Bachelor of Science in Software Engineering
June 2024

Code Generation from Large API Specifications with Open Large Language Models

Increasing Relevance of Code Output in Initial Autonomic Code Generation from Large API Specifications with Open Large Language Models

Esbjörn Lyster Golawski

James Taylor

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Bachelor of Science in Software Engineering. The thesis is equivalent to 10 weeks of full-time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author(s):
Esbjörn Lyster Golawski
E-mail: esly21@student.bth.se

James Taylor
E-mail: jata20@student.bth.se

University advisor:
Professor Niklas Lavesson
Department of Software Engineering
Industry advisors:
Ruwan Lakmal Silva
Linus Bjerstedt Blom

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background. In software systems defined by extensive API specifications, autonomic code generation can streamline the coding process by replacing repetitive, manual tasks such as creating REST API endpoints. The use of large language models (LLMs) for generating source code comprehensively on the first try requires refined prompting strategies to ensure output relevancy, a challenge that grows as API specifications become larger.
Objectives. This study aims to develop and validate a prompting orchestration solution for LLMs that generates more relevant, non-duplicated code compared to a single comprehensive prompt, without refactoring previous code. Additionally, the study evaluates the practical value of the generated code for developers at Ericsson familiar with the target application that uses the same API specification.
Methods. Employing a prototyping approach, we develop a solution that produces more relevant, non-duplicated code compared to a single prompt with local-hosted LLMs for the target API at Ericsson. We perform a controlled experiment running the developed solution and a single prompt to collect the outputs. Using the results, we conduct interviews with Ericsson developers about the value of the AI-generated code.
Results. The study identified a prompting orchestration method that generated 427 relevant lines of code (LOC) on average in the best-case scenario compared to 66 LOC with a single comprehensive prompt. Additionally, 66% of the developers interviewed preferred using the AI-generated code as a starting point over starting from scratch when developing applications for Ericsson, and 66% preferred starting from the AI-generated code over code generated from the same API specification via Swagger CodeGen.
Conclusions. Increasing the extent locally hosted LLMs can generate relevant code from large API specifications without refactoring the generated code in comparison to a single comprehensive prompt is possible with the right prompting orchestration method. The value of the generated code is that it can currently be used as a good starting point for further software development.

Keywords: Large Language Models, Autonomic Code Generation, API Specifications, Artificial Intelligence, Prompt Engineering

Contents

Chapter 1

Introduction

Artificial intelligence (AI), which can be defined as the automation of "activities that we associate with human thinking, activities such as decision-making, problem solving, learning ..." [32], has many branches, of which one is generative AI. One of the main goals of generative AI "is to generate (create) new data samples", such as images, audio and text [22]. An example of a tool that assists programmers via generative AI is GitHub Copilot, which can read existing codebases and suggest code as the human is editing a file [10]. However, it only knows about the context of the files already opened [2]. In this domain, the problem of making the AI understand the entire context of a specific problem that the AI hasn't been specifically been trained to solve is a challenge. Overcoming this challenge is very resourceful for the software development life cycle [28] as it can speed up development significantly [6].

The generative AI entity that is currently most used for general purpose conversations and tasks is the large language model (LLM). An LLM is an AI model that is trained on a lot of data which consists mostly of natural language. An LLM can be a useful entity for putting in some natural language via a prompt to generate text with coherent communication [23] via the internal *inference* [22] of the LLM. An LLM functions by splitting the input into tokens, which are essentially numerical representations of words, making them easier for the computer to understand. The LLM then finds the dependencies between the tokens to grasp the meaning of the input [23]. Using an LLM trained on natural language and source code of various relevant software programs, we can prompt the LLM with a software engineering task that it should complete. This makes LLMs a candidate for arbitrary code generation tasks. In many publicly accessible LLMs via services such as Hugging Face [8] and Ollama [24], the fixed-length context constraints hamper the ability to grasp long-term dependencies in the input. This leads to the issue where the context of the prompt is fragmented as the LLM cannot process the entire context at once. This results in impaired performance, particularly for longer contexts, as the AI cannot utilize information from earlier text segments due to these limitations [4]. Overcoming this task is a large research field. Here, it is important to balance the number of prompts to the LLMs while maintaining the context of the given task, all done autonomously, meaning the LLMs or our software needs to decide what is the next step in solving the task at all times.

In this thesis, we aim to examine solutions that can improve the LLMs' code generation abilities when the functional requirements are derived from an API specification, which when decoded by the LLM produces a number of tokens that is greater than the context size of the LLM. To highlight these improvements in a

real-world context, we conduct an experiment, and interviews at Ericsson. We only examine solutions that improve the initial generation of code, meaning we do not refactor previously generated- or existing code, as refactoring already existing code is another large topic and we want to examine the prompting methods that gets the best output on the first try.

1.1 Background

Mitigating the aforementioned limitations about context window sizes can be done by reducing the content an LLM has to process per LLM inference. Because an LLM is essentially a machine that predicts the next token in a sequence of tokens, it will predict the next token even in cases when parts of the context has been omitted by the LLM's context window size. For example, if the LLM is prompted to create an API with one thousand API endpoints, it can happen that it only outputs a smaller portion of them. This means that to improve the LLMs' understanding of the input, we should not only reduce the number of tokens the LLMs have to parse, but also the total number of concepts it has to parse, to fit it in its context window.

This has been attemped to do by tools such as GPTEngineer [25] and GPTMe [3] to simplify the input an LLM has to process. The two aforementioned tools work by the LLM initially summarizing the entire user input and breaking it down into steps which need to be done to complete the specific task. Then, they infer the LLM again once per step to solve the entire task. We evaluate the positives of such tools and then proceed to annotate the most successful aspects of each tool so that our software prototype can produce the best on average most accurate target application according to our predefined requirements.

1.2 Aim and scope

The study aims to evaluate methods that can autonomously orchestrate the inferences to LLMs with minimal human input using local-hosted, open source, commercially utilizable software and AI to generate as much relevant code output as possible for a given large API specification. The API specification is a strict-syntax description of the interface of the API, using the OpenAPI [35] specification standard. The objectives of this research is firstly to prototype a software solution that enables the autonomous orchestration of LLM inferences, which should make possible the generation of relevant, non-redundant code from LLMs, building on existing approaches. Through controlled experiments, we validate the effectiveness of the developed prototype in producing usable and accurate code compared to traditional code generation methods. The practical utility of the generated code is evaluated by professional developers from Ericsson, which highlights the practical value of the generated code in a real-world setting.

For this research a *large* API specification is defined as one that, when encoded into tokens by the LLM, exceeds the LLM's context window capacity. In this text, *open* LLMs are referred to as free, commercially utilizable LLMs that can be run on ones local computer with adequate hardware.

This study utilizes an OpenAPI specification in YAML format to describe the target application, which is a Representational State Transfer (REST) [9] API. The REST API acts as an interface to a larger industrial software system and forwards requests to other applications that communicate via another protocol for web services, SOAP [12]. This is a very specific target application, and is specified in order for the research to be focused on a specific goal. We choose this specific application as the target so that the results can be applicable for potential usage at Ericsson.

This is an area where AI-generation can be applicable since the creation of this REST API is usually a repetitive task as it only reflects the interfaces of the underlying components in the software system. The research can however be applied to other applications in other programming languages too, as the LLMs we use are trained on diverse datasets, spanning over multiple programming languages.

The generation of the application's code assumes that there is no code existing for the application yet, but that the humans have established detailed functional requirements, as well as requirements about which programming languages and frameworks to use. The research also assumes that no data can be leaked outside of the local premises, which means that cloud solutions or SaaS (Software as a Service) platforms for this case are unacceptable. Everything has to be run on the local premises. To assess practically, the research will evaluate the generated code's usefulness by interviewing software developers familiar with the target application's functionality.

The research is scoped to the generation of the specified target REST API due to its use in an enterprise environment, specifically requested by Ericsson. Focusing on the specific application allows the generated code to be easily compared with established systems at Ericsson. This scope also enables the research to recruit Ericsson personnel that have previous experience with these systems. We also scope the code generation to only generate the initial version of code, meaning that we do not prompt the AI to change code that already exists or was previously generated by the AI. This scope is set to focus on refining the code generation, as well as limit the number of possibilities for improving the LLMs' code output.

1.3 Contribution

Our principal contribution is insights into which prompt orchestration methods enhance LLMs' performance in autonomic initial code generation and the identification of limitations in using LLMs to generate REST APIs from large API specifications. Additionally, the solution we develop is designed to be adaptable to other code generation cases, making it applicable to a range of uses beyond merely creating REST APIs in Java. This versatility could serve as a template for developing other types of software.

The insights from this study can enhance the knowledge of what prompting strategies fail, and which improve autonomic, LLM-driven software development with large API specifications. The insights also provide a foundation for further exploration into optimizing autonomic AI code generation and integrating LLMs into the initial stages of software development.

The evaluations from the interviews with Ericsson personnel can validate the practical utility of the AI-generated code. This helps in assessing how the new

methods of code generation compare with traditional practices in terms of usefulness and functionality. Furthermore, they offer valuable insights into how relevant and beneficial these solutions are in real-world scenarios.

1.3.1 Societal and environmental impacts

Code generation using publicly available, pre-existing LLMs offers multiple societal benefits. Using generative AI for code generation can accelerate software development processes, leading to quicker product development and more efficient software creation, as detailed by Rajbhoj et al. [30]. This is assuming that the applications the AI generates are used for environmentally friendly purposes. Utilizing pre-existing LLMs for tasks such as code generation offers significant environmental benefits over training new models from scratch, as it requires considerably less energy and resources. For instance, the LLM LLama 2 13B by Meta consumed an estimated 73,728 kWh during its training [18] from just the GPUs. In contrast, downloading a pre-trained model such as the Llama 2 7B via the Ollama command-line interface mainly involves the emissions associated with running the servers and clients during the download process. However, as highlighted by Rillig et al. [31], LLMs require substantial energy for both training and inference stages, leading to high carbon emissions depending on the energy sources used. Furthermore, the outputs of LLMs, while often appearing expert, lack genuine understanding. This can lead to the propagation of inaccuracies or biased information, especially in sensitive areas such as environmental science, thereby potentially spreading misinformation. Additionally, there are ongoing debates around the ownership and use of the extensive data involved in training these models. Although current regulations aim to support technological advancement and AI-driven innovation, they might also restrict the scope of permissible text and data mining. Such limitations could hinder the broader application of AI in critical areas, inadvertently stifling potential benefits [17].

