i++) { aux[i] = list.get(i); } Arrays.sort(aux, comparator); list.clear(); for (int i = 0; i < n; i++) { list.add(aux[i]); } return list; }	public static <T> List<T> sortWithReadability(List<T> list, Comparator<T> comparator) { list.sort(comparator); return list; }	Ignored as part of the first question, explicity retry: import org.junit.Test; import java.util.ArrayList; import java.util.Comparator; import java.util.List; public class SortingFunctionsTest { @Test public void testSortWithMinComparisons() { // Test with a small list List<String> list = new ArrayList<>(); list.add("c"); list.add("b"); list.add("a"); list.add("d"); list.add("e"); list.add("f"); Comparator<String> comparator = Comparator.naturalOrder(); List<String> sortedList = sortWithMinComparisons(list, comparator); assertEquals(list, sortedList); // Test with a large list list = new ArrayList<>(); for (int i = 0; i < 1000; i++) { list.add("elem" + i); } comparator = Comparator.naturalOrder(); sortedList = sortWithMinComparisons(list, comparator); assertEquals(list, sortedList); } @Test public void testSortWithFewestLines() { // Test with a small list List<String> list = new ArrayList<>(); list.add("c"); list.add("b"); list.add("a"); list.add("d"); list.add("e"); list.add("f"); Comparator<String> comparator = Comparator.naturalOrder(); List<String> sortedList = sortWithFewestLines(list, comparator); assertEquals(list, sortedList); // Test with a large list list = new ArrayList<>(); for (int i = 0; i < 1000; i++) { list.add("elem" + i); } comparator = Comparator.naturalOrder(); sortedList = sortWithFewestLines(list, comparator); assertEquals(list, sortedList); } @Test public void testSortWithMinTimeComplexity() { // Test with a small list List<String> list = new ArrayList<>(); list.add("c"); list.add("b"); list.add("a"); list.add("d"); list.add("e"); list.add("f"); Comparator<String> comparator = Comparator.naturalOrder(); List<String> sortedList = sortWithMinTimeComplexity(list, comparator); assertEquals(list, sortedList); // Test with a large list list = new ArrayList<>(); for (int i = 0; i < 1000; i++) { list.add("elem" + i); } comparator = Comparator.naturalOrder(); sortedList = sortWithMinTimeComplexity(list, comparator); assertEquals(list, sortedList); } @Test public void testSortWithReadability() { // Test with a small list List<String> list = new ArrayList<>(); list.add("c"); list.add("b"); ------ cutoff Retry only 4: import org.junit.Before; import org.junit.Test; import java.util.ArrayList; import java.util.List; import java.util.Comparator; public class SortWithReadabilityTest { private List<String> list; private Comparator<String> comparator; @Before public void setup() { list = new ArrayList<>(); list.add("apple"); list.add("banana"); list.add("cherry"); comparator = (o1, o2) -> o1.compareTo(o2); } @Test public void testSortWithReadability() { List<String> sortedList = SortWithReadability.sortWithReadability(list, comparator); assertEquals(list, sortedList); assertTrue(sortedList.isEmpty()); } @Test public void testSortWithReadabilityReverse() { List<String> sortedList = SortWithReadability.sortWithReadability(list, comparator.reversed()); assertEquals(list, sortedList); assertTrue(sortedList.isEmpty()); } @Test public void testSortWithReadabilityCustomComparator() { Comparator<String> customComparator = (o1, o2) -> o1.length() - o2.length(); List<String> sortedList = SortWithReadability.sortWithReadability(list, customComparator); assertEquals(list, sortedList); assertTrue(sortedList.isEmpty()); } }
			T[]aux=newT[n]; not allowed! Type parameter 'T' cannot be instantiated directly		only withMinComparison passes