Chapter 2

Related work

This chapter describes research and works that coincide with our topic on automating the generation of REST API code using LLMs, and how they are related and how our work's topic is not explored in the related works.

2.1 Fully autonomous programming with large language models

Liventsev et al. [16] address the *near miss syndrome* in current LLM models, where outputs are semantically similar to correct solutions but fail technically. They propose the Synthesize, Execute, Debug (SED) method, an iterative process to refine generated code which generates, debugs and tests the generated code. While this study shares our goal of utilizing LLMs for code generation, our research distinctly focuses on using large API specifications for the code generation and not refactoring code that already exists, or has been generated by AI. We emphasize user-centered evaluations and conduct comparative analyses across different LLMs, tailoring our approach to the unique requirements of our target enterprise application.

2.2 Self-collaboration Code Generation via ChatGPT

Dong et al. [7] introduce a framework where multiple LLMs are distinct experts. Each model specializes in different aspects of a software development task, including analysis, coding, and testing. The goal of the framework is for each LLM to collaborate in generating code for complex tasks without any human intervention. This approach significantly improves performance in code generation tasks compared to traditional models. The research is similar to ours as it aims to generate complex coding tasks with minimal human input, however our research extends on a local on-premise execution which aligns with data privacy and security requirements, particularly relevant in corporate settings such as at Ericsson. Additionally, we are focusing specifically on the initial generation of a REST API via a large API specification, which presents unique challenges and complexities not addressed by generalist models.

2.3 AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation

Building on a similar premise, the AutoGen framework, presented by Wu et al. [38], discusses leveraging the capabilities of LLMs within a multi-agent conversational paradigm to automate software development tasks. The AutoGen framework can combine the strengths of LLMs with human guidance and tools to optimize the task the agents should perform. AutoGen supports various conversation patterns with different levels of autonomy, and human-involvement [19]. Unlike Autogen, which only supports Python and requires the Python code to run successfully, our research aims to write code even if the code execution fails. This feature is beneficial because it allows users to fix issues themselves or measure the generated codes accuracy despite execution failures.

2.4 MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework

Another approach for automating software development tasks via AI is the MetaGPT framework, which is introduced by Hong et al. [13]. Similar to AutoGen, MetaGPT also utilizes a multi-agent paradigm. The framework utilizes a communication protocol and standard operating procedures to manage the workflow among agents. However, in contrast to AutoGens approach, MetaGPT specifically assigns predefined roles to its agents, structuring them to simulate the functions and hierarchy of a software company. This research coincides with ours in many aspects, however it does not explicitly focus on large API specifications.

2.5 CAMEL: Communicative Agents for "Mind" Exploration of Large Language Model Society

The CAMEL (Communicative Agents for Mind Exploration of Large Language Model Society) framework, as described by Li et al. [14], proposes a role-playing framework where agents, while also predefined in roles, are focused on exploring cognitive processes and achieving autonomous cooperation. This setup leans heavily on inception prompting to guide agents toward task completion, *simulating a mind* or society of agents working together. While it also aims to automate tasks like the previously mentioned frameworks, it contributes to the field by offering insights into how agents can cooperate autonomously. It coincides with our research in a similar way AutoGen does.

2.6 Auto-GPT for Online Decision Making: Benchmarks and Additional Opinions

Moreover, the Auto-GPT for Online Decision Making paper by Yang et al. [39], introduces Auto-GPT [1] *an autonomous agent that leverages recent advancements in adapting LLMs for decision-making tasks*, bench-marked against realistic scenarios. They concluded that Auto-GPT was successful in adapting complex decision-making tasks with straightforward prompt design, surpassing baseline models specifically designed for these tasks. This research parallels our efforts in auto-generating code autonomously via the usage of LLMs.

2.7 MemGPT: Towards LLMs as Operating Systems

A prompting orchestration of an LLM, called MemGPT, has been introduced by Packer et al. [26], drawing inspiration from operating systems. It is designed to manage extended context beyond the fixed limits of LLMs' context windows with a virtual memory management technique that mimics hierarchical memory systems in a traditional OS. This allows data to be summoned in and out of the *main memory* into long-term storage based on the LLMs decisions. This allows for interactions that are not otherwise possible with typical fixed-context LLMs such as extended interactions and handling larger documents. The ability to manage extensive context and recall necessary information could be applied to our research for extended LLM capabilities when generating relevant coding tasks. However, it does not specifically handle the convolutions of dealing with large API specifications.

2.8 Research gap

While AutoGen, MetaGPT, CAMEL and similar tools highlight the capabilities of LLMs across various aspects of software development, our project is specifically focused on code generation by LLMs for projects defined by large API specifications. Unlike these tools, our approach aims to improve the code output generated by LLMs with prompts that have large API specifications appended, that exceed the LLMs' context sizes. We develop a method that aims to overcome the limitations these large documents pose on the context size of the inference in LLMs, specifically the ability to generate relevant code from a large API specification.

Chapter 3

Research questions

The research questions we aim to answer with this thesis are the following:

RQ1: What prompting orchestration method for pre-trained LLMs can improve the initial autonomous code generation from a large OpenAPI specification?

The motivation for this research question is to find methods of orchestrating the prompting of commercially utilizable, locally hostable, pre-trained LLMs that without refactoring existing code produce more relevant code than a single comprehensive prompt, without human input. The research question focuses specifically on a REST API in Java described by an OpenAPI specification, which is motivated by the usage of it in an existing enterprise system at Ericsson, making the answers to this research question practically utilizable in the enterprise environments.

We aim to answer this research question by prototyping a solution that improves the amount of relevant generated code, and the relevance rate of the code compared to a single comprehensive prompt with the API specification.

RQ2: To what extent can the prompt orchestration method found in RQ1 autonomously generate relevant code using a large OpenAPI specification and pre-trained LLMs?

We want to determine how much of the code the open, pre-trained, commercially utilizable LLMs generates, omits and leaves unfinished during the generation, without any human input or assistance after the initial prompt. We measure the extent of the AIs ability to write code without refactoring previously existing code by determining how many relevant lines of code (LOC) it can produce, using the solution developed to answer research question 1 3.

We plan to answer this question by looking at the different aspects of the success of the autonomic code generation, which is a concept with multiple aspects. We choose the software's file-writing success, relevance of the output, and length of the output as accurate metrics for measuring the extent the prompt orchestration method

found in research question 1 3 autonomously can generate relevant code using a large OpenAPI specification with pre-trained open LLMs.

RQ3: What is the perceived value of using generative AI as a complement in the development of a REST API application?

This research question aims to examine the practicality and effectiveness of leveraging LLMs to develop the initial portions of a REST API in Java at Ericsson, via the AI output that produces the most relevant output, and by generating the OpenAPI specifications used for the REST API by using LLMs.

It seeks to determine whether using AI-generated code can be a viable method for accelerating development processes, specifically focusing on the early stages of coding. By isolating the investigation to the perceived value of AI output in this context, the study aims to provide insights into how our solution can impact productivity without delving into the computational costs or broader economic implications.

Chapter 4

Methodology

This section describes how the research is conducted to answer the defined research questions.

We use literature and the knowledge gained from examining existing solutions to prototype a custom solution and then use it to examine the application of it for autonomic code generation within the specific Ericsson system. We then evaluate the outputs of the tool and use the output to analyze the feasibility of the usage of the tool by conducting interviews with Ericsson developers that are familiar with the target application. A map of the phases the study undergoes can be found in the appendix A.1.

4.1 Researching prompting methods

We research prompting orchestration methods that can improve code generation by reading and examining existing works. Existing research, software and LLMs are found using multiple methods. Firstly, we find literature by crafting search strings that we use to search for relevant articles in literature databases, specifically IEEE Explore, Scopus and arXiv. The search string we use is *"AI" OR "artificial intelligence" OR "large language model" OR "autonomous code generation" OR "prompt engineering"*. We look for relevant software projects by listing existing codebases on GitHub, a host for file repositories, most frequently software source code, accessible via the internet. The search on GitHub is done by listing popular repositories, filtered by topics. We use the terms in the search string as topics here.

We execute the different software with LLMs found on the internet and evaluate if the aforementioned software improves the output compared to a single, comprehensive prompt to an LLM and previously tested software-LLM combinations. If a combination is deemed fruitful, we discern what aspects of the software are the main contributors to the improved output by reading documentation and the source code of the software.

Each software under examination undergoes a systematic evaluation through iterative testing with different prompts. Each iteration employs a prompt that includes the same OpenAPI specification of the target REST API to maintain uniformity in input conditions. We change the prompt by including different combinations of information we deem important for the development of the target application. This information includes aspects of software development that we deem important to understand the context of the application one is developing, from our own experience. Examples of this information are the following:

- Description of the purpose of the target REST API.
- OpenAPI specification of the target REST API.
- Files existing in the codebase.
- Recent changes to the codebase.

Each change in the prompt or software utilizes different LLMs to facilitate a comparative analysis across models compared to the previous run. We compare the different solutions by manipulating the variables relevant to the code generation to evaluate their impacts on the relevance of the generated Java REST API code. These solutions contains many independent variables that can impact the solutions ability to generate code, including the parameters of the software and LLMs. To investigate the methods that can answer research question 1 3, we only tweak the prompting orchestration and prompts, not other variables that impact the results. This includes internal parameters of the inference software that are user-configurable. The number of tests we do per software is determined on a case-by-case basis, depending on if we deem it fruitful.

This investigation is done with every piece of software we deem relevant to our research. When we have examined the different solutions we deem relevant, we develop our own application that takes advantage of the knowledge the other software has given us. To answer research question 2 3, we conduct an experiment phase where we measure the best achieved output out of all the different solutions we have tested compared to a single comprehensive prompt, to know if the output has improved or deteriorated.

4.2 Prompting methods in existing works

During the exploratory phase when we tested software and LLMs, we noted positive aspects about the prompts that we deemed were worth further consideration. This knowledge is the most valuable knowledge we have gained during this research about what prompting orchestration methods that can improve code generation with a large OpenAPI specification. Some of the knowledge we gained by using the tools are the following:

- **GPTEngineer**: This tool breaks down the work needed to complete the task into steps. For every step, the tool prompts the LLM with an instruction on how to complete the step. We deem this method valuable as it can be used to break down the the task to smaller pieces, possibly reducing the task to a size that fits into the context window of the LLM that is being inferred.
- **MemGPT**: MemGPT is a piece of software that orchestrates prompts to an LLM. When a prompt is received from the user, MemGPT prompts the LLM, among other things, to have an internal monologue and store important details in the prompt to a database, which can then be accessed to access important details in earlier prompts again. The concept of storing important details beyond the context of the LLMs memory without losing detail by methods

such as recursive summarization [36] is deemed worth further investigation by us. Using MemGPT with the large specification directly however did not show to improve the LLMs ability to implement code that covers the entire specification. We notice when running MemGPT that this is due to the fact that when in combination with our selected LLMs, MemGPT makes attempts at remembering details about the prompt that includes the specification, but it does not store all details about the specification.

- **AutoGen**: The AutoGen framework allows for the creation of LLM agents that communicate with each other. When using multiple agents via AutoGen and our selected LLMs, as well as a simpler solution we develop, we notice that the decisions the agents make when deciding which agent is the most relevant to perform the next step in solving the coding task for the REST API in Java was impaired by the LLMs' inability to output which agent should be the next. From this experience, we gained the knowledge that assigning the responsibility of deciding what the next step in solving the coding task is to the LLM has its downsides compared to manually deciding a flow between steps. We deduce this is due to the LLMs focus being split on not only solving the main task, but also deciding what the next step is. Another reason for preferring a manually controlled flow between steps is that the LLMs we tested failed to follow specified syntax for the next step at times, introducing an additional point of failure.

Using the knowledge from literature and the aforementioned tools, we iteratively combine the found prompting orchestration methods to create a solution that outputs more relevant code content compared to a single comprehensive prompt.

4.3 Solution development

We take on the task of designing and testing a proof-of-concept solution to examine the feasibility of autonomously coding a Java REST API without refactoring. The reason for this is that by only using pre-existing frameworks and software for code generation, we cannot know if a custom-built solution would perform better or worse than using the pre-existing frameworks and software. This task aims to create a system that leverages the strengths of previously examined tools while addressing their limitations in our use case specifically, which could warrant the most accurate and relevant outputs. The custom tool is developed in Python, which is a choice motivated by that Python's undemanding syntax, readability and easy execution make it highly accessible for rapid development, making it suitable for AI applications [20]. Python does however have a disadvantage as slower execution speed compared to compiled languages due to resource intensiveness [20].

4.3.1 Design of implementation

The design process for creating the custom application is motivated by reaching the best output possible by the LLMs autonomously, with no human input and no refactoring of already existing code. To achieve this, we have to create an application

that balances between making the prompted LLM have as much context information as possible to solve the coding task, while not appending too much information to the prompt, as we notice that leads to the LLM misinterpreting what part of the prompt the actual task is. To decide what information is relevant when the LLM is prompted with a coding task, we go with a human software-engineer-like approach, from our experience as software developers. To know what to code and to which file, we include the following items in the prompt:

- The task the human has decided should be the task.
  Create a SpringBoot API based on the given OpenAPI specification. The API is essentially a gateway API that takes input via REST, and then forwards a request to another application, with all the input data translated to XML for the SOAP protocol, and then takes the XML response and translates it to JSON, for every route. Here is the OpenAPI specification: (...)

- Which files already exist in the working directory where the LLM can write to files.

- Which features are already completed.

- An instruction of how the LLM should write to a file.

Including these four items in the prompt improved the LLM output drastically compared to just including the task. It enabled the application to write to files autonomously in successful cases. However, due to the longer prompt, the results show that the output from the LLM contains fewer relevant LOC that it outputs compared to when trying to complete everything in a single comprehensive prompt. We mitigate this by dividing the size of the output the LLM has to do by splitting up the task in smaller parts. We do this by introducing a new phase before the coding phase; the *planning phase*. The planning phase is done by prompting the LLM to output a list of files with each file having a description of the relevant content of the file that is needed to complete the entire task. The following text is the system message for the planning phase that instructs the LLM to provide a detailed plan for fulfilling a specific task, including a numbered list of files needed and their respective descriptions:

> "You are a planning assistant for software coders. The software coder wants to do this: (...) Give a numbered list of files that need to exist to fulfill the coders task. For each file in the list, write exactly what needs to exist in the file (classes, imports, etc.). Make the description of each file so thorough that one could read it and complete the task without knowing any more context. That means you should include the full imports that are relative to other files in the codebase, so that the coder does not have to look up if the imports are correct. You do not need to include the imports to dependencies that are outside of this codebase. Example:
> ‘src/main/java/com/restapi/Application.java‘
> * Main class annotated with ‘@SpringBootApplication‘
> * Import necessary Spring Boot libraries

> In the filenames, include the relative path to them where they should be, i.e. src/main/com/restapi/etc...
> Do not be nice in the response. Just return the plan as a list. Do not write the content of the files, just a short summary of what the file should have."

The result of including this in the prompt shows that the number of items a list has varies a lot between every planning phase and LLM. In the successful cases when the list is successfully written and follows the predefined syntax, we parse the list, and then prompt the LLM with each step that describes the code that needs to be written. We call this the *coding phase*. Using this method of prompting, the total average output by the LLM per execution of our custom solution produces more files, with more content in each file. The coding phase introduces a new system message that instructs the role of the LLM in the current step. It makes it clear for the AI to not waste any valuable output on explaining what it does, and to not write incomplete code. The prompt reminds the AI of getters and setters because we notice it often skips that part.

> "You are a coder that writes code whenever you are requested to. You do not ask for clarifications of the task, you just follow your instincts and coding skills. You do not clarify your decisions, or explain what you do. You just produce code. You never leave anything as incomplete. You finish all the code you start, and you want to complete it. Create all the getters and setters too, as you have it easy to forget those!"

However, using only the specific step appended to the system message as the prompt to the LLM, we notice the output does not adhere fully to the context of the task. This results in a lot of duplicate code output. To improve the output in this phase, we notice improvements by appending the three following pieces of information individually, in each coding phase prompt:

1. The entire plan of the task produced in the planning phase.
2. A history of the previously generated file content by the LLM.
3. A summarization of each entry in the history of the previously generated file content.

Out of these three items and combinations of them, only including the first one improved the output the most. Including all the existing files in the working directory proved to be fruitful at times, however the LLM often started repeating the existing list of files, without outputting any code content. This lead to an average output with a smaller number of relevant LOC. Using the planning and coding phases with the aforementioned prompt content is the solution we found that shows the best output for our use case. We call this solution the *phased prompting* solution.

4.4 Experiment design

To evaluate the performance of the *phased prompting* solution and answer research question 2 3, we run the solution with multiple different LLMs to measure the relevance of the output content, compared to a single comprehensive prompt. All

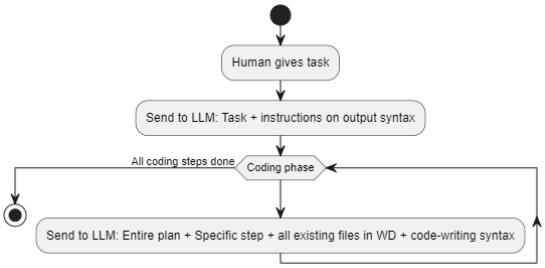

Figure 4.1: Most successful prompting orchestration method

inferences are done with default settings and the recommended context window size per LLM as a parameter for the inference. The large OpenAPI specification we are using in the prompts is provided by Ericsson. The single comprehensive prompt includes the same task as the prompt to the planning phase, but with the description of what the LLM should produce switched from a plan to the actual coding task for the REST API in Java.
The experiment is done as follows:

1. **Select LLMs to evaluate:** The selection of LLMs was conducted through random sampling from a pool of open-source LLMs that were compatible with the Ollama inference platform. This approach was chosen to minimize selection bias.

2. **For every LLM, execute the *phased prompting* application:** Run the application with the aforementioned prompts, adapted to the LLMs internal prompt format. For example, the LLM mixtral8x7b uses the format [INST] {{ .System }} {{ .Prompt }} [/INST] [24]. We execute the *phased prompting* tool three times per LLM to gain an average output.

3. **For every LLM, run inference with a single comprehensive prompt:** We run a single inference with a comprehensive prompt including the same task as the planning phase has, adapted to the specific LLMs prompt format, with instructions of how the LLM should write the code to files and the API specification appended. We execute the single inference three times to gain an average output.

4. **Collect Results:** Gather all outputs including code files and related documents for analysis.