C#

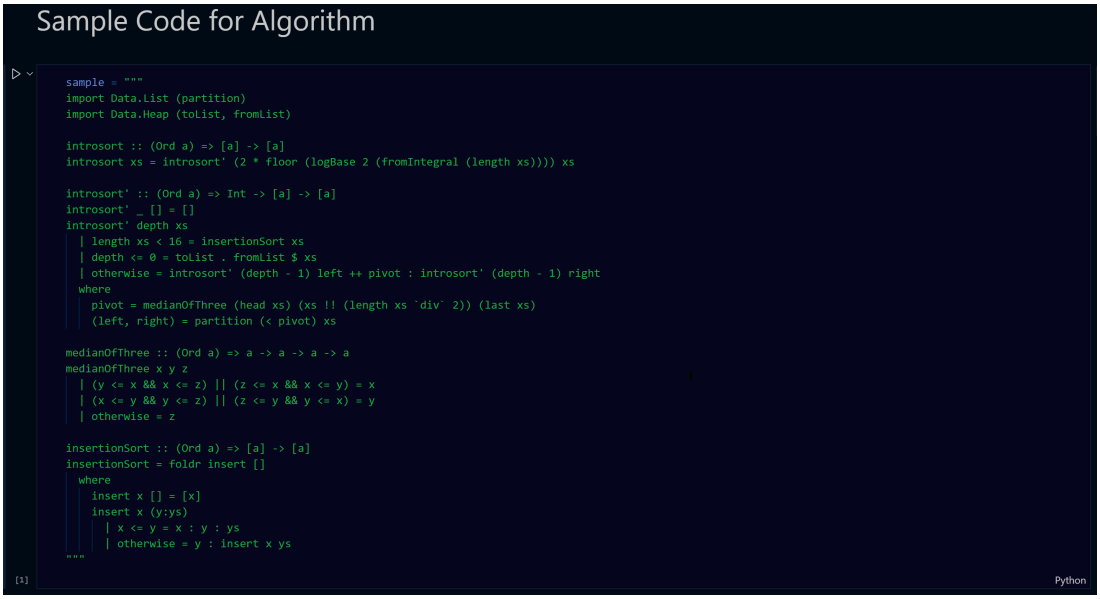
GPT	Minimize comparisons	Minimise lines	Minimize time complexity	Readability + Simplicity	Testcode
	using System; using System.Collections.Generic; public class SortingFunctions { public static void MinimizeComparisonsSort<T>(List<T> collection, Comparison<T> comparison) { List<T> sortedList = MergeSort(collection, comparison); collection.Clear(); collection.AddRange(sortedList); } private static List<T> MergeSort<T>(List<T> collection, Comparison<T> comparison) { if (collection.Count <= 1) { return collection; } int middle = collection.Count / 2; List<T> left = collection.GetRange(0, middle); List<T> right = collection.GetRange(middle, collection.Count - middle); left = MergeSort(left, comparison); right = MergeSort(right, comparison); return Merge(left, right, comparison); } private static List<T> Merge<T>(List<T> left, List<T> right, Comparison<T> comparison) { List<T> result = new List<T>(); int i = 0, j = 0; while (i < left.Count && j < right.Count) { if (comparison(left[i], right[j]) <= 0) { result.Add(left[i++]); } else { result.Add(right[j++]); } } result.AddRange(left.GetRange(i, left.Count - i)); result.AddRange(right.GetRange(j, right.Count - j)); return result; } }	using System; using System.Collections.Generic; public class SortingFunctions { public static void MinimizeLinesOfCodeSort<T>(List<T> collection, Comparison<T> comparison) { int n = collection.Count; for (int i = 1; i < n; i++) { T key = collection[i]; int j = i - 1; while (j >= 0 && comparison(collection[j], key) > 0) { collection[j + 1] = collection[j]; j--; } collection[j + 1] = key; } } }	using System; using System.Collections.Generic; public class SortingFunctions { public static void MinimizeTimeComplexitySort<T>(List<T> collection, Comparison<T> comparison) { QuickSort(collection, 0, collection.Count - 1, comparison); } private static void QuickSort<T>(List<T> collection, int low, int high, Comparison<T> comparison) { if (low < high) { int pivotIndex = Partition(collection, low, high, comparison); QuickSort(collection, low, pivotIndex - 1, comparison); QuickSort(collection, pivotIndex + 1, high, comparison); } } private static int Partition<T>(List<T> collection, int low, int high, Comparison<T> comparison) { T pivot = collection[high]; int i = low - 1; for (int j = low; j < high; j++) { if (comparison(collection[j], pivot) < 0) { i++; Swap(collection, i, j); } } Swap(collection, i + 1, high); return i + 1; } private static void Swap<T>(List<T> collection, int i, int j) { T temp = collection[i]; collection[i] = collection[j]; collection[j] = temp; } }	using System; using System.Collections.Generic; public class SortingFunctions { public static void MaximizeReadabilitySort<T>(List<T> collection, Comparison<T> comparison) { int n = collection.Count; for (int i = 0; i < n - 1; i++) { int minIndex = i; for (int j = i + 1; j < n; j++) { if (comparison(collection[j], collection[minIndex]) < 0) { minIndex = j; } } if (minIndex != i) { Swap(collection, i, minIndex); } } } private static void Swap<T>(List<T> collection, int i, int j) { T temp = collection[i]; collection[i] = collection[j]; collection[j] = temp; } }	using System; using System.Collections.Generic; using Xunit; public class SortingFunctionsTest { [Fact] public void TestMinimizeComparisonsSort() { List<int> collection = new List<int> { 3, 1, 4, 5, 2 }; Comparison<int> comparison = (x, y) => x - y; SortingFunctions.MinimizeComparisonsSort(collection, comparison); List<int> expected = new List<int> { 1, 2, 3, 4, 5 }; Assert.Equal(expected, collection); } [Fact] public void TestMinimizeLinesOfCodeSort() { List<int> collection = new List<int> { 3, 1, 4, 5, 2 }; Comparison<int> comparison = (x, y) => x - y; SortingFunctions.MinimizeLinesOfCodeSort(collection, comparison); List<int> expected = new List<int> { 1, 2, 3, 4, 5 }; Assert.Equal(expected, collection); } [Fact] public void TestMinimizeTimeComplexitySort() { List<int> collection = new List<int> { 3, 1, 4, 5, 2 }; Comparison<int> comparison = (x, y) => x - y; SortingFunctions.MinimizeTimeComplexitySort(collection, comparison); List<int> expected = new List<int> { 1, 2, 3, 4, 5 }; Assert.Equal(expected, collection); } [Fact] public void TestMaximizeReadabilitySort() { List<int> collection = new List<int> { 3, 1, 4, 5, 2 }; Comparison<int> comparison = (x, y) => x - y; SortingFunctions.MaximizeReadabilitySort(collection, comparison); List<int> expected = new List<int> { 1, 2, 3, 4, 5 }; Assert.Equal(expected, collection); } }
	Pass	Pass	Pass	Pass	

Lama	public static void SortMinComp(List<int> collection, Comparison<int> compare) { int len = collection.Count; int pivot = 0; while (pivot < len - 1) { int smallest = pivot; for (int i = pivot + 1; i < len; i++) if (compare(collection[i], collection[smallest]) < 0) smallest = i; Swap(collection, pivot, smallest); pivot++; } }	public static void SortFast(List<int> collection, Comparison<int> compare) { int[] aux = new int[collection.Count]; Array.Copy(collection.ToArray(), aux, collection.Count); Array.Sort(aux, compare); collection.Clear(); foreach (int item in aux) collection.Add(item); }	public static void SortSlowButSteady(List<int> collection, Comparison<int> compare) { for (int i = 0; i < collection.Count - 1; i++) { int minindex = i; for (int j = i + 1; j < collection.Count; j++) if (compare(collection[j], collection[minindex]) < 0) minindex = j; Swap(collection, i, minindex); } }	public static void SortEasyToRead<T>(List<T> collection, Comparison<T> compare) { // Create a temporary list to hold the sorted items var sortedList = new List<T>(); // Iterate through the original collection and add items to the sorted list foreach (T item in collection) { // Add the item to the sorted list, using the provided comparison delegate // to ensure the correct ordering sortedList.Add(item); } // Clear the original collection and add the sorted items back in collection.Clear(); foreach (T item in sortedList) { collection.Add(item); } }	using System; using System.Collections.Generic; using Nunit.Framework; namespace MyTests { public class SortingTests { [Test] public void TestSortMinComp() { // Test with empty collection var collection = new List<int>(); SortMinComp(collection, (a, b) => a.CompareTo(b)); Assert.AreEqual(0, collection.Count); // Test with single element collection collection = new List<int> { 42 }; SortMinComp(collection, (a, b) => a.CompareTo(b)); Assert.AreEqual(1, collection.Count); Assert.AreEqual(42, collection[0]); // Test with multiple elements collection = new List<int> { 3, 1, 4, 1, 5, 9, 2, 6 }; SortMinComp(collection, (a, b) => a.CompareTo(b)); Assert.AreEqual(7, collection.Count); Assert.AreEqual(1, collection[0]); Assert.AreEqual(2, collection[1]); Assert.AreEqual(3, collection[2]); Assert.AreEqual(4, collection[3]); Assert.AreEqual(5, collection[4]); Assert.AreEqual(6, collection[5]); Assert.AreEqual(9, collection[6]); } [Test] public void TestSortFast() { // Test with empty collection var collection = new List<int>(); SortFast(collection, (a, b) => a.CompareTo(b)); Assert.AreEqual(0, collection.Count); // Test with single element collection collection = new List<int> { 42 }; SortFast(collection, (a, b) => a.CompareTo(b)); Assert.AreEqual(1, collection.Count); Assert.AreEqual(42, collection[0]); // Test with multiple elements collection = new List<int> { 3, 1, 4, 1, 5, 9, 2, 6 }; SortFast(collection, (a, b) => a.CompareTo(b)); Assert.AreEqual(7, collection.Count); Assert.AreEqual(1, collection[0]); Assert.AreEqual(2, collection[1]); Assert.AreEqual(3, collection[2]); Assert.AreEqual(4, collection[3]); Assert.AreEqual(5, collection[4]); Assert.AreEqual(6, collection[5]); Assert.AreEqual(9, collection[6]); } [Test] public void TestSortSlowButSteady() { // Test with empty collection var collection = new List<int>(); SortSlowButSteady(collection, (a, b) => a.CompareTo(b)); Assert.AreEqual(0, collection.Count); // Test with single element collection collection = new List<int> { 42 }; SortSlowButSteady(collection, (a, b) => a.CompareTo(b)); ~~~~output 4th seperate: using System; using System.Collections.Generic; using Nunit.Framework; namespace MyTests { public class SortingTests { [Test] public void TestSortEasyToRead() { // Test with empty collection var collection = new List<int>(); SortEasyToRead(collection, (a, b) => a.CompareTo(b)); Assert.AreEqual(0, collection.Count); // Test with single element collection collection = new List<int> { 42 }; SortEasyToRead(collection, (a, b) => a.CompareTo(b)); Assert.AreEqual(1, collection.Count); Assert.AreEqual(42, collection[0]); // Test with multiple elements collection = new List<int> { 3, 1, 4, 1, 5, 9, 2, 6 }; SortEasyToRead(collection, (a, b) => a.CompareTo(b)); Assert.AreEqual(7, collection.Count); Assert.AreEqual(1, collection[0]); Assert.AreEqual(2, collection[1]); Assert.AreEqual(3, collection[2]); Assert.AreEqual(4, collection[3]); Assert.AreEqual(5, collection[4]); Assert.AreEqual(6, collection[5]); Assert.AreEqual(9, collection[6]); } } }
		Test acts like duplicate 1 would be deleted. double element is preserved	Passes	Does not use the provided comparison, returns identical list, fails test	