4.4.1 Evaluation metrics

In the REST API that is defined as the target application, it is a difficult task to measure how *complete* the generated code is compared to its functional and non-functional requirements specified in the large API specification. For example, code can exist for a feature defined in the specification without the code fully adhering to its functional requirements. There also exists the case when code for a feature is fully existing and working correctly when used as a standalone component, but does not function in the context of the application. An example of a situation where

that can occur is when there exists semantically incorrect references to other files or modules of code. Counting the number of features specified in the code the LLMs have generated is not also an absolute value of the quality, as features differs in complexity and code needed for a feature to be complete. Some features are also deemed more important than others by the human, i.e. implementing the API endpoints may be more important than creating logic for HTTP error codes.

Software can be *over-engineered*, meaning it contains more code and logic than needed to fulfill the purpose the code exists for. In cases when the generated code is *over-engineered*, we determine on a case-by-case basis if the code generated by the AI is relevant to the purpose of the prompt and the task. We set the granularity of the pieces of the code output that can be deemed relevant or irrelevant to a single LOC. The coding standards we adhere to are specified in the official Java Language Specification, Java SE 22 Edition [29]. In cases when the AI produces an LOC that is longer than what is officially recommended, we split that physical line of content into multiple LOC according to what is officially recommended.

To determine the total number of relevant LOC an execution has generated, we subtract the LOC that are deemed irrelevant from the number of total generated LOC. Duplicates of features in the generated code only count once towards the number of relevant LOC. To deem an LOC relevant it needs to match the following criteria:

1. The code should use the correct language and framework that is defined in the requirement(s).

2. The feature the code is trying to implement should be defined in the requirement(s).

3. The code should not be irrelevant to the requirement(s), e.g. a "toString" Java-function is not needed when it is not used for the endpoint described in the OpenAPI specification.

4. The code should have correct syntax of the language it is written in.

5. The code should not thematically be a duplicate of code that already exists in the code base.

The evaluation of the generated outputs from the controlled experiment [37], involves several key metrics designed to quantify the accuracy of the prompting solution and the single comprehensive prompts to the LLM. These metrics include:

- **Average number of LOC**: The total number of LOC produced by the LLM per task.

- **Average number of relevant LOC**: The number of lines relevant to the task, indicating the precision of the generated code.

- **Percent relevant LOC**: The percentage of LOC that are relevant out of the generated LOC.

- **Average number of files successfully written to**: The number of files that the LLM successfully writes to. A write is only deemed successful if only the code is written to the file. No characters that are products of the code-writing syntax should be written to the file for the file-write to be deemed successful, i.e. the ‘-characters encapsulating the code in the Markdown syntax for code blocks [11].
- **Average number of times LLM failed to write to a file**: Number of times the LLM failed to write to a file by not following the code-writing syntax successfully. Includes the times the LLM produced code relevant to the task, but did not follow the file-writing syntax at all.
- **Percent successful files-writes**: The percentage of the times the code successfully was written to a file.

We deem these aspects as an accurate measure for an LLM's ability to autonomously generate code. The number of files and the percentage of files that were successfully written to are both relevant to include as splitting content between multiple files is a good idea to increase readability in Java [29]. The LLMs ability to correctly follow the syntax on how to write code so that our solution can correctly parse it is crucial for the autonomy of the code generation solution. However, if an LLM correctly produces code for the entire specification but fails to follow the specified syntax, we decide to still include all code generated in the relevant LOC, as some LLMs may be more fit to follow certain syntax better than others.

4.4.2 Test environment

The experiment will be conducted in a controlled environment to ensure consistency across every run.

4.4.2.1 Hardware specification

The *phased prompting* solution is run on a laptop computer with an Intel Core i5-1145G7 CPU running at 2.60GHz with 32.0GB of RAM, with an onboard Intel Iris Xe Graphics graphics processor. The LLM inference is done on a separate computer, accessed via the local area network via the *phased prompting* Python application. The computer running the LLM inferences have an Intel Xeon Silver 4310 CPU running at 2.10GHz with 131.6GB of RAM, with an NVIDIA RTX A6000 GPU.

4.4.2.2 Software specification

The *phased prompting* application is run via Python 3.10.12 in a Python virtual environment, run in Ubuntu 22.04 in the Windows Subsystem for Linux on Windows 10. All inferences are done via the Ollama 0.1.32 REST APIs */api/chat* route. The Ollama API is ran in Docker on an Ubuntu 22.04 machine. All LLMs were downloaded from Ollama on 30th of April 2024 via the *ollama pull <LLM name>* command in the Ollama application. The LLMs we select for the experiment are mixtral8x7b, codellama:34b, llama2:70b_chat_q4_0, nous-hermes2-mixtral:47b, mistral:7b_instruct_q8_0, mistral:7b-text-v2.0-q8_0 and wizardlm28x22b. The names

of the LLMs are as they appear in the Ollama model registry. We choose these as they have shown good performance in common LLM benchmarks and are able to be inferred on our "limited" hardware, compared to larger-size models. From the results of these benchmarks, we draw conclusions whether the *phased prompting* solution improved the autonomic code generation compared to a single comprehensive prompt. To answer research question 3 3 we use the outputs that score the highest across these metrics are selected for further evaluation in an interview process.

4.5 Interviews

Interviews are conducted with Ericsson developers that are familiar with the target REST API in order for us to answer research question 3 3. The interviews are conducted in a structured format asking each question consecutively, focusing on the value of the AI-generated output [37]. The sessions adhere to the survey research guidelines as presented by Petersen [27].

4.6 Blind interviews and further assessments

We conduct interviews about the outputs from the evaluation of the *phased prompting* solution 4.4 to answer research question 3 3.

4.6.1 Application of outputs in blind interview

The AI output with the largest number of relevant LOC, as determined by the previously established criteria, will be placed in a directory. This directory does not disclose that it's content is generated by AI. Adjacent to this directory is another *anonymous* directory containing code generated by Swagger CodeGen [33], a tool which allows generation of API server stubs, documentation and more automatically given an OpenAPI specification. The interviewees do not know which directory contains the AI-generated code and which contains the Swagger CodeGen-generated code until a reveal is made.

We provide these folders to the interviewees three days prior to the interviews. This gives the interviewees time to review and familiarize themselves with the code in each directory and to prepare their thoughts for the interview. We also provide a few examples of AI-generated OpenAPI specifications based on the underlying SOAP web services' WSDL specifications to see if the interviewees see any value in using the generative AI for this use case as well. The reason for conducting a blind interview is to ensure the developers' assessments are as unbiased as possible by any preconceptions about the source of the code. The anonymous setting promotes an unbiased assessment, allowing interviewees to freely express their preferences between the two directories of code. This side-by-side comparison aims to evaluate the effectiveness of the AI-generated code against traditional tools used in similar contexts. The interview questions are the following:

- How many years of experience do you have in professional programming?

- After reviewing these code samples, which one would you rather continue development from? Can you explain why?
- How do the two code samples compare in terms of readability and maintainability?
- (After reveal) What do you think of the AI-generated code, considering it is entirely AI crafted?
- Would you rather start from scratch or from the AI-generated code in other cases when we don't have Swagger CodeGen?
- When creating the OpenAPI specification from hand, would you rather begin from the AI-generated one or from scratch?

Each interview is expected to last approximately 15 minutes. The sessions are recorded and transcribed for analysis, with the consent of the participants. We do not reveal any personal information of any of the participants and try to remove outlying examples that may be used to identify the individuals. Each of the interviews are conducted face-to-face, if possible, for the best possible interaction quality. If an interview cannot be conducted face-to-face, it is conducted via an online-meeting with web-cameras switched on.

4.6.2 Data analysis

The qualitative data from the interviews is analyzed using a content analysis. A content analysis is where text data is systematically coded into categories to quantify and analyze semantic content. This method is useful for making sense of human activities and interactions within software teams according to research presented by Joanna F. DeFranco and Philip A. Laplante [5]. To analyze the data acquired via the interviews, we perform the following steps:

1. **Data familiarization:** Reading the interview transcripts multiple times to gain a comprehensive grasp of the answers provided by the participants.
2. **Develop a coding scheme:** Using an inductive approach, a coding scheme is developed from the data itself. Initially, the data will be examined without predefined codes, allowing themes to emerge naturally from the responses.
3. **Coding:** The coding scheme is applied to the transcript data, marking phrases, sentences, or paragraphs that align with the identified thematic categories related to software development preferences.
4. **Refine codes:** Throughout the coding process, adjustments are made to refine the coding scheme. This includes merging similar codes, subdividing broad categories, and eliminating redundant or irrelevant codes.
5. **Analyze and report:** Analyze the patterns and themes that emerge from the coded data. Report the findings, the process, and the implications of these findings.

4.6.3 Sampling method

Ericsson provided us with nine participants for our interviews that have experience with the development of the target application. The sampling method used here is convenience sampling [34] since every participant is provided to us by Ericsson and have the same workplace. Selecting participants who work in the same office simplifies logistics related to scheduling and conducting interviews. All participants having the same workplace context also provides a consistent background against which their experiences and opinions can be evaluated.

4.7 Validity and reliability of research approach

The validity and reliability of our research and the results varies over the different research methods we conduct in this thesis. This section examines the potential pitfalls of the multiple stages of the study.

4.7.1 Finding prompting orchestration methods

Our chosen research method of iteratively examining different prompting orchestration methods to answer research question 1 3 is relevant because the research is of an exploratory nature. Picking out the state-of-the-art in this field that can potentially improve code generation for large API specifications compared to a single comprehensive prompt comes down to what existing work and methods one chooses to examine. Our decisions of where to look for relevant work influences our results. We do not look for relevant methods anywhere else than stated in the 4.1 section. Every piece of software we statically deem irrelevant could be a lost opportunity for a better solution that we do not examine for our custom solution. A factor that obstructs the validity of this research method is that we have not tested every software there is, nor prompting orchestration method that exists. This factor can influence the results negatively in the aspect that the results are potentially not examining state-of-the-art tools and AI software.

The prompting orchestration method we found that improves code generation output, the *phased prompting* method is only proven to do so using the exact same context, environment and variables as we did when collecting the data on the output, whose validity and reliability is discussed next.