#sabkell
cabal install --lib Hunit
Stack build
Stack test

GPT	Minimize comparisons	Minimise lines	Minimize time complexity	Readability + Simplicity	Testcode
	mergeSort :: (a -> a -> Ordering) -> [a] -> [a] mergeSort _ [] = [] mergeSort _ [x] = [x] mergeSort cmp xs = merge cmp (mergeSort cmp left) (mergeSort cmp right) where (left, right) = splitAt (length xs `div` 2) xs merge :: (a -> a -> Ordering) -> [a] -> [a] -> [a] merge _ [] ys = ys merge _ xs [] = xs merge cmp (x:xs) (y:ys) | cmp x y == GT = y : merge cmp (x:xs) ys | otherwise = x : merge cmp xs (y:ys)	insertionSort :: (a -> a -> Ordering) -> [a] -> [a] insertionSort _ [] = [] insertionSort cmp (x:xs) = insert x (insertionSort cmp xs) where insert y [] = [y] insert y (z:zs) | cmp y z == GT = z : insert y zs | otherwise = y : z : zs	quickSort :: (a -> a -> Ordering) -> [a] -> [a] quickSort _ [] = [] quickSort cmp (x:xs) = quickSort cmp [y | y <- xs, cmp y x == LT] ++ [x] ++ quickSort cmp [y | y <- xs, cmp y x /= LT]	selectionSort :: (a -> a -> Ordering) -> [a] -> [a] selectionSort _ [] = [] selectionSort cmp xs = minElem : selectionSort cmp (removeElem minElem xs) where minElem = minimumBy cmp xs removeElem :: Eq a => a -> [a] -> [a] removeElem _ [] = [] removeElem x (y:ys) | x == y = ys | otherwise = y : removeElem x ys	module SortingFunctionsTest where import Test.HUnit import Data.List (sort) import Data.Function (on) import SortingFunctions -- Test data unsorted :: [Int] unsorted = [3, 1, 4, 5, 2] sorted :: [Int] sorted = sort unsorted -- Test cases testMinimizeComparisonsSort = TestCase $ assertEqual "Minimize comparisons sort" (mergeSort compare unsorted) sorted testMinimizeLinesOfCodeSort = TestCase $ assertEqual "Minimize lines of code sort" (insertionSort compare unsorted) sorted testMinimizeTimeComplexitySort = TestCase $ assertEqual "Minimize time complexity sort" (quickSort compare unsorted) sorted testMaximizeReadabilitySort = TestCase $ assertEqual "Maximize readability sort" (selectionSort compare unsorted) sorted -- Test suite main = runTestTT $ TestList [testMinimizeComparisonsSort , testMinimizeLinesOfCodeSort , testMinimizeTimeComplexitySort , testMaximizeReadabilitySort]
	Pass	Pass	Pass	Error on removeElem: No instance for (Eq a) arising from a use of ‘removeElem’ Possible fix: add (Eq a) to the context of the type signature for: selectionSort :: forall a. (a -> a -> Ordering) -> [a] -> [a] • In the second argument of ‘selectionSort’, namely ‘(removeElem minElem xs)’ In the second argument of ‘(:)’, namely ‘selectionSort cmp (removeElem minElem xs)’ In the expression: minElem : selectionSort cmp (removeElem minElem xs) typecheck(-Wdeferred-type-errors) removeElem :: forall a. Eq a => a -> [a] -> [a] Fix: selectionSort :: Eq a => (a -> a -> Ordering) -> [a] -> [a]	issue with code setup However: ghci> mergeSort compare unsorted [1,2,3,4,5] ghci> insertionSort compare unsorted [1,2,3,4,5] ghci> quickSort compare unsorted [1,2,3,4,5] ghci> selectionSort compare unsorted [1,2,3,4,5]
Lama	sort' :: (Ord a, Ord b) => [(a, b)] -> [(a, b)] sort' = mergeSort . map swap . mergeSort where swap (a, b) = (b, a) mergeSort [] = [] mergeSort [x] = [x] mergeSort (x:y:xs) = mergeSort (x:ys) ++ [y] ++ mergeSort xs where ys = filter (/= x) (y:xs)	sort'' :: (Ord a, Ord b) => [(a, b)] -> [(a, b)] sort'' = foldr (\(a, b) acc -> (b, a):acc) []	sort''' :: (Ord a, Ord b) => [(a, b)] -> [(a, b)] sort''' = quickSort . map swap where swap (a, b) = (b, a) quickSort [] = [] quickSort [x] = [x] quickSort (x:y:xs) = quickSort smaller ++ [y] ++ quickSort larger where smaller = filter (/= x) (y:xs) larger = filter (== x) (y:xs)	sort'''' :: (Ord a, Ord b) => [(a, b)] -> [(a, b)] sort'''' = sort' . map swap where swap (a, b) = (b, a)	import Test.Hspec tests :: Spec tests = do describe "sort'" $ do it "works for empty list" $ do sort' [] `shouldBe` [] it "works for single-element list" $ do sort' [1] `shouldBe` [1] it "works for sorted list" $ do sort' [1, 2, 3] `shouldBe` [1, 2, 3] it "works for reverse-sorted list" $ do sort' [3, 2, 1] `shouldBe` [1, 2, 3] describe "sort''" $ do it "works for empty list" $ do sort'' [] `shouldBe` [] it "works for single-element list" $ do sort'' [1] `shouldBe` [1] it "works for sorted list" $ do sort'' [1, 2, 3] `shouldBe` [1, 2, 3] it "works for reverse-sorted list" $ do sort'' [3, 2, 1] `shouldBe` [1, 2, 3] describe "sort'''" $ do it "works for empty list" $ do sort''' [] `shouldBe` [] it "works for single-element list" $ do sort''' [1] `shouldBe` [1] it "works for sorted list" $ do sort''' [1, 2, 3] `shouldBe` [1, 2, 3] it "works for reverse-sorted list" $ do sort''' [3, 2, 1] `shouldBe` [1, 2, 3] describe "sort''''" $ do it "works for empty list" $ do sort'''' [] `shouldBe` [] it "works for single-element list" $ do sort'''' [1] `shouldBe` [1] it "works for sorted list" $ do sort'''' [1, 2, 3] `shouldBe` [1, 2, 3] it "works for reverse-sorted list" $ do sort'''' [3, 2, 1] `shouldBe` [1, 2, 3]
					Very literal

E. Example Jupyter Notebook



Response

```
output = chain.invoke({"task": prompt })
```

[6] Python

+ Code + Markdown

Output

```
output_with_newlines = output.replace('\\n', '\n')

print(output_with_newlines)
```

[7] Python

```
1. Function Signature and Description

```haskell
introsort :: (Ord a) => [a] -> [a]
```
The `introsort` function takes a list of elements that can be ordered and returns a sorted version of that list using the introspective sort algorithm.

```haskell
introsort' :: (Ord a) => Int -> [a] -> [a]
```
The `introsort'` function is a helper function for `introsort` that also takes a maximum recursion depth in addition to the list to be sorted.

```haskell
medianOfThree :: (Ord a) => a -> a -> a -> a
```
The `medianOfThree` function takes three elements and returns the middle value, or median.

```haskell
insertionSort :: (Ord a) => [a] -> [a]
```
The `insertionSort` function takes a list of elements that can be ordered and returns a sorted version of that list using the insertion sort algorithm.

2. Full Text Explanation

The `introsort` function is an implementation of the introspective sort algorithm, a hybrid sorting algorithm which provides both fast average performance and opti
...

The `medianOfThree` function finds the median of the first, middle, and last element of the list, which is used as the pivot for the quicksort partition.

The `insertionSort` function is a straightforward implementation of insertion sort, which keeps a sorted sublist and repeatedly inserts the next element at the cor
Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

```
weights_file = eval_dir + '\\' + weights_file_name
model_eval_json_path = eval_dir + '\\' + group + '_' + use_case  + '_' + complexity + '_' + model + '.json'

if not os.path.exists(model_eval_json_path):
    with open(model_eval_json_path, 'w') as f:
        with open(weights_file, 'r') as w:
            weight_data = json.load(w, parse_float=decimal.Decimal)

        attributes = weight_data[group][use_case]
        new_json = {attribute: False for attribute in attributes}
        new_json_string = json.dumps(new_json)
        f.write(new_json_string)


with open(model_eval_json_path, 'r') as g:
    eval_crits = json.load(g)

checkboxes = []

def update_json(change):
    eval_crits[change['owner'].description] = change['new']
    with open(model_eval_json_path, 'w') as file:
        json.dump(eval_crits, file)

for key, value in eval_crits.items():
    checkbox = Checkbox(value=value, description=key)
    checkbox.observe(update_json, names='value')
    checkboxes.append(checkbox)

checkboxes_display = VBox(checkboxes)

display(checkboxes_display)
```

✓ 0.0s Python

- [x] Complete
- [x] Relevant
- [x] Correct
- [x] Consistent
- [x] Readable

Evaluation Result