4.7.2 Data collection for the prompting orchestration solution

Validity: The validity of the experiment we use to collect the output relevance is limited to our specific use case and context, due to the experiment only being performed with one API specification. We do not run the *phased prompting* solution with any other API specification than the OpenAPI specification retrieved from Ericsson. This means the validity of the research is limited to the specific OpenAPI specification we used. We do not have any data on the code generation output on any other large API specifications, and therefore no conclusions should be drawn about the *phased prompting* and single prompt solutions ability to generate code on any other specification size, type or specification. The specific Ericsson OpenAPI

specification is 1 123 lines long, following the OpenAPI 3.0.0 specification standard [21]. The LLM prompting we do is fundamentally affected by the software we use for inference, as well as downloading AI models, which is Ollama in our case. The motivation for using Ollama is that it abstracts a lot of the details for LLMs and provide default settings per LLM. This results in a larger focus on the prompting orchestration, rather than the LLM inference parameters and details. This means we do not try any different combinations of parameters LLMs can be provided with during inference.

Reliability: Using the same software, LLMs and prompts as described in 4.4 in conjunction with the specific OpenAPI specification, the same average code generation ability should be measured using the same metrics to retrieve statistics that are comparable to ours. The stochastiveness of current open LLMs increases the randomness of the outputs, which is the fundamental reason that examining the average of the output relevance is more reliable when measuring the individual outputs. On the other hand, our sample size is low with only three data points per prompting method-LLM combination. This is a low sample size and treats data points equally, when each data point could potentially present as a statistical outlier when measuring the median of a larger sample size.

4.7.3 Data collection of the perceived value of using generative AI for developing the target application at Ericsson

Validity: We only use one method and tool, Swagger CodeGen, to generate the code stubs for the comparison with the AI-generated code. The preferences of the interviewees could potentially be completely different if another tool was used to generate the non-AI-generated code. Therefore, the answers for this question are relative to the specific tool that was used to compare the AI-generated code to.

Interviewees are not disclosed which directory of code was generated by the *phased prompting* solution and which was generated from Swagger CodeGen, ensuring that their evaluations were based solely on their opinions of statically reviewing the code. However, in some cases the interviewees can figure out which code is generated by which solution by recognizing patterns in either code examples, which can reintroduce biases. The diversity in experience among the interviewees, although beneficial in capturing a wide range of perspectives, could also lead to variability in the ability to critically assess the code, particularly considering their varying familiarity with OpenAPI specifications and the target application with its technologies. Each interview is recorded and subsequently transcribed. Following transcription, the data undergoes a content analysis in the original language to ensure clear understanding of the answers.

Reliability: All the interviews are recorded with accurate transcriptions, ensuring we preserve the exact responses of the interviewees. The analysis is performed using both qualitative coding techniques and quantitative methods to ensure understanding of the feedback on the code generated by both methods. The coding scheme is developed based on readings of the transcripts and refined as analysis progresses.

Chapter 5

Results and analysis

In this chapter, we present the results on if the *phased prompting* method improved the output when generating code via open, commercially utilizable, publicly available LLMs. From these results, we analyze the extent the LLMs can autonomously generate relevant initial code for the target REST API, to answer research question 2 3. Lastly, this chapter presents the themes of values and statistics of preference the Ericsson developers express about the value of the AI-generated code, to answer research question 3 3.

5.1 Results of the found prompting orchestration method's code generation

Tables of the execution metrics of all the runs with the *phased prompting* and single comprehensive prompt solutions are summarized in tables 5.1 and 5.2.

Content relevance: The results show that using the *phased prompting* orchestration solution produced 361 more relevant LOC on average in the best average statistic (mixtral8x7b as LLM) than in the best case of the single prompting method (mixtral8x7b as LLM). This is an increase of 647% compared to the best average in the single prompting method. The results also show that in the best average statistic (mixtral8x7b as LLM), the *phased prompting* method produces more relevant LOC, but a smaller percentage of relevant LOC, 67% compared to 91%.

Table 5.1: Code benchmarks for different LLMs using the *phased prompting* and single prompt methods

Prompting method	LLM name	Avg. no. LOC	Avg. no. relevant LOC	Percent relevant LOC
Phased prompting	mixtral8x7b	637	427	67%
Phased prompting	codellama:34b	259	24	9%
Phased prompting	llama2:70b-chat-q4_0	479	297	62%
Phased prompting	nous-hermes2-mixtral:47b	0	0	-
Phased prompting	mistral:7b-instruct-q8_0	140	0	0%
Phased prompting	mistral:7b-text-v2.0-q8_0	2	0	0%
Phased prompting	wizardlm28x22b	0	0	-
Single prompt	mixtral8x7b	72	66	91%
Single prompt	codellama:34b	0	0	-
Single prompt	llama2:70b-chat-q4_0	0	0	-
Single prompt	nous-hermes2-mixtral:47b	0	0	-
Single prompt	mistral:7b-instruct-q8_0	0	0	-
Single prompt	mistral:7b-text-v2.0-q8_0	0	0	-
Single prompt	wizardlm28x22b	0	0	-

File writing: The results show that using the *phased prompting* method successfully wrote to files 8 more times in the best average statistic (mixtral8x7b as LLM) than in the best case of the single prompting method (mixtral8x7b as LLM). This is an increase of 266% compared to the best case in the single prompting method. The results in table 5.1 show that the best performing LLM with the *phased prompting* method, as well as via the single prompting method is mixtral8x7b. This LLM was the most successful at following the code-writing instructions, and never failed to correctly write code to files in the working directory. In 10 out of 14 runs, the LLMs did not produce any relevant output. However, when prompted to create a simple REST API with a single route that translates the input JSON to XML, without any API specification appended, they succeeded. On the contrary, when including the large OpenAPI specification, they failed to produce any relevant output, as seen in table 5.1.

Table 5.2: File-writing success benchmarkss for different LLMs using the *phased prompting* and single prompt methods

Prompting method	LLM name	Avg. no. files	Avg. no. files failed code-writing	Percent successful writes to files
Phased prompting	mixtral8x7b	11	0	100%
Phased prompting	codellama:34b	1	4	14%
Phased prompting	llama2:70b-chat-q4_0	8	3	39%
Phased prompting	nous-hermes2-mixtral:47b	0	0	-
Phased prompting	mistral:7b-instruct-q8_0	0	4	0%
Phased prompting	mistral:7b-text-v2.0-q8_0	1	0	0%
Phased prompting	wizardlm28x22b	0	0	-
Single prompt	mixtral8x7b	3	0	-
Single prompt	codellama:34b	0	0	-
Single prompt	llama2:70b-chat-q4_0	0	0	-
Single prompt	nous-hermes2-mixtral:47b	0	0	-
Single prompt	mistral:7b-instruct-q8_0	0	0	-
Single prompt	mistral:7b-text-v2.0-q8_0	0	0	-
Single prompt	wizardlm28x22b	0	0	-

No run produced an application covering every relevant aspect of the OpenAPI specification. No run produced code that could be compiled into an executable Java program, with the most prevalent reason being due to imports being incorrect, as well as imports of data types existing in previously generated files not being referenced correctly. This issue occurred in both the *phased prompting* runs and the single prompt runs. To examine the usefulness of the AI-generated code for this use case, we can examine the opinions of the developers that are experienced with the target application.

To answer research question 1 3, the *phased prompting* method of prompting the LLM to produce a plan upon which files with their individual code content should be generated, then parsing the list and finally programmatically prompting the LLM once per parsed step in the plan with the relevant step and the entire plan can improve the amount of relevant output while remaining a high relevance rate in the output when using commercially utilizable pre-trained LLMs to autonomously generate a REST API in Java from a large OpenAPI specification.

To answer research question 2 3, the extent the *phased prompting* method can autonomously generate relevant code for the target application by the large OpenAPI specification using existing commercially utilizable pre-trained LLMs is variable, and at it's most extensive output, it does not generate code that covers all functional requirements.

5.2 Content analysis on interviews

As we delve into the interview analysis, we categorize the responses into several thematic codes: *code sample preference*, *views on the AI-generated code*, *preferred starting points for projects* and *evaluation of AI-generated OpenAPI specifications*. These categories aim to provide an examination of the value of the AI-generated content, to answer research question 3 3.

5.2.1 Code sample preference

The majority of responses favor the code generated in example 1 by the *phased prompting* solution with the output from one of the outputs by mixtral8x7b, over the Swagger CodeGen-generated example because it was simpler, which made it easier to understand and build upon. The skeletal- or minimalist nature of the code was frequently highlighted with a positive attribute because it aids in understanding the basic structure without the distraction of excessive detail. The interviewees often cited concerns about the maintainability of example 2, generated by Swagger CodeGen, noting it was too complex or over-engineered. The AI-generated code was perceived as more maintainable due to its straightforwardness.

> "If you just look at it, less is more. If I were to get help from a tool, I would want skeleton code. I don't want a bunch of stuff thrown in my face, like you get in the other example. It has generated a terrible amount of things, so I hope then that it's not AI, but that it's some tool."

> "I prefer the first example because it is understandable. The other just spit out a bunch of objects and is not maintainable. Number one can definitely be built upon, It has at least a sensible structure."

There was an outlier that preferred the Swagger CodeGen code, valuing its completeness and more structured setup. This might indicate that a segment of developers might favor a more robust template that they can refine rather than build up from a skeleton.

> "The second is much more implemented than the first. The names are better on the first one but I don't know how correct it is. I would probably have used the second and stripped away what isn't needed, if I had used any of them. The reason is that you get all the objects pre-generated so that you can use them and all the paths are correct"

One interviewee preferred neither specific example unequivocally. One interviewee leaned slightly towards the AI-generated code for having less to remove but did not find either particularly good. Another answer highlighted difficulties in understanding how elements integrate in both examples, suggesting a need for clearer architectural guidance or diagrams.