```python
with open(model_eval_json_path, 'r') as g:
    evals = json.load(g)
    eval_crit_names = list(evals.keys())

with open(weights_file, 'r') as f:
    weights_All = json.load(f, parse_float=decimal.Decimal)
    weights = weights_All[group][use_case]

sum_of_weights = 0
total_values = 0

for key, value in evals.items():
        if key in weights:  # Check if attribute exists in weights
            total_values += weights[key]  # Increment the total count of values considered
            if value:
                sum_of_weights += weights[key]

# Calculate the percentage
if total_values != 0:
    percentage = (sum_of_weights / total_values) * 100
else:
    percentage = 0

print(f"Score: {sum_of_weights}")
print(f"Max: {total_values}")
print(f"Percentage: {percentage:.2f}%")
```

[3] Python

```
Score: 13.20
Max: 13.20
Percentage: 100.00%
```

Results & Findings

Hypothesis accepted: Yes

F. Cria docker compose

```yaml
version: "3.8"

services:
  cria:
    build:
      context: ../
      dockerfile: Dockerfile-cpu
    ports:
      - 3000:3000
    volumes:
      # Specify you Llama model path here
      - /home/linus/ggml/llama-2-13b.ggmlv3.q8_0.bin:/app/model.bin
    environment:
      - CRIA_SERVICE_NAME=cria
      - CRIA_HOST=127.0.0.1
      - CRIA_PORT=3000
      - CRIA_ZIPKIN_ENDPOINT=http://zipkin-server:9411/api/v2/spans
  zipkin-server:
    container_name: sirdai-zipkin-server
    image: openzipkin/zipkin
    ports:
      - "9411:9411"SS
```

G. *Initial qualitative evaluation criteria*

Code Generation:

Use Case 1: Assist developers in generating Haskell code for specific tasks or components.

Evaluation Criteria:

a. Code Quality: Assess the generated code's correctness, readability and composition.
b. Efficiency: Measure the ability of the model to produce code that is optimized for performance (how relevant for Haskell?).
c. Customization: Evaluate the model's capability to adapt to specific project requirements and coding standards. (Minor further specify minor changes-> similar output?)
d. Speed: Consider the time required for the model to generate code, ensuring it aligns with development timelines.
e. Integration: Assess the ease of integrating the model into existing development workflows and tools.

Use Case 2: Scaffold Generation for new Projects: Automatically create a basic project structure, including directories, configuration files, and boilerplate code for new Haskell projects.

Evaluation Criteria:

a. Completeness: Assess the generated project structure for its comprehensiveness and readiness for development.
b. Configuration Accuracy: Ensure that configuration files, dependencies, and settings are appropriate for the project's goals.
c. Modularity: Evaluate the generated code for its modularity and extensibility.
d. Customization: Measure how easily developers can customize the generated project structure to suit their needs.
e. Time Saved: Calculate the time saved by developers in setting up new projects.

Use Case 3: Automatic Documentation Generation: Automatically generate code documentation, including function and module descriptions, based on the source code.

Evaluation Criteria:

a. Documentation Quality: Assess the accuracy, completeness, and clarity of the generated documentation.
b. Consistency: Evaluate the consistency of documentation style and formatting.
c. Efficiency: Consider the time and effort saved by automatically generating documentation.

Debugging:

Use Case 1: Bug Identification: Assist developers in identifying and categorizing bugs.

Evaluation Criteria:

a. Bug Detection Accuracy: Measure the model's ability to identify different types of bugs, including syntax errors, logical issues, and runtime errors.
b. Bug Severity Estimation: Evaluate the model's capability to estimate the severity of identified bugs.
c. False Positives/Negatives: Assess the number of false positives and false negatives percentages in bug identification
d. Debugging Efficiency: Determine the time saved by using the model for bug identification.

Use Case 2: Debugging Assistance: Provide developers with debugging hints, suggestions, and potential fixes for identified issues.

Evaluation Criteria:

a. Quality of Suggestions: Assess the relevance and effectiveness of the suggestions and fixes provided.
b. Fix Accuracy: Measure the correctness of suggested fixes.
c. suggested by the model.
d. Developer Productivity: Calculate the time saved and the speed of issue resolution when using debugging assistance.

Use Case 3: Composition Analysis: Assess the usage of composition patterns to ensure that they adhere to best practices and maintainability standards.

Evaluation Criteria:

a. Composition Quality: Evaluate the effectiveness of composition patterns in achieving code modularity and reusability.
b. Code Readability: Assess the clarity and comprehensibility of composed code.
c. Consistency: Check for consistent use of composition patterns throughout the codebase.

Use Case 4: Functional Abstraction Assessment: Evaluate the use of functional abstractions and higher-order functions.

Evaluation Criteria:

a. Functional Abstraction Quality: Assess the effectiveness of functional abstractions in improving code structure and maintainability.
b. Functional Purity: Evaluate the adherence to functional purity principles in the use of abstractions.
c. Readability and Expressiveness: Measure how well functional abstractions enhance code readability and expressiveness.
d. Testing and Debugging: Assess the impact of functional abstractions on testing and debugging ease.
e. Code Reusability: Determine the reusability of functional abstractions in different parts of the codebase.

Testing:

Use Case 1: Test Case Generation: Automatically generate test cases for functions and modules.

Evaluation Criteria:

a. Code Coverage: Measure the percentage of code coverage achieved by the generated test cases.
b. Test Quality: Evaluate the ability of the test cases to reveal edge cases and corner scenarios.
c. Integration with Testing Frameworks: Assess the compatibility of generated test cases with popular Haskell testing frameworks (hunit, hspec, tasty, ...).

Use Case 2: Test Suite Augmentation: Enhance existing test suites by generating additional test cases for Haskell code and improving test coverage.

Evaluation Criteria:

a. Test Suite Coverage: Measure the increase in code coverage achieved by the augmented test suite.
b. Regression Detection: Evaluate the model's ability to identify code changes that require updated test cases.
c. Test Case Quality: Assess the effectiveness of added test cases in revealing bugs and issues.
d. Automation Integration: Determine how well the model integrates with continuous integration and testing pipelines.

Use Case 3: Test Output Analysis: Automatically analyze test outputs to identify failures.

Evaluation Criteria:

a. Failure Detection: Measure the accuracy of identifying test failures.
b. Root Cause Analysis: Evaluate the ability of the model to pinpoint the root cause of test failures.
c. False Positives/Negatives: Assess the number of false alarms or missed failures.
d. Automated Reporting: Determine the quality and usability of automated test output analysis reports.
e. Speed of Analysis: Consider the time required to analyze test outputs and provide results.

References

[1] G. Orru and L. Longo, “The Evolution of Cognitive Load Theory and the Measurement of Its Intrinsic, Extraneous and Germane Loads: A Review”, in *Human Mental Workload: Models and Applications*, L. Longo and M. C. Leva, Eds., Cham: Springer International Publishing, 2019, pp. 23–48.

[2] M. Schurz, L. Maliske, and P. Kanske, “Cross-network interactions in social cognition: A review of findings on task related brain activation and connectivity”, *Cortex*, vol. 130, pp. 142–157, 2020, doi: https://doi.org/10.1016/j.cortex.2020.05.006.

[3] T. McCabe, “A Complexity Measure”, *IEEE Transactions on Software Engineering*, no. 4, pp. 308–320, 1976, doi: 10.1109/TSE.1976.233837.

[4] G. A. Campbell, “Cognitive complexity, because testability != understandability”. [Online]. Available: https://www.sonarsource.com/blog/cognitive-complexity-because-testability-understandability/

[5] V. Lenarduzzi, T. Kilamo, and A. Janes, “Does Cyclomatic or Cognitive Complexity Better Represents Code Understandability? An Empirical Investigation on the Developers Perception”. 2023.

[6] M. Schreiner, “GPT-4 architecture, datasets, costs and more leaked”. Jul. 2023.

[7] B. Wodecki, “AI News Roundup: Microsoft May Have Leaked ChatGPT Parameters”. Oct. 2023.