> "I wouldn't prefer either. One thing that was a bit difficult was to understand how everything flows together. It would be better with a sequence diagram so you know how it was meant to function, when there's a lot of code and you don't know how it hangs together."

Most of the interviewees showed a preference for using the AI-generated code as a starting point for development. The preference mostly came from the Swagger CodeGen-generated code over-complicating matters. Some of the interviewees noticed the familiar way in which the second example over-complicated code stubs. The findings suggest that while many developers favor a simpler approach for ease of understanding and quick starting points, there is a critical need to balance how many details are provided initially, as too few details can be as limiting as too many.

5.2.2 Views on the AI-generated code

The overall sentiment range from cautiously optimistic to critical, with several nuanced views on the potential applications and limitations of AI in code generation.

> "Not everything was correct, but they're easy fixes. It takes 2 minutes and then it's good to go. As long as there's still some job left for us, I'm satisfied."

> "I would say that it looks neat. It nicely points out 'here you should write your code' in a nice little comment. I couldn't ask for much more, really. They usually say the fewer comments the better because then the code explains itself, and that's how I feel it is here. There are no strange names. I have nothing to complain about, very good."

> "It doesn't compile... Otherwise, it's nice to have something to go after. The only time I feel I need to use AI-generated code is when you enter a new framework so you can prototype a bit faster. I would have used it first just to steal some stuff to build something else. But basically, That's how I would have used it."

Other interviewees offered more critical perspectives on the AI-generated code, expressing concerns about its reliability and accuracy. They highlighted the need for human oversight and review when using AI for coding. While AI can assist in the development process, it is not yet capable of replacing careful and thorough human review. Concerns were also raised about the code's lack of logical structure, particularly in its object-oriented aspects, and the absence of necessary annotations or comments.

> "I think it's a bit thin, and when I've generated AI code myself, you get a lot of comments which are completely missing in this. I think this lacks documentation altogether but I guess you can easily get that through AI-generated code."

5.2.3 Preferred starting point for further development

The majority of the respondents see value in starting with AI-generated code, particularly as a means to speed up the initial stages of development and establish a common groundwork for further enhancements.

> "I can actually imagine starting from the AI-generated code stubs."

> "There's a lot of API stuff lately. Boilerplate deluxe: doing it this way saves us several days."

A significant minority of interviewees prefer starting from scratch compared to using the AI-generated code. However, for larger or more complex projects where a deeper understanding and control over the codebase is crucial, some where skeptical. This suggests a balanced approach might be optimal, where AI-generated code is used to kickstart projects but with sufficient flexibility and documentation to allow developers to adjust, refactor, or rewrite as necessary to meet specific project needs and quality standards. The lack of documentation is due to us prompting the LLM to not add comments, because we noticed during the prototyping phase that the LLM produced less code when adding comments.

> "It depends unfortunately, if it's going to be a bigger thing I would start from scratch. But to put something together quickly and easily as a prototype I would definitely use this."

> "It might save some time at the beginning but it will punish you later because you barely know what it has done, so I would rather do it by hand."

5.2.4 Evaluation of AI-generated OpenAPI specifications

The majority of the interviewees appreciate the efficiency and ease of starting with AI-generated OpenAPI specifications due to the complexities and tedious nature of writing the specifications manually.

> "I don't understand how there aren't better things already, I would definitely have used something like that."

> "I hate writing Swagger, so I would rather have started from a generated one, like the third example."

The other interviewees express a preference for starting from scratch to ensure accuracy and clarity, indicating a mistrust or discomfort with the level of detail or correctness in AI-generated outputs.

> "I really don't know what the goal is more than the specification I got, so I would have started myself. When there's a lot like this it's hard to know if it's right or wrong."

The analysis reveals a generally positive reception towards AI-generated OpenAPI specifications, one interviewee mentions *"there is more use for it here..."* referring to generative AI. This sentiment indicates the potential for AI-generated tools to enhance productivity, when prototyping or dealing with repetitive and structured tasks like OpenAPI specifications.

Table 5.3: Summary of Interview Responses

Interviewee	**Years of experience**	**Preferred generated code**	**Would develop from AI-generated code over scratch**	**Would write API spec. from AI-generated spec. over scratch**
1	4	AI-generated	Yes	Yes
2	7	AI-generated	Yes	Yes
3	19	None	Yes	Yes
4	17	AI-generated	Yes	Yes
5	11	AI-generated	Yes	Yes
6	8	Swagger CodeGen	No	Yes
7	8	AI-generated	No	No
8	23	AI-generated	Yes	Yes
9	7	None	No	No

The interviews with the Ericsson developers show that 6 out of 9 developers would rather start with the AI-generated code than from the code generated by Swagger CodeGen from the same API specification. 6 out of 9 developers would continue development from the AI-generated code over starting from scratch, and 7 out of 9 developers would continue writing the OpenAPI specification from the AI-generated one, rather than starting from scratch.

Chapter 6

Discussion

The results of the AI code generation with a large API specification present skeptical values of the both solutions' abilities to generate code for a REST API in Java without any refactoring. No run produced an entire functional REST API with all the functional requirements specified in the OpenAPI specification. Nor did a single run produce all content to cover all aspects of the specification, such as API endpoints, request and response objects, response codes and security. However, the results presented in table 5.1 show that using a specific prompting orchestration, in our case particularly the *phased prompting* solution, we can increase the number of relevant LOC generated compared to a single comprehensive prompt. The measured extent to which the solution can improve the number of relevant LOC is fundamentally affected by the LLMs we chose. One potential reason that mixtral8x7b showed the success it presented, was due to us finding success with it in the early stages of the development of the custom solution. This led to us continuing development with mixtral8x7b as an example of the most successful case, evaluating the changes in the output by the most successful case, while using the other LLMs during development as a less successful case. If another LLM had relatively more success in the early stages of the development, that could potentially had been the LLM with the most amount of relevant LOC during the controlled experiment.

6.1 AI-generated output

The contents of the generated files show that the LLMs' output differs in what content is generated. The total sum of the AI-generated code includes code for every part of the OpenAPI specification, such as API endpoints, JSON to XML conversion and vice versa, SOAP integration, security, request objects, response objects and response codes. These areas varied per run as all areas were implemented to different extents. However, every run when the LLM successfully produced code always implemented the main application file for the REST API, which is not listed in the OpenAPI specification, but elsewhere in the prompt. As there exists non-AI tools that reliably automatically generates the skeleton, API stubs and boiler-plate code for many different types of API specifications such as Swagger CodeGen, the advantage AI can have over these tools is that it has the potential to generate code that is not specified in the API specifications. This means AI can generate features that the automatic generation tools do not offer, such as the integration toward other web services. In our case, via SOAP.

6.2 Best case relevance loss

From just viewing the results that show that the best case of the single prompt outperforms the best case of the *phased prompting* solution, one can argue that the reason for the larger portion of relevant content in the single prompt run is due to the context being handled internally in the LLM with its context-handling functionalities. This is in contrast to the context having to be transferred from the planning phase to the coding phase, and between the different coding steps via a sequence of prompts. However, these results should be viewed with consideration, as the results are only based off three runs for each prompting method, respectively.

6.3 Significance of benchmarks

The benchmarks are only applicable to our specific use case, with our specific large API specification. We have not performed the experiment with any other API specification, any other type of software specification, nor any other sizes of specifications. Therefore, no conclusions should be drawn about the results' applicability to other use cases. On the other hand, the *phased prompting* solution can potentially be used in other use cases with other software specifications.

6.4 Planning phase discussion

By diving the task to multiple steps in the planning phase, the best average case statistic show that we achieve more relevant content. This is due to the planning phase successfully splitting the task into multiple steps, and then there being multiple prompts toward the LLM. We noticed that in the majority of the executions during the development phase of the study, mixtral8x7b only outputted content up to a certain length. If we prompted it to produce a greater number of classes and functions in Java, it did so, but reducing the content per class or function. This is a balance where we have to properly weigh between number of things to code per prompt, and the size of content per thing to implement per prompt. Because this study only touches upon the initial code generation, meaning no refactors can be made to the generated code, we need to maximize the quality of the code generation for each feature of the application on the first try. This means the prompts need to contain enough detail to not require refactors. This means that if a potential future solution determines a planning phase is relevant, it needs to produce more detailed output than what our *phased prompting* solution could do. In our solution, the planning phase is done via one prompt only, and produces only as much content as the aforementioned certain length limit can fit. This means that only using the open, commercially utilizable, publicly available LLMs we selected, we can produce more detailed steps for the planning phase via multiple prompts. This is something that future extensions of this research can consider. This could potentially be implemented by recursively breaking down the steps produced into as tiny pieces as possible, making each prompt during the coding phase as detailed as possible. However, there needs to be future work that examines ways to keep the context of the solution intact when breaking down the task into steps.

6.5 Value of AI-generated output

The sentiments towards AI-generated code were quite similar despite the interviewees' varying years of experience. A majority of the interviewees preferred the code generated by the AI solution, *phased prompting*, over the more extensive code output generated from the Swagger CodeGen tool. The results show that the developers interviewed have a positive attitude about having the AI to automate the generation of *boilerplate* code, as it can reduce the time required to start new projects and focus efforts on more complex tasks of REST API development in Java. The developers expressed concerns about AI-generated codes complexity and reliability, suggesting that while the initial structure is helpful, they often need to apply modifications to meet their projects' needs. This is a noteworthy balance between the desire for automation and the need for human oversight. The developers appreciate the speed and assistance the AI tools provide but remain skeptical of their ability to handle more specific and complex development tasks without the human intervention.

Writing an OpenAPI specification can be monotonous, as the interviews convey. The enthusiasm for AI-generated specifications suggests that developers appreciate tools that can decrease the burden of these initial, often repetitive tasks. While the automation of specification writing was welcomed, there were concerns surrounding the accuracy and completeness of AI-generated specifications.

Chapter 7

Conclusions and future work

7.1 Conclusion

Our research highlights that by dividing the AI code generation process into a planning and a coding phase with multiple prompts, we were able to produce a greater quantity of relevant code output compared to a single comprehensive prompt. This method has proven to improve the output quality of code generated by commercially utilizable open pre-trained LLMs, in the context of developing a REST API in Java from a large OpenAPI specification for the specific enterprise Ericsson system. However, in the best case, the results show that the percentage of relevant code generated is 67% with the *phased prompting* solution, compared to 91% with the single comprehensive prompt, which is a 26% decrease in relevant content. A key takeaway here is that even with a lot of guidance, the LLMs could not produce the target REST API on it's own. Using our method, it is most likely more resource- and time efficient to guide the LLMs with human assistance, i.e. in a *chat bot* implementation, instead of forcing an autonomous code generation solution. The collaboration with Ericsson provided a practical real-world environment to test our solution. Feedback from professional developers during blind interviews highlighted the practical application of generative AI in supporting and accelerating the software development life cycle. Importantly, developers frequently saw the value in using the AI-generated code as a baseline for initiating development projects. Most developers chose to use the AI-generated outputs, not just as a starting point for REST API development but also for writing OpenAPI specifications. However, it was also noted that while the generated code served as a good start, it often lacked the necessary detail and proper documentation, and the code would not compile without further human intervention.

7.2 Future work

As presented in the 4 and 5 chapters, we find a method for prompting LLMs that improves the number of relevant content the LLMs produce. However, we do not close the research gap regarding autonomic code generation with large API specifications we presented in the 2 chapter. Prompt engineering is large topic that has a massive spread of research. We do not state that our solution is the only solution that can improve the quantity of relevant LOC outputted from an autonomic AI coder. Since there is theoretically an infinite number of ways you can form a string of characters

to prompt an LLM with, it is probable that our solution is not the only method that improves the output.

This study would be interesting to extend with LLMs that we did not select. We were limited to using LLMs whose size did not exceed the size of the VRAM on the GPU 4.4.2, which is 48GB of GDDR6 memory. Any LLM exceeding this size crashed the inference. Using LLMs with larger context window sizes could potentially increase the relevance and the length of the code generated.

This study is limited to the initial code generation, meaning the LLMs cannot refactor code that has already been produced. Other methods for initial code generation that were not examined in this paper that we deem are worth examining are the following:

- As mentioned in the 6.4 section of the discussion chapter 6, increasing the amount of relevant content can be done by making the *planning phase* output more explicit steps of how to solve the given coding task. There needs to be more research of how to produce a more extensive list of tasks that the coding phase can take on.

- Using other software for inference. Our experiment used Ollama for inference. We have not performed any experiments on if our prompting orchestration method performs better with other software for LLM inference.

- Using LLMs with different internal architectures. The *transformer* architecture used in many LLMs imposes great memory demands, which limit their ability to handle long sequences, thereby creating challenges for tasks involving extended sequences or long-term dependencies. Liu, et al. presents a distinct approach, *ring attention*, which empowers the training and inference of sequences with lengths that scale in proportion to the number of devices used, essentially achieving near-infinite context size [15].

- Splitting the specification manually or programmatically and then prompting the LLM with each part individually. This method could reduce the size of the prompts, but is however not always acceptable as one may want to send the context of the entire task to the LLM during the planning phase in order for it to come up with a better solution, such as a different structure of the endpoints in a REST API. This is also not an option in cases when many parts of the specification are dependent on other large parts of it, such as the structure of specific JSON objects described in an OpenAPI specification. In this case, you could make the input to the LLM smaller by splitting up the large objects too, for example splitting a request object by its properties, and prompt with a small number of properties at a time.

- For specifications or descriptions of applications that are not easily programmatically splittable, such as natural language, a recursive-summarization approach could be examined [36]. This method has been shown to improve the long-term memory in conversations, however no performance gain have been documented on code generation tasks.

References

[1] AutoGPT, "Autogpt," 2023, accessed on: 10/04/2024. [Online]. Available: https://autogpt.net/

[2] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," in *Proceedings of ACM on Programming Languages, vol. 7*, 2023.

[3] E. Bjareholt, "Welcome to gptme's documentation!" 2023, accessed on: 04/13/2024. [Online]. Available: https://erik.bjareholt.com/gptme/docs/

[4] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov, "Transformer-xl: Attentive language models beyond a fixed-length context," *arXiv preprint 1901.02860 [cs.LG]*, 2019.

[5] J. F. DeFranco and P. A. Laplante, "A content analysis process for qualitative software engineering research," *Innovations in systems and software engineering*, vol. 13, no. 2-3, pp. 129–141, 2017.

[6] G. Deniz and S. Harrysson, Hussin, "Unleashing developer productivity with generative ai," 2023, accessed on: 13/05/2024. [Online]. Available: https://www.mckinsey.com/capabilities/mckinsey-digital/our-insights/unleashing-developer-productivity-with-generative-ai#/

[7] Y. Dong, X. Jiang, Z. Jin, and G. Li, "Self-collaboration code generation via chatgpt," *arXiv prepriunt 2304.07590v2 [cs.SE]*, 2023.

[8] H. Face, "Hugging face text generation inference," accessed on: 10/05/2024. [Online]. Available: https://huggingface.co/docs/text-generation-inference/index

[9] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[10] GitHub, "Github copilot: Your ai pair programmer," accessed on: 10/05/2024. [Online]. Available: https://github.com/features/copilot

[11] J. Gruber, "Markdown," 2004. [Online]. Available: https://daringfireball.net/projects/markdown/

[12] M. Gudgin, N. Mendelsohn, M. Nottingham, and H. Ruellan, "Soap message transmission optimization mechanism," 2005. [Online]. Available: https://www.w3.org/TR/soap12-mtom/

[13] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber,

"Metagpt: Meta programming for a multi-agent collaborative framework," *arXiv preprint 2308.00352 [cs.AI]*, 2023.

[14] G. Li, H. A. A. K. Hammoud, H. Itani, D. Khizbullin, and B. Ghanem, "Camel: Communicative agents for "mind" exploration of large language model society," *arXiv preprint 2303.17760 [cs.AI]*, 2023.

[15] H. Liu, M. Zaharia, and P. Abbeel, "Ring attention with blockwise transformers for near-infinite context," *arXiv:2310.01889 [cs.CL]*, 2023.

[16] V. Liventsev, A. Grishina, A. Härmä, and L. Moonen, "Fully autonomous programming with large language models," L. Paquete, Ed. New York: Assoc Computing Machinery, 2023, pp. 1146–1155.

[17] T. Margoni and M. Kretschmer, "Ai, machine learning and eu copyright law: A socio-legal analysis of ownership issues in training data," in *EPIP 2022, Date: 2022/09/14-2022/09/16, Location: Cambridge UK*, 2022.

[18] Meta, "Llama model card," 2023. [Online]. Available: https://github.com/meta-llama/llama/blob/main/MODEL_CARD.md

[19] Microsoft, "Multi-agent conversation framework," 2023, accessed on: 17/04/2024. [Online]. Available: https://microsoft.github.io/autogen/docs/Use-Cases/agent_chat

[20] S. Mihajlović, A. Kupusinac, D. Ivetić, and I. Berković, "The use of python in the field of artifical intelligence," in *International Conference on Information Technology and Development of Education–ITRO*, 2020.

[21] D. Miller, J. Whitlock, M. Gardiner, and R. Ratovsky, "Openapi specification v3.0.0," 2017. [Online]. Available: https://spec.openapis.org/oas/v3.0.0

[22] K. P. Murphy, *Probabilistic Machine Learning: Advanced Topics*. MIT Press, 2023.

[23] H. Naveed, A. U. Khan, S. Qiu, M. Saqib, S. Anwar, M. Usman, N. Akhtar, N. Barnes, and A. Mian, "A comprehensive overview of large language models," *arXiv preprint 2307.06435*, 2024.

[24] Ollama, "Ollama - get up and running with large language models locally," accessed on: 06/05/2024. [Online]. Available: https://github.com/ollama/ollama

[25] A. Osika, "Welcome to gpt-engineer's documentation," Apr. 2023, accessed on: 23/04/2024). [Online]. Available: https://gpt-engineer.readthedocs.io

[26] C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, and J. E. Gonzalez, "Memgpt: Towards llms as operating systems," *arXiv preprint arXiv:2310.08560*, 2024.

[27] K. Petersen, *Guidelines for Case Survey Research in Software Engineering*. Cham: Springer International Publishing, 2020, pp. 63–92.

[28] K. Pothukuchi and Mallikarjunaradhya, "Impact of generative ai on the software development lifecycle (sdlc)," *International Journal of Creative Research Thoughts,Volume 11, Issue 8 August 2023*, 2023.

[29] J. C. Process, *The Java® Language Specification, Java SE 22 Edition*. Oracle, 2024. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se22/html/

[30] A. Rajbhoj, A. Somase, P. Kulkarni, and V. Kulkarni, "Accelerating software development using generative ai: Chatgpt case study," in *Proceedings of the 17th Innovations in Software Engineering Conference*, ser. ISEC '24. Association for Computing Machinery, 2024.

[31] M. C. Rillig, M. Ågerstrand, M. Bi, K. A. Gould, and U. Sauerland, "Risks and benefits of large language models for the environment," pp. 3464–3466, 2023.

[32] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2010.

[33] S. Software, "Swagger codegen." [Online]. Available: https://github.com/swagger-api/swagger-codegen

[34] S. J. Stratton, "Population research: Convenience sampling strategies," *Prehospital and Disaster Medicine*, vol. 36, no. 4, p. 373–374, 2021.

[35] The Linux Foundation, "New collaborative project to extend swagger specification for building connected applications and services," 2015. [Online]. Available: https://www.linuxfoundation.org/press/press-release/new-collaborative-project-to-extend-swagger-specification-for-building-connected-applications-a

[36] Q. Wang, L. Ding, Y. Cao, Z. Tian, S. Wang, D. Tao, and L. Guo, "Recursively summarizing enables long-term dialogue memory in large language models," *arXiv:2308.15022 [cs.CL]*, 2023.

[37] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in software engineering*, 1st ed. Berlin: Springer, 2012, vol. 9783642290442.

[38] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. Awadallah, R. W. White, D. Burger, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation," *arXiv preprint 2308.08155 [cs.AI]*, 2023.

[39] H. Yang, S. Yue, and Y. He, "Auto-gpt for online decision making: Benchmarks and additional opinions," *arXiv preprint 2306.02224 [cs.AI]*, 2023.

Appendix A

Supplemental information

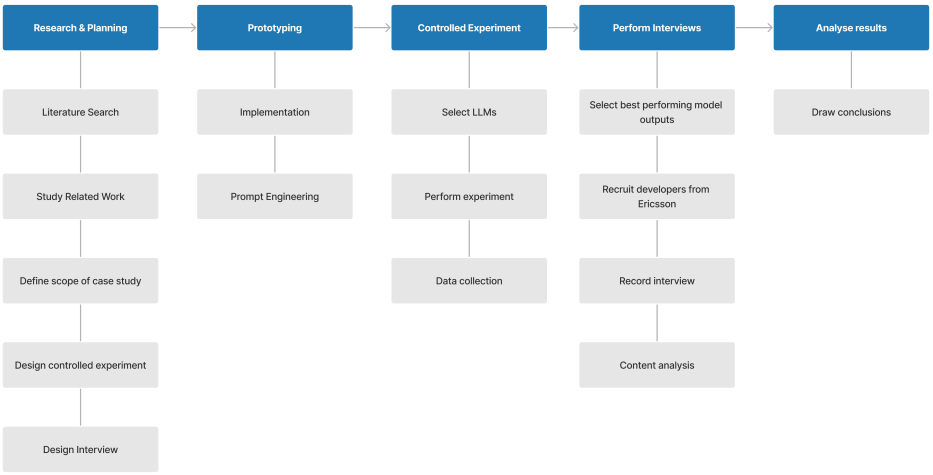

Figure A.1: Map of the phases the research undergoes

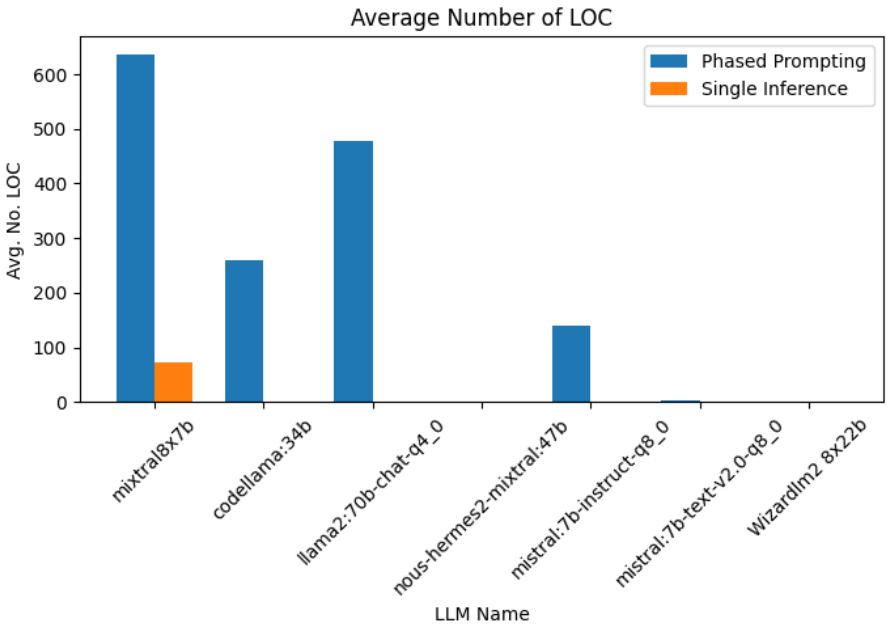

Figure A.2: Average number lines of code

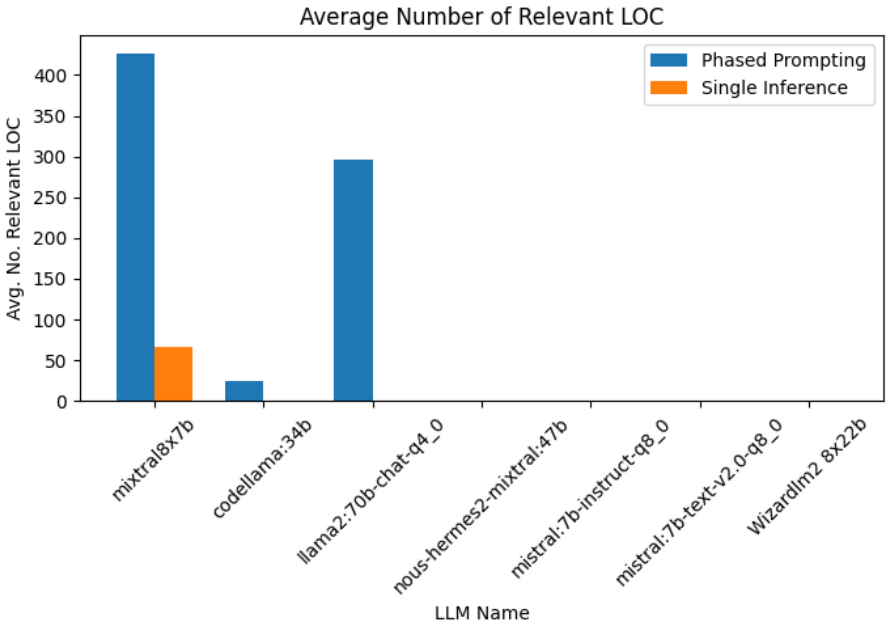

Figure A.3: Average number relevant lines of code

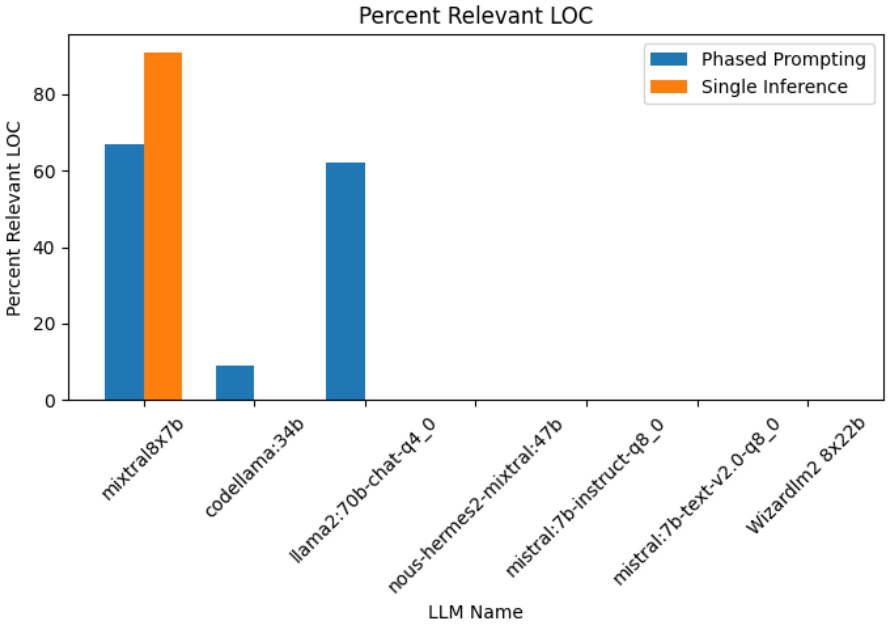

Figure A.4: Average percent relevant lines of code

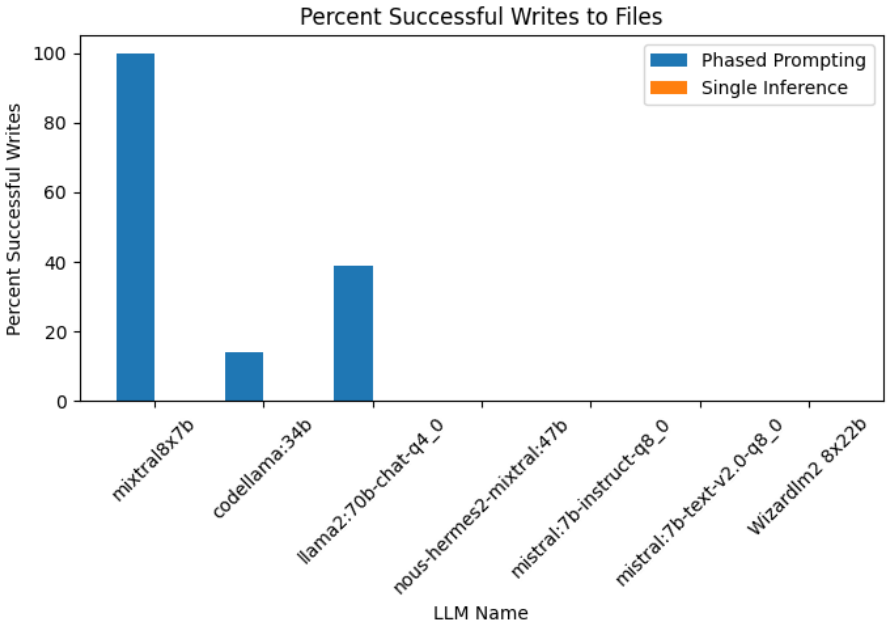

Figure A.5: Average percent write files

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden