BACHELOR'S THESIS 2025

# From Logs to Insights: Optimizing LLM Parameters for Root Cause Detection in CI/CD Build Failures

Erik Skeppner & Kristmann Thorsteinsson

ISSN 1651-2197
LU-CS/HBG-EX: 2025-09

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

# From Logs to Insights: Optimizing LLM Parameters for Root Cause Detection in CI/CD Build Failures

LUND UNIVERSITY
Campus Helsingborg

**LTH Faculty of Engineering at Campus Helsingborg**
**Department of Computer Science**

Bachelor thesis:

Erik Skeppner
Kristmann Thorsteinsson

© Copyright Erik Skeppner, Kristmann Thorsteinsson

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

LTH vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

Printed in Sweden
Media-Tryck
Biblioteksdirektionen
Lunds universitet
Lund 2025

# Acknowledgments

The knowledge that we have gained during the course of this thesis work, and the joy of working on such an interesting topic, have only been made possible by the people who have helped us.

First we would like to express our sincerest gratitude and thanks to Gustaf Lundh, Sandy Sefi and Roger Henriksson for their invaluable guidance, support, and feedback throughout the course of this thesis. Their expertise and encouragement were essential for the completion of this work.

We would also like to thank the entire Axis Tools team for providing the resources, time, patience, and the environment necessary for conducting this research.

Another thanks to the AI team and community at Axis that helped us with access to the internal models, and their advice and patience.

The knowledge shared by these people was an essential part of making this work.

This thesis would not have been possible without the contributions and support of those mentioned above.

**Abstract**

A significant time-consuming aspect of software development is diagnosing and understanding errors. While tools exist to streamline this process, their effectiveness often depends on predefined error patterns. For instance, some tools rely on users to identify and define error identity to later automatically recognize and assist with similar issues when they reoccur. [1]

This thesis investigates whether large language models (LLMs) can be used by developers to read build logs and identify sections in the log that are directly linked to the root cause of failures. The study evaluates LLMs from various providers and explores the impact of prompt engineering, input filtering techniques, and temperature settings on model performance. The goal is to determine whether these parameters influence the LLMs' effectiveness in diagnosing failed build logs.

The evaluation was done with a one-on-one LLM battleground format. Wherein two LLMs with their specific parameter adjustments are faced with the same problem of solving a failed build log. The models are assigned Elo ratings that are updated throughout the course of matches depending on the result of each match.

Model variants were implemented during the course of four phases, and systematically added throughout each distinct phase where a final winner is declared at the end of the final phase. To address the research question of whether modern LLMs can assist developers in efficiently identifying the root causes of build failures, interviews were conducted with senior and experienced developers at the case company. The findings indicate that LLMs, when optimized through parameter adjustments, possess the capability to effectively analyze and resolve build log errors. However, further fine-tuning may be necessary to reduce hallucinations and improve the clarity and formatting of the generated responses.

**Keywords**: Large language models (LLMs), LLM parameters, Elo rating, Prompt engineering, Artificial intelligence, CI/CD pipelines, Build logs, LLM Fine-tuning, LLM evaluation.

**Abstract**

En väsentlig och tidskrävande del av mjukvaruutveckling är att diagnostisera och förstå fel. Även om det finns verktyg som syftar till att effektivisera denna process, är deras effektivitet ofta beroende av fördefinierade felmönster. Vissa verktyg förlitar sig till exempel på att användare identifierar och definierar fel, manuellt konfigurera verktyget, för att verktyget ska senare kunna känna igen felet och hjälpa till med samma problem när de återkommer. [1]

Denna uppsats undersöker om stora språkmodeller (LLM:er) kan användas av utvecklare för att läsa byggloggar och identifiera de delar av loggen som är direkt kopplade till orsaken till fel. Studien utvärderar LLM:er från olika leverantörer och utforskar hur promptdesign, filtreringstekniker och temperaturinställningar påverkar modellernas prestanda. Målet är att fastställa huruvida dessa parametrar påverkar LLM:ers effektivitet i att diagnostisera misslyckade byggen.

Utvärderingen genomfördes med ett en-mot-en-format där två LLM:er med specifika parameterinställningar ställs inför samma problem – att lösa ett misslyckad bygge. Modellerna tilldelas Elo-betyg som uppdateras under matchernas gång beroende på resultaten.

Modellvarianter implementerades under fyra olika faser och lades systematiskt till i varje fas, där en slutlig vinnare utses i den avslutande fasen. För att besvara forskningsfrågan om huruvida moderna LLM:er kan hjälpa utvecklare att effektivt identifiera rotorsaker till byggfel, genomfördes intervjuer med seniora och erfarna utvecklare på fallföretaget. Resultaten visar att LLM:er, när de optimeras genom justering av parametrar, har förmågan att effektivt analysera och lösa fel i byggloggar. Dock kan ytterligare finjustering vara nödvändig för att minska hallucinationer och förbättra tydligheten och formateringen i de genererade svaren.

**Nyckelord**: Large language models (LLMs), LLM parametrar, Elo rating, Prompt engineering, Artificiell intelligens, CI/CD pipelines, Byggloggar, Finjustering av LLM:er, LLM evaluering.

# Contents

1 Introduction 3
1.1 Introduction . . . 3
1.2 Research questions . . . 4
1.3 Purpose with thesis . . . 4
1.4 Goals with thesis . . . 4
1.5 Aim of the thesis . . . 5
1.6 Limitations of work . . . 5
1.7 Division of labor . . . 5

2 Background 6
2.1 The need for LLM-read logs . . . 7
2.2 Evaluation setup . . . 7
2.2.1 LLM arena . . . 7
2.2.2 Build logs . . . 8
2.3 Technical background . . . 8
2.3.1 Large language models . . . 8
2.3.2 Jenkins . . . 11
2.3.3 Elasticsearch . . . 11
2.3.4 Docker . . . 11
2.3.5 VM . . . 12
2.3.6 Ansible . . . 12

3 Related work 13
3.1 Earlier attempt . . . 13
3.2 Chatbot Arena . . . 13
3.3 Enhancing DevOps efficiency through AI-driven predictive models . . . 14
3.4 STRESSED . . . 14

4 Method 15
4.1 Log fetch and programs used . . . 15
4.2 Elo rating system . . . 16
4.2.1 Elo algorithm . . . 16
4.2.2 Continuous measurement . . . 17

4.3 Context window . . . 17
4.4 Evaluation . . . 18
4.4.1 Evaluation set up and criteria . . . 18
4.4.2 Data collection . . . 20
4.5 Interviews . . . 22
4.6 Prompt Engineering . . . 23
4.7 Server . . . 24
4.8 Testing . . . 25
4.9 User feedback . . . 25
4.10 Source criticism . . . 26
4.10.1 Peer reviewed articles: [4], [5], [6], [8], [18], [21], [23] . . . 26
4.10.2 Open-access platforms: [2], [3], [7], [19] . . . 26
4.10.3 Technical documentation: [1], [14], [15], [17] . . . 27
4.10.4 Online articles: [9], [10], [11], [12], [16], [24] . . . 27
4.10.5 Books: [13], [20], [22] . . . 27
4.10.6 Summary . . . 27
4.10.7 AI tools . . . 27

**5 Result** **28**
5.1 Phase results . . . 29
5.1.1 Phase one . . . 29
5.1.2 Phase two . . . 29
5.1.3 Phase three . . . 30
5.1.4 Phase four . . . 31
5.1.5 Interview . . . 31

**6 Discussion** **32**
6.1 Program improvements . . . 32
6.2 Phase discussion . . . 33
6.2.1 Phase one . . . 33
6.2.2 Phase two . . . 33
6.2.3 Phase three . . . 34
6.2.4 Phase four . . . 34
6.2.5 Interview . . . 34

**7 Conclusion** **35**
7.1 Conclusion: . . . 35
7.2 Future work: . . . 36
7.3 Ethical reflection: . . . 36

**8 Appendix** **39**

# Chapter 1

# Introduction

This thesis presents an evaluation of large language models. The work was conducted at Axis Communications AB. The main focus is to investigate whether LLMs can be adapted—without additional training—to serve as effective tools for analyzing failed build logs.

## 1.1 Introduction

The rise of large language models (LLMs) in recent years has incentivized developers to use these models to implement practical tools for their work, such as tools for debugging code to save development time. The proposed work was to find a trained LLM that could handle complicated build script failure logs and give practical instructions to the developer on how to solve a given error or a set of mistakes. At the time of writing this thesis, there are not many tools to optimize the process of debugging failed build logs, in contrast to application code errors, such as the error handler present in most integrated development environments (IDEs).

The technical challenge of the work is how to evaluate these trained models, how to test them against each other, what sections of the build log should be used, how to measure performance, and whether the model can give helpful instructions to the developer. The concept of creating an AI battleground program was adopted, which is a proven benchmarking technique used to evaluate the performance of different large language models (LLMs) based on their ability to solve predefined tasks, guided by human preference. This methodology is partly inspired by the work of Frick et al., who employed a similar approach to evaluate reward models using human feedback [2] as well as by the creators of Chatbot Arena, an open platform for head-to-head LLM comparisons [3]. In this study, this framework is applied to the specific context of resolving build log failure errors.

Engineers at the case company were asked to evaluate the LLMs' ability to solve problems and provide useful responses. In each match, an engineer voted on which model produced the most helpful answer. At the end of four phases for evaluation,

interviews were held to assess how well the winning model performed, and if it could do the task.

To track the LLMs' progress, the Elo rating system was proposed due to its simplicity and long-standing use since the 1960s to rate chess players, as stated in a paper by D. G. de Pinho Zanco, L. Szczecinski, E. V. Kuhn, and R. Seara [4]. Each model was assigned an initial Elo score, and after every match, the ratings were updated based on the outcome, allowing for continuous performance tracking across evaluation rounds.

## 1.2 Research questions

This thesis aims to answer the following questions:

1. Are modern LLMs sufficiently developed to interpret messages from build logs after failed attempts to build application code?
2. What criteria are needed when choosing LLMs?
3. How can a comparative analysis be performed on different LLMs?

## 1.3 Purpose with thesis

The purpose of this thesis is to investigate if different LLMs can be used to analyze build logs from CI/CD-build tests. It is also meant to investigate the possibility of a *Jenkins* integrated AI tool that identifies the root cause of a build log failure. This is interesting since it would save developers time spent on debugging. This, in turn, would free up developers' time for other tasks and enable companies to reduce costs spent on debugging.

## 1.4 Goals with thesis

During the course of the thesis, a comparative analysis will be done to assess LLMs and their respective parameters 2.2.1 in order to assess the first research question. A LLM should be able to automatically identify errors in the build log messages and suggest solutions for the application code with high accuracy. The LLMs will be evaluated based on which one is best suited for the task. This will be done by creating an LLM arena, examining the strengths and the limitations of each model variant as well as comparing each LLM performance to each other. The thesis report will give reasons for the choice of LLM best suited for the intended task. There will be recommendations for the future development of the prototype and for the research.

## 1.5 Aim of the thesis

The thesis aims to test and evaluate different LLMs based on their parameters 2.2.1 and whether they can automatically identify errors in build logs. Since the use of LLMs for build log analysis is a relatively unexplored area with limited academic literature, this project offers an opportunity to contribute new knowledge. For the company, this has the potential to save developers valuable time that would otherwise be spent debugging code. This time can instead be used for more productive work, such as developing new and useful software, which can ultimately reduce development costs significantly. The choice of LLMs will be documented, tested and recommendations for future development will be included. The project has the opportunity to accelerate software development that contributes to society's benefit.

## 1.6 Limitations of work

This work is limited to using the large language models that are used and cleared at Axis. This is because the data used for evaluations has business secrets that could be revealed by using unsafe models.

For the evaluations, there are only a small number of engineers that are trained in reading and solving the logs that are being used. This, together with the fact that evaluations are being done when they have spare time at work, limits the amount of data that can be collected.

## 1.7 Division of labor

The division of labor for this project was equally divided between both thesis workers. This distribution was applied to all aspects of the work, including coding, writing, and other collaborative tasks.

# Chapter 2

# Background

Continuous Integration and Continuous Delivery/Deployment (CI/CD) refers to the automated processes that build code and run tests to ensure that changes likely to cause errors are not pushed to the code-base. These processes facilitate collaboration among developers working within a shared code-base by providing immediate feedback on code changes.

When a CI process gets an error, the developer who tried to upload the code is notified. One common problem is that build- and test logs created by the error can be very large, in some cases up to several thousand lines. This makes it hard for the developer to find what caused the problem, which in turn makes it time-consuming to find what caused the failure.

At Axis they use Jenkins, a popular open-source tool for CI/CD. Jenkins has a plugin that's called Build Fail Analyzer (BFA, for further information https://plugins.jenkins.io/build-failure-analyzer/). The BFA analyzes the output/build log that is created when the tests from Jenkins detect an error and identifies known errors. BFA has some limits. It needs manual configuration for updates of new errors, and it is limited to known error patterns.

A possible solution to the problem is to use large language models to simplify logs from CI builds & tests, potentially giving the developer clear instructions on what to correct to make their code work.

The technical challenges are how to conduct the testing and which Large Language Model (LLM) is best suited to solve this type of problem. One concern with LLMs is that secret information will leak, or that information that gets received is trademarked. To prevent this, both agreements with LLM suppliers can be made to get control of what data the LLM is using. And the LLMs can be run from local servers, which prevents data from leaking.

This thesis is draws inspiration from an earlier master thesis [5], where the authors examined if it's possible to identify errors with machine learning from failed build log messages.

The cooperating company is Axis Communications. They operate in many dif-

ferent areas, including security cameras, network solutions, intercom, and much more. The thesis is done at R&D Tools. R&D Tools work on the tools that other developers in the company use; the aim is to make it easier and more efficient for developers to work.

## 2.1 The need for LLM-read logs

A fairly new way of developing software has occurred with the rise of generative AI. Today, tools have become more frequent in workplaces where they can help with code reviews and test generation. Some companies have used deep learning to train models on data they have to make predictions about timelines for a project, tests that need to be done, and to perform code reviews. This has been done with success in reducing time to completion in projects. [6]

The authors of [6] did experiments that provided results that strengthen further implementation of deep learning in projects. They also discuss the fact that a large amount of data is needed for training purposes, which comes with a cost of large computational resources needed. It is also discussed that deep learning has some limitations with not being generalized.

LLMs that are available today could be the solution to handle these more general problems as build logs are. It is not as standard in a build log that errors occur in the same place or of the same kind. And, as mentioned, it is not uncommon that build logs from CI/CD pipelines exceed a thousand lines.

## 2.2 Evaluation setup

This section will introduce what kind of format has been used for the evaluation in the method. It also describes what the evaluation has been done on.

### 2.2.1 LLM arena

To evaluate different LLMs, inspiration was drawn from Chatbot Arena, a benchmark that assesses models based on human preference. In this approach, preferences are determined by the quality of answers the LLMs generate in response to specific questions and prompts. These models are then pitted against each other, with humans acting as judges to determine which model's response is superior [3].

In contrast, other evaluation methods are grounded in a "ground truth" approach, where the model's task is to identify a single correct answer to a problem, typically derived from a static or live dataset.

In the evaluation setup, the goal extends beyond merely identifying the correct answer. The aim is to provide a concise, detailed explanation of why a build failed, including clear instructions on how to resolve the issue. To ensure the models'

effectiveness in a real-world scenario, an unmodified live dataset sourced directly from Jenkins servers is used for testing.

### 2.2.2 Build logs

There are many types of logs that can be parsed and analyzed, each offering different levels of complexity and insight. During the project, two kinds of Jenkins logs were used. Both types consist of files with hundreds and several thousand rows. The two log types differ by source. The first includes a wide range of software and firmware builds on products, providing a diverse set of build information. In contrast, the other type of logs is specific to the Axis internal system build process and is significantly more complex. This complexity stems from the large number of services involved in building the systems that Axis uses, making these logs denser and more difficult to interpret. Consequently, when a build failure occurs, identifying the root cause in the other type of logs can be far more time-consuming and challenging for developers.

## 2.3 Technical background

The following has been used. The parts described here have either been part of setting up the program that was used for making the evaluation. Or it has been the main part that the evaluations have been done on.

### 2.3.1 Large language models

A Large Language Model (LLM) is an artificial intelligence system trained on vast amounts of text data to understand and generate human-like language. Using deep learning techniques to better understand text patterns, it can process and produce coherent, context-aware responses for tasks such as translation, summarization, and conversation [7].

An LLM takes a prompt as a query to generate an answer. Using a different setup for the prompt to the LLM can be used to fine-tune answers and make the user request more understandable for the LLM. This can also be used to make the LLM answer in a more compelling way to the user [7].

A prompt can be defined by [8]:

> ”The practice of designing, refining, and implementing prompts or instructions that guide the output of LLMs to help in various tasks. It is essentially the practice of effectively interacting with AI systems to optimize their benefits.”

In machine learning and, when talking about LLMs. A parameter is a form of data that is used in training a LLM. A parameter is used by the LLM to make predictions of what should be answered to a prompt (for more on LLM training, see [9] and [10]). LLMs today are trained on a lot of parameters, some exceeding billions.

There are different setups that can be made with a query. **Single-Turn Instructions** is when a query is sent one time to the LLM and the query is all the information about the task. This is used with *Zero-Shot Prompts* and *Few-Shot Prompts*. **Multi-Turn Instructions** is used for agent-like LLM responses (the LLM gets a chance to reason about the question). A loop is used where responses and feedback are used up to several rounds. [7].

Other prompt setup options determine how the LLM receives and processes information. A **Zero-Shot Prompt** involves providing the LLM with a task and all relevant information, but without any examples of how to solve it.

Another approach is **In-Context Learning**, where the LLM is given examples or demonstrations within the prompt to guide its response. This method is often referred to as *Few-Shot Prompting*, as it allows the inclusion of several examples to help the model generate the desired output [7].

For further reading on how to make prompts more efficient, see the GitHub page from [11].

In machine learning, and when talking about LLMs, a parameter is a form of data that is used in training a LLM. A parameter is used by the LLM to make predictions of what should be answered to a prompt (for more on LLM training, see [9] and [10]). LLMs today are trained on a lot of parameters, some exceeding billions.

The effectiveness of LLMs for error interpretation in build logs is examined using the parameters described below.

1. Build log input
2. Model brand
3. Prompted question
4. Temperature

The list is ordered by initial thoughts of how big an impact it would have on the answer, and the difference in quality of the answers for the engineers that vote on it.

Build log input:
With build log input, adjustments were made on which segment of the log is sent to the LLM. This can be, for instance, the last sixty thousand characters of the log, the whole log or some variation where parts of the log are taken and combined as input.

Model brand:
The number of model brands available for use was limited to those internally accessible at Axis: **Llama 71B**, **Mistral-large**, **Codestral**, and later **Gemini** (accessed after the initial testing phases). Each of these models brings different characteristics relevant to the evaluation:

- **Llama 3 71B** is a large-scale general-purpose language model developed by Meta, known for its strong performance across a wide range of natural language understanding and generation tasks.
- **Mistral-large 25.02** is an open-weight model designed for fast inference and strong general reasoning capabilities, often used as a lightweight yet competitive alternative to larger proprietary models.
- **Codestral 25.01** is a code-oriented model optimized for software engineering tasks, such as debugging, code synthesis, and technical log analysis—making it particularly well-suited for build log use cases.
- **Gemini 2.0: Flash**, developed by Google, is a proprietary multimodal model with strong general reasoning and problem-solving abilities. It became available for internal use only in later stages of testing.

These models were evaluated using different prompt formulations designed to guide them in solving build log errors. Additionally, variations in the input (e.g., different segments of the log) and temperature settings were experimented with to assess their influence on the quality and consistency of the outputs.

Prompt:
This is the instruction for what the LLM is asked to solve, and how the LLM is asked to present the answer. This, together with the build log input, makes the prompt sent to the LLM. The prompt could, for example, be in the form of:

"Find the reason for failure. Ignore warnings. Present an answer with bullet points, use headlines: Reason for failure and solution."

Temperature:
The temperature is a measurement of the creativity given to the LLM. It is a scale

from 0.0 to 1.0. When a lower temperature is chosen, the creativity of the LLM is lower. This means that the tokens chosen have a higher probability of following the previous one. Higher creativity, closer to or 1.0, gives the LMM more flexibility to generate answers that are more random. Lower temperatures are recommended for tasks that require more precision, while higher temperatures are recommended for tasks like creative writing. [12]

### 2.3.2 Jenkins

Jenkins is an open-source automation tool that is used for continuous integration and continuous delivery (CI/CD). It helps developers automate the process of building, testing, and deploying software, making it easier to manage and streamline development workflows. Jenkins supports a wide range of plugins and integrates with many tools to enhance software development efficiency. [13] The setup involved accessing Jenkins servers used by Axis to collect build logs, via URL, from various failed jobs. These logs formed the basis of the dataset used to test the LLMs.

### 2.3.3 Elasticsearch

Elasticsearch is an open-source, distributed search and analytics engine renowned for its scalability, real-time search capabilities, and ability to manage large data volumes. Built on Apache Lucene, it provides advanced full-text search features such as tokenization, streaming, and scoring. Elasticsearch interacts through a RESTful API using JSON. Elasticsearch was utilized to retrieve real-time logs from Axis Jenkins servers. This integration allowed for indexing and querying the URL of build failure logs efficiently and to supply these URLs as input to fetch the logs using Jenkins. [14]

### 2.3.4 Docker

Docker is an open-source platform that enables developers to package applications and their dependencies into lightweight portable containers, ensuring consistency across various environments. By isolating applications from the underlying system, Docker simplifies deployment, scaling, and management in development, testing, and production. Docker containers can run on any machine that supports Docker, streamlining the process of managing applications. Docker was used to containerize the LLM arena application, which orchestrates head-to-head comparisons of model outputs. This container was deployed on a virtual machine to simulate real-world conditions and ensure reproducibility in the evaluation. [15]

### 2.3.5 VM

A virtual machine (VM) is a software-based replica of a physical computer that runs its own operating system and applications, just like a real machine. This technique enables multiple isolated environments to run on a single physical system, with each VM having dedicated resources such as CPU, memory, and storage. A VM was used as the hosting environment for the containerized LLM arena application. This VM acted as the central node where all evaluations were conducted, and it provided a controlled, isolated environment to avoid interference from external factors during testing. [16]

### 2.3.6 Ansible

Ansible is an open source, agent-less automation tool that simplifies the management and configuration of systems. It uses SSH (Secure Shell) for communication, eliminating the need for additional agents to be installed on remote systems. Ansible was utilized to automate the provisioning of the virtual machine, including the installation of required dependencies such as Docker, Python packages, and configuration files. This ensured that the evaluation environment was consistently and reliably prepared for running the LLM comparisons, minimizing manual setup errors and promoting repeatability. [17]

# Chapter 3

# Related work

The thesis has taken inspiration from different sources. These have provided ideas on what can be compared and what could be used for the experiment.

## 3.1 Earlier attempt

This thesis is based on an earlier master's thesis. In it, the students used machine learning to build a model that read build logs and suggested solutions to errors in the logs [5]. Their results concluded that their model was good at observing and solving known errors, which the model had learned from the training set, but had trouble solving previously unseen errors that diverged from the training set. For the training set, they used a library of build logs with errors and known solutions that engineers had previously identified.

## 3.2 Chatbot Arena

The testing environment was designed using the arena format inspired by Chatbot Arena, an AI benchmarking tool that evaluates large language models (LLMs) based on human preferences. The platform is maintained by researchers at UC Berkeley, SkyLab, and LMArena. The benchmark pits two LLMs against each other in response to a prompt, allowing users to compare their outputs side by side. Prompts are open-ended and not standardized, covering a broad range of general questions. Users are then directed to a leaderboard displaying various LLMs and their performance scores in relation to the prompt.

## 3.3 Enhancing DevOps efficiency through AI-driven predictive models

In a study conducted by Aliyu Enemosah at the University of Liverpool, AI-driven predictive models were explored as a means to optimize build systems within Continuous Integration and Continuous Deployment (CI/CD) workflows. The paper discusses a range of applications for these models, including build optimization, anomaly detection, and other related processes. While the study provides a broad overview of how predictive models can enhance various stages of the CI/CD pipeline, the work specifically focuses on the debugging of failed build logs. This study investigates how large language models (LLMs) can serve as a tool for individual developers to identify and resolve build issues more effectively during development and deployment. [18]

## 3.4 STRESSED

STRESSED (A Security Log Analysis System) is an AI-driven security log analyzer designed to identify potential threats within web server traffic. Developed by the .txt team, it leverages structured language generation to improve the accuracy and reliability of its analysis. At its core, STRESSED uses the Outlines library—also developed by the .txt team—which enables constrained language generation, a technique that directs large language models (LLMs) to produce outputs that conform to a predefined structure or schema. This constraint not only enhances predictability but also simplifies parsing and validation, making it especially valuable in security-critical contexts. [19]

The core methodology behind STRESSED shares similarities with the thesis project, particularly in the goal of identifying anomalies or errors through log analysis. However, while STRESSED focuses on detecting security issues in web server logs using fixed models, the approach is centered on diagnosing build failures in CI/CD pipelines. Specifically, the aim is to evaluate and fine-tune various LLMs by adjusting parameters such as context window size, prompt design, and temperature settings, in order to isolate and highlight the root causes of build errors and finally find an LLM that could be further implemented into a tool to help developers debug their failed builds faster and more effectively.

# Chapter 4

# Method

The work has been done agile and iterative. During the work progress, new parts in the gui did change after feedback from the Axis engineers, and when a new model became accessible, it was made an implementation to fit it for evaluation.

The phases that have been done for this work: the research phase has been done from the beginning and lasted 4/5 of the time, development of the program was done in the middle 4/6 of the time, work on the paper has been done throughout the time of work, and data collection including interviews was done in the last 2/5 of the work.

The literature study was done most intensely during the beginning of the work, most used platforms for search were: https://scholar.google.com/, https://ieeexplore.ieee.org/, and https://www.google.com/.

## 4.1 Log fetch and programs used

This thesis has used a mix of methods to get a result. The following will explain what has been done, and what mechanisms have been used for that result.

To begin with, the logs that have been the goal to make evaluations on. These logs were first fetched by URL with *Elasticsearch* query; the program that was made was for the most part already existing in the collaborative team (for the interested reader, see https://www.elastic.co/docs/reference/query-languages/query-dsl/query-dsl-query-string-query). The URLs were selected from the past one week, for relevance.

When the URL was fetched, they were used to fetch *Jenkins-logs*. This as well was done with help provided from the team; an API key was used to access *Jenkins* (for the interested reader, see https://www.jenkins.io/doc/book/using/remote-access-api/).

In Figure 8.1 in the Appendix, the different programs made and used for this work are laid out. They all run separately where a top-down use of each would be: *URL fetch*, *Log fetch*, *LLM answer* which sends the Jenkins logs to Axis AI *Prisma*,

and before the *Evaluation* is started, the database used in evaluation is prepared using *Manual DB editor*.

## 4.2 Elo rating system

To be able to conclude if a model is better than another, the Elo score is used. It is done in a setup where the score of the models competing is re-evaluated after each match. More in details are explained in this section.

### 4.2.1 Elo algorithm

The tournament rating format that was decided on is based on the Elo rating system inspired by chess. The Elo rating system is used to calculate the relative skill of two players in a two-player game format. The rating of each model is done by giving them an Elo score, where the default score is 1200 and it is then recalculated and updated after each match. The default of 1200 is selected by being border between amateurs (still ranked over 1200) and novices (under 1200) at the time Elo wrote his book [20]. The default value doesn't have to be this, but can be arbitrarily selected.

Elo scores are typically initialized with a sufficiently high starting value to prevent players from receiving negative scores during competitions [20]. Since the total number of matches to be played throughout the project was uncertain, the decision was made to choose a higher initial Elo score to ensure all ratings remained positive.

If a higher-rated model wins, the change in its Elo rating is relatively small, whereas a lower-rated model winning leads to a larger rating adjustment. The Elo rating system employs a mathematical formula to update a player's—or in this case, a model's—rating based on the expected probability of winning and the actual match outcome.

The expected score $E$ for a player with rating $R_{\text{old}}$ against an opponent with rating $R_{\text{opponent}}$ is given by:

$$E = \frac{1}{1 + 10^{\frac{R_{\text{opponent}} - R_{\text{old}}}{400}}}$$

The updated rating $R_{\text{new}}$ is then calculated as:

$$R_{\text{new}} = R_{\text{old}} + K \times (S - E)$$

where $S$ is the actual score of the match (1 for a win, 0 for a loss, and 0.5 for a draw), and $K$ is a constant determining the maximum possible rating change per match. This approach ensures that the ratings dynamically and accurately reflect a model's relative problem-solving ability over time.

The K value in the formula is a coefficient that can be adjusted based on the desired amount of Elo points a player can gain or lose depending on the outcome of the match. In chess, the K value varies depending on the skill level of the player, such that higher-rated players have a lower K value compared to new players. Due to the amount of model variants that were used being much fewer compared to the world of chess players, the Elo rating changes are less pronounced in comparison, and models are not expected to achieve very high Elo scores. The models' abilities vary in solving their assigned tasks, and the pool is more volatile relative to skill levels. Therefore, the United States Chess Federation (USCF) standard $K$-factor value of 32, which is typically applied to players with ratings below 2100 Elo. Due to the limited duration of the testing period and the restricted number of qualified evaluators available to assess the models' responses, the models is not expected to surpass this threshold.

### 4.2.2 Continuous measurement

The models compete against each other continuously, which is why the Elo rating system is chosen. It is predicted that, over time and with enough matches, the models' ratings will diverge, allowing a clear winner to emerge after several weeks of competing [20]. This continuous measurement system is ideal for evaluating the models in real time. Throughout the testing period, new model variants will be able to dynamically be introduced into the battleground and be observed how their ratings evolve in comparison to the existing competition.

## 4.3 Context window

The context window of an LLM refers to the number of tokens (with 1 token being roughly 4 characters) that a model can process at a time. For most of the models evaluated, this limit was approximately 32,000 tokens. In contrast, a medium-sized build log can easily reach 4.6 MB in size—equivalent to about 1,615,171 tokens—which vastly exceeds the input capacity of these models.

To address this significant mismatch between build log size and model context limitations, several log parsing strategies designed to reduce the input to a manageable size while preserving relevant information were implemented. These methods made it possible to evaluate model performance meaningfully on real-world logs without exceeding the models' processing limits.

The following strategies were applied to parse and filter the build log:

- End-Slice Retrieval: Retrieve the last portion of the build failure log based on the model's maximum token capacity. For most models, this corresponds to the last 32,000 tokens, as the most relevant failure information is typically

found near the end of the log. For models with larger input windows—such as Gemini, which supports up to 100,000 tokens—a longer segment can be retrieved to enable deeper context analysis.

- Chunk method: The chunk method is an algorithm which was applied to the build failure log by chopping the log into smaller pieces of around 30,000 tokens each and storing them in a vector; each piece was then sent to the same model for analyzing where each answer was then stored into a new vector which held each answer. Then finally, the vector would be sent with all the stored answers for a final analysis by the model. The maximum amount of tokens with which a model could respond was set to a maximum of 512 tokens, so that the final answer vector would not cap out the models context window. By applying the chunk method, the model could in fact iterate through the whole build log without having to process the whole log at once.

- Half-Chunk method: This method refines the chunking approach by focusing exclusively on the second half of the build failure log. The decision to exclude the first half is based on the observation that it often contains verbose setup information that is typically not related to the actual failure. By limiting analysis to the latter part, where critical error messages and relevant context are more likely to appear, it significantly reduce processing time and resource usage while maintaining the accuracy and relevance of the insights generated.

## 4.4 Evaluation

The following is a description of what was asked to be evaluated, and how the evaluation program was used.

### 4.4.1 Evaluation set up and criteria

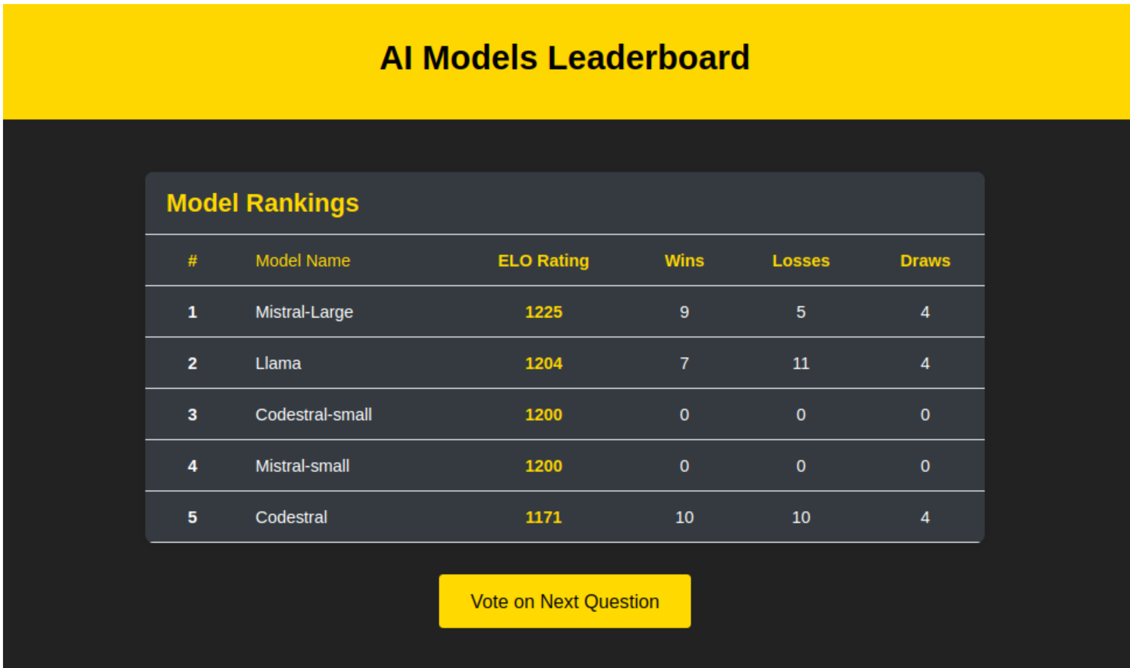
A website was developed to run on Axis's internal Wi-Fi network, facilitating the evaluation of different LLM versions. The homepage featured a leaderboard displaying each LLM's name, score, number of wins, losses, and draws. From this page, users could navigate to the voting page via a dedicated button.

The voting page introduced a battleground format, where two LLM versions competed head-to-head in a 1v1 setup (see Figure 4.3). A build failure log containing errors was shown at the top, followed by responses generated by two different LLMs. Below each response, there was a vote button allowing users to choose the more helpful answer. A "Draw" button was also available for cases where both responses were equally useful in identifying the errors. Additionally, a "Skip" button allowed users to load a new log and corresponding LLM responses in case the user did not understand the log or was unsatisfied with both LLM responses.

Below the LLM answers, a text box provided an introduction to the project along with a set of evaluation criteria to guide users when casting their votes. The following list was asked to be prioritized with 1 as the most important factor and 3 as the least important when voting:

1. To evaluate the correctness of the answers given.
    - Does the answer identify the error in the log?
    - Does the answer provide a solution to the error?
2. To evaluate the helpfulness of the answers given.
    - Is the answer easy to understand?
    - Is the answer easy to follow?
    - Is the answer complete?
3. To evaluate the overall quality of the answers given.
    - Is the answer well structured?
    - Is the answer well written?
    - Is the answer well formatted?

Moreover, a *leaderboard page* was implemented (see Figure 4.1), providing a clear visual overview of each model's performance over time. This addition not only helped monitor results internally but also gave users a sense of which models were most effective based on collective voting outcomes. To reduce bias in evaluations, the identities of the competing models were concealed during each match. This ensured that users assessed outputs based solely on content quality, rather than preconceived notions of a model's capabilities or brand.


AI Models Leaderboard

Model Rankings

| # | Model Name | ELO Rating | Wins | Losses | Draws |
|---|---|---|---|---|---|
| 1 | Mistral-Large | 1225 | 9 | 5 | 4 |
| 2 | Llama | 1204 | 7 | 11 | 4 |
| 3 | Codestral-small | 1200 | 0 | 0 | 0 |
| 4 | Mistral-small | 1200 | 0 | 0 | 0 |
| 5 | Codestral | 1171 | 10 | 10 | 4 |

Vote on Next Question

Figure 4.1: LLM leaderboard

### 4.4.2 Data collection

This section describes the approach used to collect data, which was conducted in four distinct phases. To effectively evaluate and compare different large language models (LLMs), the process began by testing models from various providers.

Next, the prompts sent to each LLM were adjusted to influence their responses toward specific objectives. In addition, temperature settings were varied across different LLMs to assess whether the level of randomness (or creativity) influenced their ability to address complex problems. Further experiments were conducted to assess whether an LLM requires access to the full build log or can perform adequately with only partial information.

Each phase includes the variants with the targeted changes, and the best-performing variants from each phase were carried forward to the next. The new models were given the default value of 1200. This phased approach aimed to identify the optimal configuration for task performance. Not all model variants were included in every phase; this selective inclusion ensured that only relevant data points were collected, preventing weaker models from diluting the overall results.

#### Phase one

In Phase One, the performance of different model brands was evaluated. This included two general-purpose models trained on diverse internet data, Llama and Mistral-large, and one code-oriented model, Codestral, which was primarily trained on programming-related data. To conduct this evaluation, the End-Slice Retrieval method was applied and the context filtering strategies described in Section 4.2 for

preparing build log inputs.

The first strategy used a Single-Turn Instructions/Zero-Shot Prompt approach [7], where only the final portion of the build log was included together with the prompt. The maximum token capacity of each model determined how much of the log could be included.

The second strategy followed a Multi-Turn Instructions/In-Context Learning approach [7], in which the build log was divided into chunks—up to the model's token limit—and processed sequentially from top to bottom. The response from each chunk was stored in a vector, which was then sent back to the model for summarization into a final answer.

All model variants used the same prompt and temperature (see Table 8.1).

### Phase two

This phase focused on experimenting with different prompts. The Elo rating was reset and a new database was created. The two top-ranked brands from phase one were kept as base models for different variants. These four base models were kept as they were (two chunk and two Single-Turn/Zero-Shot) in phase two. Two new prompts were introduced in phase two (see Table 8.1, prompt 2 & 3). These two new prompts were used on each of the four base models to make eight new model variants. The reflection and fact-check list prompt patterns, derived from the work of White et al. [21], were the two primary strategies employed in the design of the prompts for phase two. These patterns are aimed at improving the model's reasoning and accuracy by encouraging iterative review and verification of generated responses.

### Phase three

In this phase, the two top-performing model variants from Phase Two were used to investigate the effect of temperature on model behavior. To capture a broad contrast in performance, two temperature settings were selected: one at the lower extreme (0.1) and one at the higher extreme (0.9) [12]. Each model variant was tested at both levels to assess how temperature variation influences output quality and consistency.

### Phase four

In the final phase, the investigation focused on whether changes to the prompt or adjustments to the amount of input from the build log would lead to improved responses from the LLMs. A new prompt was designed based on user feedback, specifically from engineers in the co-op team, who highlighted elements they felt were missing in the LLM-generated answers. Based on their input, revised prompt was developed (see Table 8.1, prompt 4) tailored to better address their needs.

To improve handling of the build log input, modifications were implemented to the chunking strategy by applying the Half-Chunk method (see Section 4.3). This approach focuses exclusively on the second half of the build failure log, based on feedback from engineers who noted that the initial portion typically contains CI/CD setup information that is rarely useful for diagnosing failures. By using only the latter half—where critical errors and relevant context are more likely to appear. The goal was to enhance the relevance of the input while reducing processing time and resource usage.

## 4.5 Interviews

Interviews were conducted at the end of phase four. Three engineers were selected; they were selected from the team that which this thesis collaborated. The basis for this was that they had experience with most of the logs that were used, and that they did not daily use AI-tools for work. The experience of these engineers varied from experienced- to senior engineer, where senior is the highest level, and experienced is one step under.

The interview was conducted in the same room as the interviewees, where they were asked to talk loud what they thought, and looked at. The answers to their questions, and the thoughts they had was documented by the interviewers.

The task for the interviewees was to evaluate whether the winning model variant's solving capabilities are effective enough to assist developers in locating the root cause of build failure more quickly and effectively, saving work time. And to estimate if a tool with AI would be helpful.

The interviewees used a variant of the arena program designed with only one log output, and a single solution generated by an LLM and a "*Next Question*" button to iterate to the next problem (see figure 4.2).

During the interviews, eight logs were selected. The engineers were asked to do one log and then answer questions. After the questions, they could continue to the next log and repeat the cycle. The engineers were told that they could skip a log if they do not work with that kind and are not able to assess the correctness.

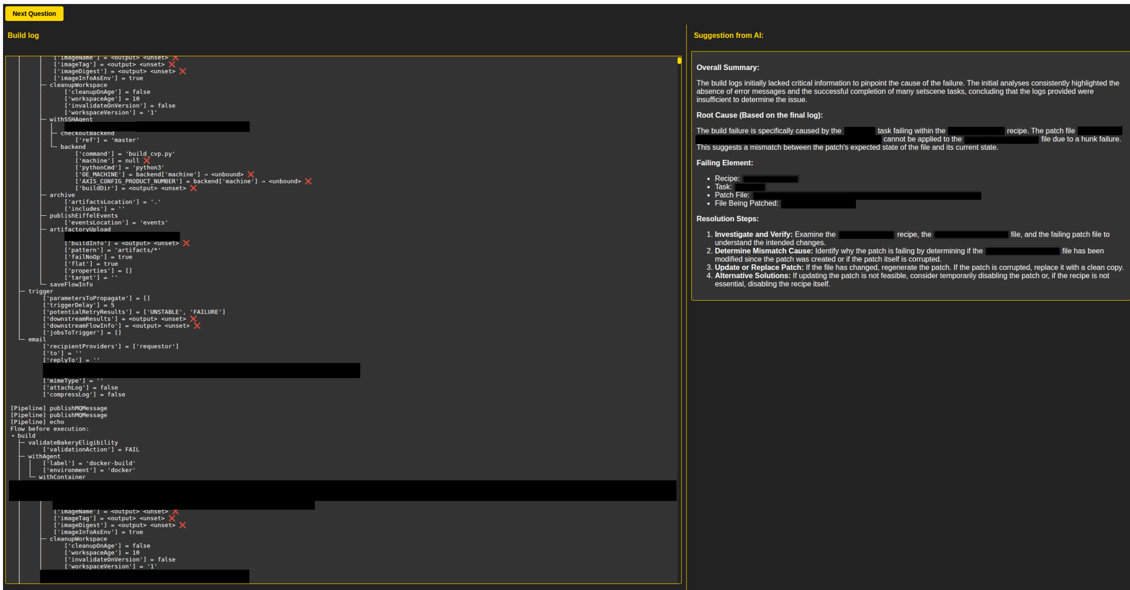



Figure 4.2: Interview program

The following questions were posed during the interview:

1. From a scale from 1 - 5:
   How do you think the models answer performed in identifying the root cause of failure in the log

2. Is the solution easy to read and follow?

   - If so, why?
   - If not, why?

3. Based on your experience using the program and reading the answers, do you think it's worth developing an LLM-based tool for this case?

   - Yes
   - Maybe
   - No

## 4.6 Prompt Engineering

The ability of a large language model (LLM) to generate concise and accurate solutions to build log failures is strongly influenced by how the prompt is formulated. As a starting point for the prompt design, inspiration was drawn from the paper called
*"A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT"* [21], which presents a comprehensive catalog of prompt patterns tailored to address common interaction challenges with LLMs. The paper identifies various problem cate-

gories and proposes corresponding strategies to improve response quality and relevance.

For this purpose, the focus was on the "Error Identification" category, selecting two key patterns: *Fact Check List* and *Reflection*. These were chosen for their potential to help the model better reason through the noisy and often ambiguous nature of build logs, ultimately generating more actionable and reliable responses.

In addition to these structured prompt patterns, a prompt design based on user feedback was incorporated, emphasizing the importance of the model explicitly identifying the specific log line associated with the failure. This insight shaped the development of prompt 4.

These strategies were implemented and evaluated through four prompt variants, as described below and summarized in table 8.2:

- **Prompt 1 (Agency):** The first prompt used as the base prompt. It established the LLM's role as an autonomous agent responsible for analyzing build logs and delivering a concise yet thorough summary of the failure.
- **Prompt 2 (Fact check list):** This pattern directs the model to first identify and list relevant error messages or failure indicators before proposing a solution. By encouraging a systematic breakdown of the log, this approach improves transparency and reduces the risk of overlooking critical information.
- **Prompt 3 (Reflection):** This prompt follows a two-phase structure: the model is first asked to generate a solution, then to reflect on whether the solution fully addresses the identified error. This pattern introduces a layer of self-evaluation, aimed at enhancing completeness and reducing hallucinated or superficial answers.
- **Prompt 4 (Highlight):** Prompt 4 was developed in response to user feedback. It explicitly instructs the model to highlight the specific log line responsible for the failure, thereby improving the clarity and direct usefulness of the output for developers.

The application of these structured prompt designs was intended to systematically guide the LLMs toward generating outputs that are not only informative but also grounded in the actual contents of the build logs, thereby improving both usability and trust in their responses.

## 4.7 Server

A RESTful API was developed using the Python FastAPI framework to efficiently retrieve locally stored log answer data, as well as to facilitate communication between the database and the front end. To validate the voting mechanism, the Pydantic

framework was utilized to ensure proper data transfer from the client to the server, and then to the database. This allowed for the update of each model's Elo score and to initiate a new match. The server was designed to use APIs to access the models. This makes it fully possible for customization, i.e. future models can be tested with this tool. Which has been a major help since the area is changing fast, and new models could be integrated into the testing without any major changes to the core program.

To improve server performance, log answers were pre-loaded into local storage, enabling faster retrieval directly from the hard disk. This optimization allowed the server to display answers more quickly. Additionally, a server initializer was used to handle the necessary API calls to the LLMs, storing each answer at a specific file path. This setup ensured that, once the LLM arena application was launched, answers could be quickly accessed. [22]

## 4.8 Testing

During the development of the program, two main pages were created to collect and display data. The Python Jinja framework was used to display the data by passing build logs and model answers to the front-end code.

The battleground interface involves two fields, one for each LLM's solution to the build failure log, as well as a field for the log itself, allowing users to read through and compare with the LLM's solutions. The vote buttons retrieve the user's vote, which decides the winner and loser of the match. This information is passed on to the server code.

The score of each respective model is updated using the Elo model and the response from the vote button and is reflected in the database. After a match is set, the webpage is then automatically reloaded and a new match is set with a randomly selected build-log and two randomly selected model variants that answer the selected build-log (see Figure 4.3). An additional instructions field was included to inform the engineers on the purpose of the application and to clarify the criteria they should use when evaluating and prioritizing the model responses.

## 4.9 User feedback

Feedback was gathered from the end users during the initial phases of testing, which guided iterative improvements to the battleground application. One of the most requested features was a *skip function*, allowing users to bypass a match without updating either model's Elo score. This was implemented directly after multiple users pointed out that, in cases where both models produced equally unhelpful answers, awarding Elo points even as a draw did not accurately reflect performance.

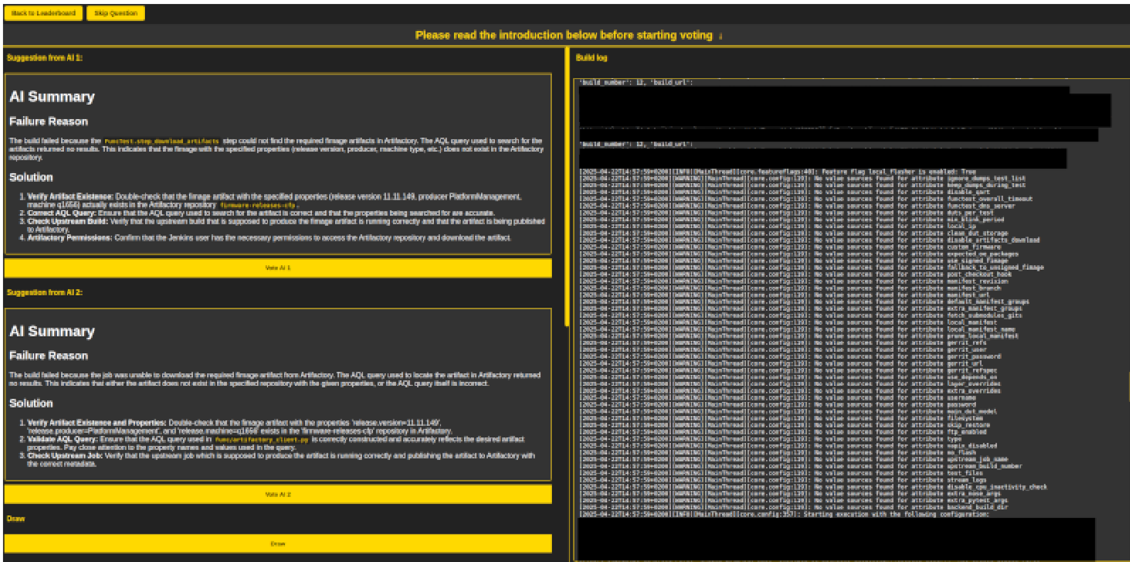



Figure 4.3: Battleground interface

These modifications highlight the significance of continuous user involvement in the development process. Incorporating user feedback enabled iterative refinements to both the interface and the evaluation workflow, thereby enhancing usability with the end user's expectations.

## 4.10 Source criticism

This section presents the sources used in the work, ordered in categories of source type.

### 4.10.1 Peer reviewed articles: [4], [5], [6], [8], [18], [21], [23]

Several sources originate from peer-reviewed publications such as the works on the stochastic analysis of the elo rating algorithm [4], prompt engineering catalog [21], retrieval-augmented generation for NLP tasks [23] and a master thesis which was referenced for prior related work on build log analysis [5]. These sources were used to get an understanding of the research field and what possible future improvements could be implemented into the thesis. These articles are generally trustworthy due to their peer-reviewed publication processes.

### 4.10.2 Open-access platforms: [2], [3], [7], [19]

Some sources such as [2], [3] and [19], originate from open-access platforms such as arXiv. These are up-to-date papers that provide valuable insight into current research and are often cited by other researchers; they have not undergone formal peer review and should therefore be interpreted with some caution.

### 4.10.3 Technical documentation: [1], [14], [15], [17]

Technical documents were used for a deeper understanding of the practical aspects of the research, e.g. the Jenkins Build Failure Analyzer plugin [1]. Although these are not academic sources, they are valuable for the structure of the project.

### 4.10.4 Online articles: [9], [10], [11], [12], [16], [24]

Online articles were used alongside technical documentation to gain a clearer understanding of how the technologies function and how to implement them effectively. They originate from reputable organizations such as IBM and Elastic. One source references a Wikipedia definition, which should be interpreted with caution due to its open-edit nature.

### 4.10.5 Books: [13], [20], [22]

The thesis references three books by established authors, including the creator of the Elo rating system [20]. These books offered valuable insights into the workings of the Elo algorithm, as well as practical knowledge about FastAPI and Jenkins. The Elo rating algorithm, developed by Arpad Elo, has been widely used in the chess world since its introduction in 1978, establishing its credibility over decades of practical application. The books on FastAPI and Jenkins are published by well-regarded technical publishers—Packt and O'Reilly—known for producing high-quality material, supporting their reliability.

### 4.10.6 Summary

In summary, the sources used are varied in type and reliability. Critical evaluation was applied in their selection and application to ensure that the conclusions drawn in this thesis are based on trustworthy and relevant information.

### 4.10.7 AI tools

Except for the AI tools (LLMs) evaluated, the built-in GPT model in Overleaf has been used for spelling correction and grammar suggestions. ChatGPT has also been used for some searches on how to do Docker commands, or give suggestions on SQL statements to fetch data.

# Chapter 5

# Result

The following chapter displays the final results from each testing phase and the results from the first interview question.

The final Elo rating scores of each model variant from the last phase of testing, see table 5.1.

Table 5.1: Model results after final phase

| **Model** | **Score** | **Model** | **Score** |
|---|---|---|---|
| Gemini-Max-half-Chunk-Temp-0.1 | 1284 | Gemini-Max-Temp-0.1 | 1283 |
| Gemini-Max-half-Chunk-Temp-0.9 | 1276 | Mistral-large | 1235 |
| Gemini | 1229 | Mistral-large-Chunk-p4 | 1228 |
| Mistral-large-half-Chunk | 1227 | Mistral-large-Temp-0.1 | 1226 |
| Gemini-Max-Temp-0.9-p4 | 1223 | Mistral-large-Chunk-p3 | 1215 |
| Gemini-Max-Temp-0.9 | 1209 | Gemini-Max-Temp-0.1-p4 | 1206 |
| Mistral-large-p4 | 1196 | Llama-p3 | 1195 |
| Mistral-large-Chunk | 1193 | Llama-Chunk-p3 | 1189 |
| Mistral-large-p3-Temp-0.1 | 1186 | Gemini-Temp-0.1 | 1184 |
| Mistral-large-Chunk-p2 | 1180 | Mistral-large-half-Chunk-p2 | 1175 |
| Llama-p2 | 1174 | Mistral-large-p3 | 1166 |
| Llama | 1162 | Mistral-large-p2 | 1161 |
| Llama-Chunk | 1149 | Llama-Chunk-p2 | 1138 |
| Llama-p2-Temp-0.1 | 1111 | Codestral | 1105 |
| Codestral-Chunk | 1092 | | |

## 5.1 Phase results

The results for each phase are presented below:

### 5.1.1 Phase one

At the end of phase 1, the initial set of model variants had been evaluated. The Codestral variants performed the worst among all models, showing significantly lower effectiveness. In contrast, Llama-Chunk emerged as the overall winner of the phase, followed closely by Mistral-large-Chunk in second place (see Figure 5.1).

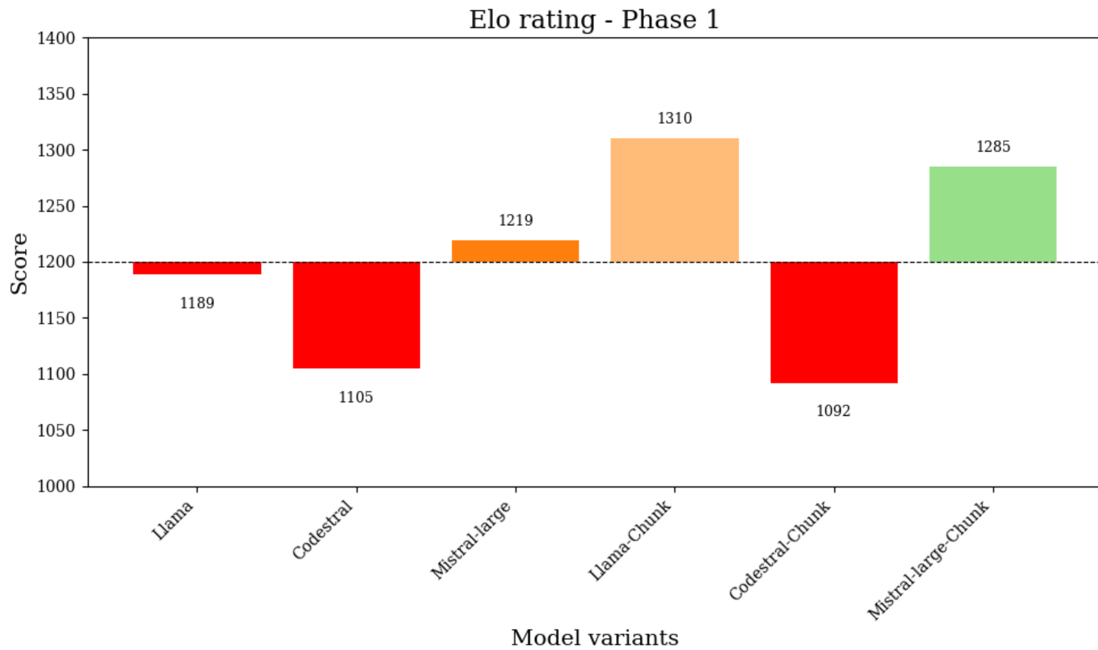



Figure 5.1: Phase one: Results

### 5.1.2 Phase two

During phase two, the performance of the model variants was monitored over time as new versions were introduced and additional evaluation rounds were conducted. Codestral was excluded from this phase due to its exceptionally poor performance in phase one.

All models began with a baseline score of 1200. Over the course of the phase, Mistral-large consistently outperformed the other models, maintaining the top position throughout. It was followed by Mistral-large-p3 in second place and Mistral-large-Chunk-p2 in third (see Figure 5.2).

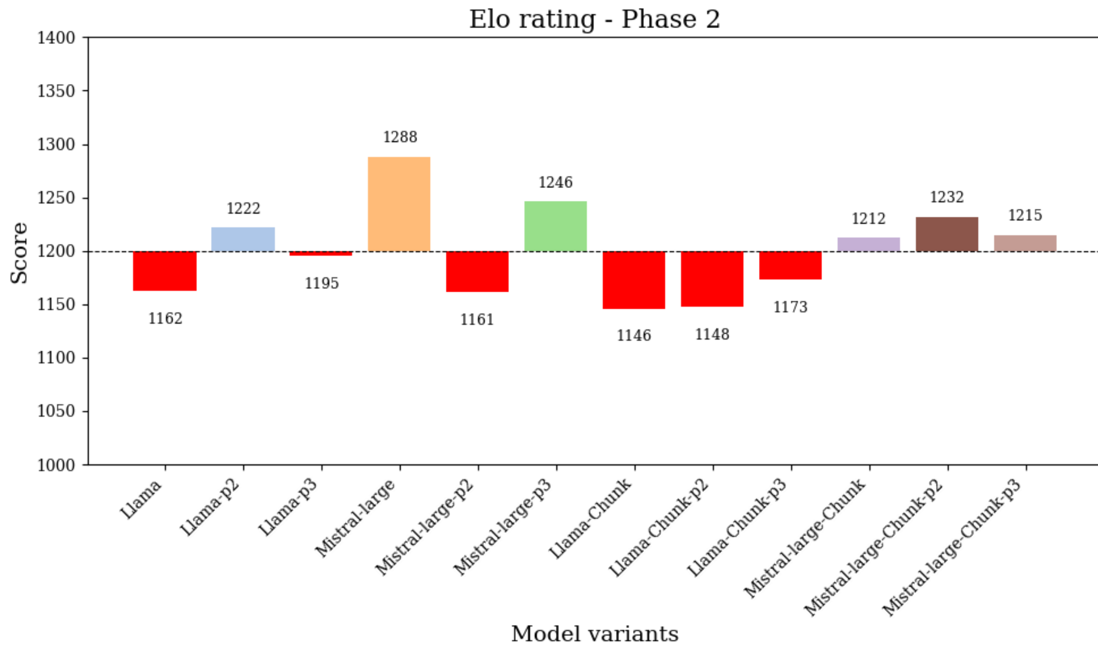



Figure 5.2: Phase two: Results

### 5.1.3 Phase three

In phase three, the Gemini model was introduced, and the impact of temperature settings on performance was evaluated. The top-performing model was Mistral-large with a temperature of 0.9. In second place was Gemini, configured with its maximum token input (1,000,000) and the End-Slice Retrieval method (see Section 4.3), also using a temperature of 0.9. The third-place result was the same Gemini configuration, differing only in its lower temperature setting of 0.1 (see Figure 5.3).

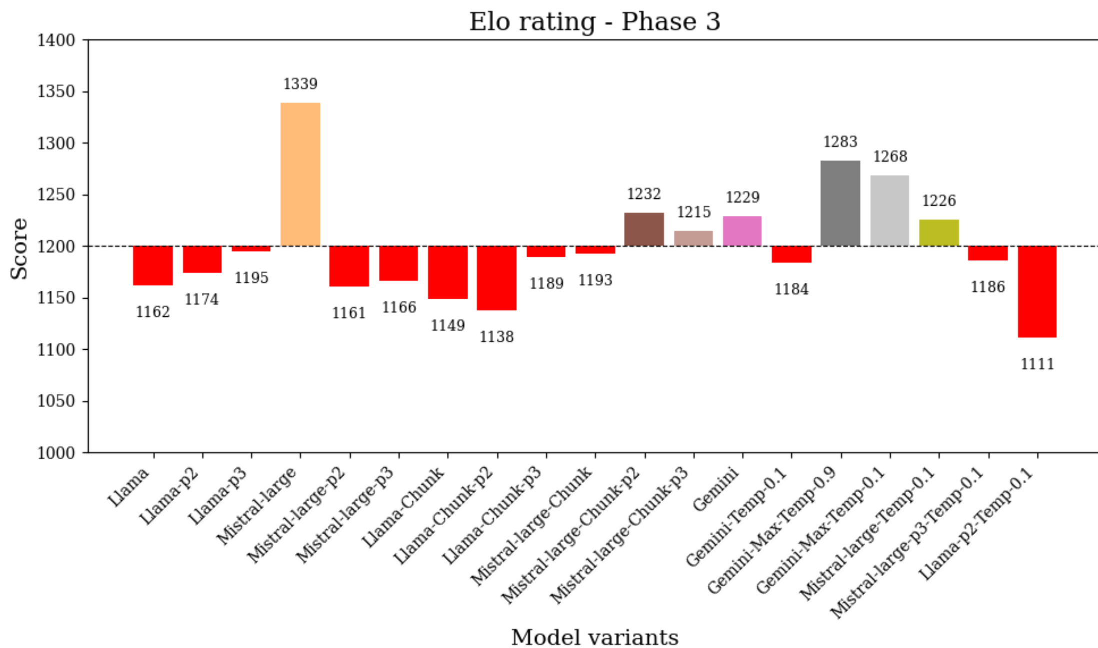



Figure 5.3: Phase three: Results

### 5.1.4 Phase four

Phase four introduced half-chunk and prompt 4. The top performer from the phase was Gemini-Max-half-Chunk-Temp-0.1 with second place being Gemini-Max-Temp-0.1 and Gemini-Max-half-Chunk-Temp-0.9 in third place. Overtaking the previous winner Mistral-large, which ended in fourth place (see Figure 5.4).

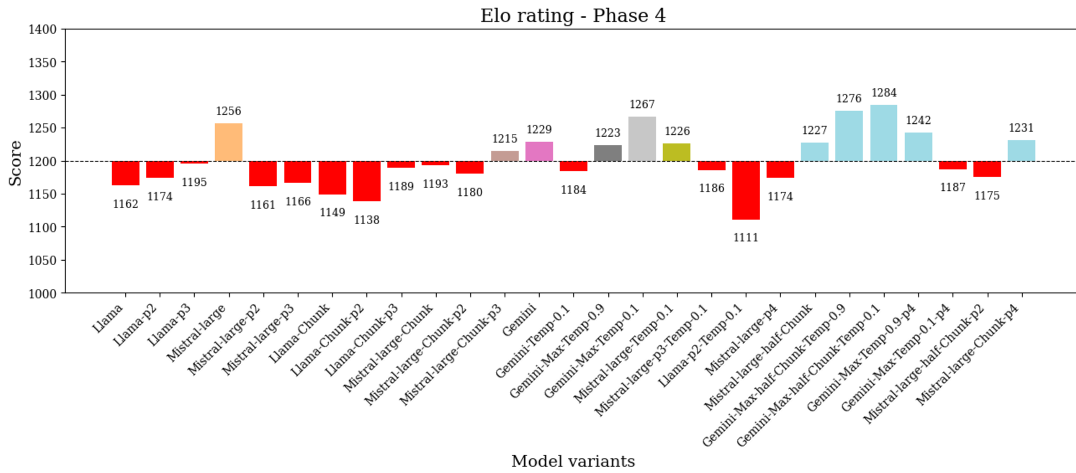



Figure 5.4: Phase four: Results

### 5.1.5 Interview

Average performance score across all engineers: 3.25/5. Summary of the rest of the answers from the interview can be found under subsection 6.2.5

Table 5.2: Interviewees scoring based on the first question

| Engineer | Problem | Score |
|---|---|---|
| A | Log 1 | 4 |
| A | Log 2 | 5 |
| A | Log 3 | 2 |
| A | Log 4 | 3 |
| B | Log 5 | 4 |
| B | Log 6 | 3 |
| B | Log 7 | 2 |
| B | Log 8 | 4 |
| B | Log 1 | 3.5 |
| C | Log 1 | 3.5 |
| C | Log 2 | 4 |
| C | Log 3 | 1 |

# Chapter 6

# Discussion

The following is a discussion on program improvements and the results from the various testing phases.

## 6.1 Program improvements

The program was designed according to the principles of a zero-sum game, similar to chess, where each match results in either a win, loss, or draw for the participating models.

In retrospect, the inclusion of an additional mechanism specifically, a punish button could have enhanced the effectiveness of Elo scoring. In many instances, both models produced flawed outputs, making it difficult for the evaluator to determine a clear winner. A punish function would have enabled evaluators to reduce the Elo ratings of both models in such cases, thereby accelerating the elimination of unsuccessful model variants, in contrast to the skip function that only stepped to the next match without affecting the models' Elo rating.

Although the current Elo-based system does eventually demote weaker models—since they tend to lose against stronger competitors, the absence of a punishment mechanism prolongs this process. Consequently, suboptimal models may remain in contention longer than necessary, reducing the overall efficiency of the evaluation framework.

Another improvement that could have facilitated the evaluation process would have been the implementation of a log filtering function within the arena program. Given the diversity of build log formats, not all evaluators were familiar with every variant. As a result, they often had to skip matches until they encountered logs they could interpret confidently, which limited their ability to assess model outputs accurately. Introducing a filtering feature would have significantly reduced this friction, likely improving the evaluator experience and enabling the collection of more consistent and comprehensive evaluation data.

## 6.2 Phase discussion

From the graphs in the Appendix, Figure 8.2, Figure 8.3, and Figure 8.4 with a rolling average of Phase 2-4, all models were found to have recorded both wins and losses.

Based on the results shown in these graphs, it would have been interesting to extend the duration of each phase. For instance, *Llama-Chunk-p3* in *Phase 2* (see Figure 8.2) exhibited a strong start but experienced a significant drop in points after match 9. Toward the end of this phase, the downward trend appears to be leveling off, which may indicate that the model is beginning to recover and gain points.

Figure 8.4 shows that *Mistral-Large* achieved a higher average than the other model variants, but received a lower score than some of the new variants introduced in that phase. This discrepancy is due to its higher initial Elo rating.

### 6.2.1 Phase one

From the results of phase one, it is evident that the general-purpose language models Llama and Mistral—large, outperformed the code-specialized model, Codestral. Codestral's lack of performance is attributed to its underlying training approach. As a fill-in-the-middle model primarily designed for debugging structured application code (e.g., Python, C++, Java, etc.), Codestral appears less effective at interpreting failed build logs and generating creative solutions. This limitation is likely a consequence of the narrow training objectives and domain-specific focus of the model.

### 6.2.2 Phase two

From the results of phase two, it was surprising to find that the base model, Mistral-large, achieved the highest overall score, suggesting that its original configuration was already well-suited to the task. Prompt three produced above-average results with two model variants—one using chunk mode and the other limited to the last 32,000 tokens of the build log—indicating that the prompt is versatile across different context window filtering strategies. Similarly, prompt two also resulted in two above-average model variants under the same conditions. However, none of these new model variants outperformed the Mistral-large base model.

### 6.2.3 Phase three

The addition of the Gemini model was particularly noteworthy due to its exceptionally large input token capacity. Initially, there was concern that the extended input length might lead to performance degradation, potentially caused by attention dilution. However, both Gemini configurations utilizing the maximum token input performed strongly, securing the top two positions and demonstrating rapid improvement over the course of the phase. This suggests that, contrary to initial concerns, the increased token input had a positive effect. In contrast, the Gemini variant limited to 32,000 tokens achieved a lower score, further supporting the advantage of utilizing the full context window.

Meanwhile, Mistral-large maintained its position as the top-performing model, achieving the highest overall score across all evaluations.

### 6.2.4 Phase four

The results from phase four indicate that adjusting the model's parameters positively impacted its ability to identify the root cause of failure in build logs. The *half-chunk* method appears to be more effective than the *chunk* method, likely due to the structure of build logs—where the first half often lacks relevant failure information. By excluding this portion, the model is less prone to distraction or confusion. Additionally, the temperature setting suggests that a lower value, which reduces generative creativity, may lead to fewer hallucinations compared to higher temperatures.

### 6.2.5 Interview

The general consensus among the interviewees was that the model's suggestions often contained excessive information. They expressed a preference for shorter, more concise responses that pointed directly to the root cause of failure. According to the engineers, the model occasionally produced hallucinations; however, in the vast majority of cases, it successfully identified the underlying cause of the build failure. Furthermore, the engineers indicated a desire for a feature that highlights the relevant section of the log or provides a hyperlink redirecting them to the exact row where the model identified the most probable cause of failure.

All participating engineers agreed that it would be worthwhile to further develop the system into a tool to assist developers in debugging build failure logs, suggesting that the project holds significant promise.

# Chapter 7

# Conclusion

The following chapter includes the conclusion of the research, future work, and an ethical reflection.

## 7.1 Conclusion:

With regard to the research question of whether modern LLMs are sufficiently developed to interpret messages from build logs after failed attempts to build application code. The results demonstrate that parameter adjustments in large language models (LLMs) have a significant impact on their ability to analyze and resolve failed build logs. Furthermore, the quality and relevance of a model's training data play a crucial role in determining its effectiveness in identifying root causes of build failures.

While the current top-performing model variant may not yet meet the company's standards for deployment to developers, it has consistently shown the capability to diagnose complex build failures. However, occasional hallucinations and overly verbose responses remain an area where improvements can be made.

One problem that caused some hallucinations and gave misleading answers was when the reason for failure was from the internal system such as GitHub and internet connections. In those cases, it didn't show in the build-log where the source was, which made it impossible for the LLMs to locate it. By prompting for those cases where there are out-of-reach reasons that caused the error, the LLMs could be led to answer in a more concise way that, for example, there is a network issue, and not further extend the answer.

Overall, the findings suggest that modern LLMs possess strong potential for automated build failure analysis. With targeted fine-tuning and usability improvements, such as reducing hallucinations and enhancing response clarity, these models could become valuable tools for developer workflows.

## 7.2 Future work:

Integrating the *Gemini-Max-half-Chunk-Temp-0.1* with a Retrieval-Augmented Generation (RAG) system backed by Elasticsearch could enhance the model's performance over time [23]. By continuously indexing new build logs from the Axis internal network, the system would provide the model with up-to-date context and examples, enabling more accurate and relevant responses. Over the long term, this setup could also support periodic retraining or fine-tuning of the model on streaming build log data, thereby improving its problem-solving capabilities in real-world scenarios.

To further improve usability, the addition of hyperlinks in the model's responses or a highlighting mechanism that points directly to the causing log row would help developers quickly locate and understand the root cause of a build failure.

Additionally, implementing a time-tracking mechanism would allow for a quantitative evaluation of the actual developer time saved. This could be compared to the time required when resolving build issues manually or using existing tools such as the Jenkins Build Failure Analyzer plugin.

## 7.3 Ethical reflection:

According to the Swedish engineering honor principles, "Engineers ought to strive to improve technology and technological knowledge so as to achieve more efficient use of resources without harmful effects." [24] The pros of the project in regard to the aforementioned principle are that with this new technology, a developer's time spent on debugging code could possibly decrease by a large margin, enabling more time to be spent on deploying more software, thereby benefiting stakeholders.

As of the time of writing the thesis, the current tool in use is the Build Failure Analyzer which needs to be manually updated in order to detect relevant error patterns, a time-consuming process that for the majority of the time does not work effectively. The program could replace the BFA tool, enabling automatic failure detection, and removing the need to manually configure and update the tool.

The cons of this technology are that LLMs use a lot of energy when used frequently, which raises environmental concerns. To make a fair assessment of this factor, the two tools should be tested and compared for energy use to be able to correctly assess whether energy-saving optimization should be implemented into the program, reducing climate effects. Job culture could be negatively affected as not all modern developers are interested in AI-tools and are concerned about job security. Although the program is currently a tool with the sole purpose of assisting developers in debugging their code, which requires human judgment and action to work, there are risks that it could be further developed to be fully automated, eliminating the need for human interference.

# Bibliography

[1] Jenkins Contributors, "Build failure analyzer plugin." https://plugins.jenkins.io/build-failure-analyzer/, 2025. Accessed: 2025-05-17.

[2] E. Frick, T. Li, C. Chen, W.-L. Chiang, A. N. Angelopoulos, J. Jiao, B. Zhu, J. E. Gonzalez, and I. Stoica, "How to evaluate reward models for rlhf," 2024.

[3] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, H. Zhang, B. Zhu, M. Jordan, J. E. Gonzalez, and I. Stoica, "Chatbot arena: An open platform for evaluating llms by human preference," 2024.

[4] D. G. de Pinho Zanco, L. Szczecinski, E. V. Kuhn, and R. Seara, "Stochastic analysis of the elo rating algorithm in round-robin tournaments," 2024.

[5] L. Axlin and K. Broman, "Identification of relevant error descriptions in build logs using machine learning," 2022. LUP Student Paper.

[6] K. Li, A. Zhu, W. Zhou, P. Zhao, J. Song, and J. Liu, "Utilizing deep learning to optimize software development processes," *SAUS PRESS*, 2024.

[7] 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," 2024.

[8] B. Meskó, "Prompt engineering as an important emerging skill for medical professionals: Tutorial," *J Med Internet Res*, vol. 25, p. e50638, Oct 2023.

[9] Ivan Belcic, Coel Stryker, "What is learning rate in machine learning?." https://www.ibm.com/think/topics/learning-rate, 2024-11-27. Accessed: 2025-05-17.

[10] Ivan Belcic, Coel Stryker, "What are model parameters in machine learning?." https://www.ibm.com/think/topics/model-parameters, 2025-05-05. Accessed: 2025-05-17.

[11] E. Saravia, "Prompt Engineering Guide," *https://github.com/dair-ai/Prompt-Engineering-Guide*, 12 2022.

[12] Jacob Murel, Joshua Noble, "What is LLM Temperature? — IBM — ibm.com." https://www.ibm.com/think/topics/llm-temperature, 2024. [Accessed 06-05-2025].

[13] J. F. Smart, *Jenkins: The Definitive Guide*. O'Reilly Media, Inc., 2011. Accessed: 2025-04-28.

[14] Elastic, "What is elastic observability?." https://www.elastic.co/docs/solutions/observability/get-started/what-is-elastic-observability, 2024. Accessed: 2025-04-28.

[15] C. Anderson, "Docker [software engineering]," *IEEE software*, vol. 32, no. 3, pp. 102–c3, 2015.

[16] Wikipedia contributors, "Virtual machine — definitions." https://en.wikipedia.org/wiki/Virtual_machine#Definitions, 2025. Accessed: 2025-04-28.

[17] Ansible Project, *Ansible Documentation: Introduction*. Red Hat, 2024. Accessed: 2025-04-28.

[18] A. Enemosah, "Enhancing devops efficiency through ai-driven predictive models for continuous integration and deployment pipelines," *International Journal of Research Publication and Reviews*, vol. 6, no. 1, pp. 871–887, 2025.

[19] B. T. Willard and R. Louf, "Efficient guided generation for llms," *arXiv preprint arXiv:2307.09702*, 2023.

[20] A. Elo, "The rating of chess players, past and present," 1978. Accessed: 2025-04-08.

[21] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with chatgpt," 2023. Accessed: 2025-04-28.

[22] F. Voron, *Building Data Science Applications with FastAPI: Develop, manage, and deploy efficient machine learning applications with Python*. Packt Publishing Ltd, 2023.

[23] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. tau Yih, T. Rocktäschel, S. Riedel, and D. Kiela, "Retrieval-augmented generation for knowledge-intensive nlp tasks," 2021.

[24] Sveriges Ingenjörer, "Hederskodex," 2025. Accessed: 2025-05-20.

# Chapter 8

# Appendix

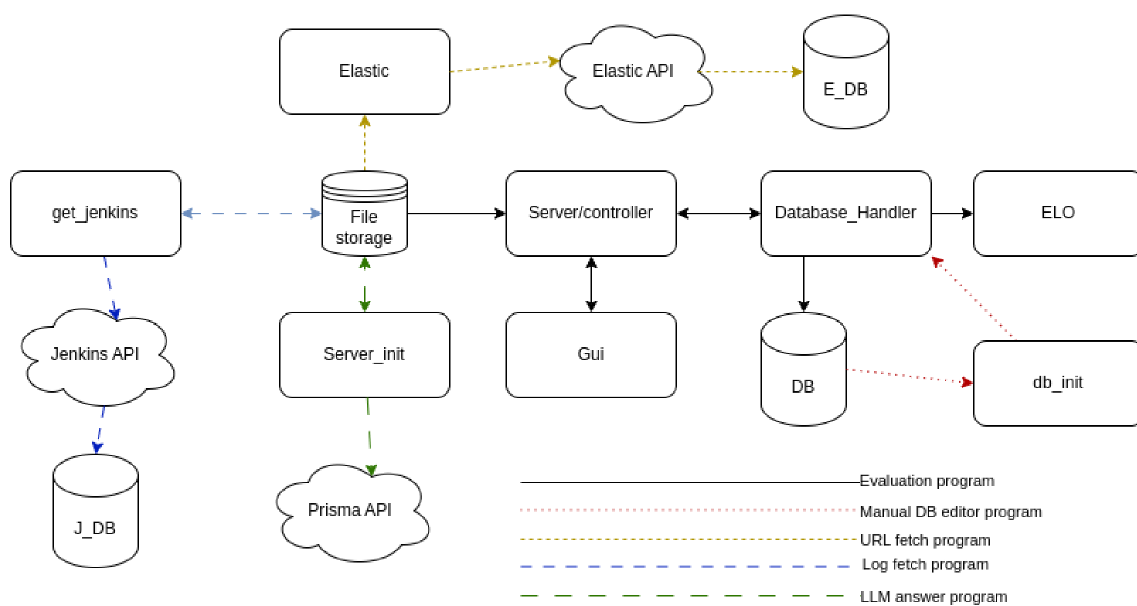



Figure 8.1: Overview of program

Table 8.1: Model variant specifications

| Model | Prompt | Context | Temperature |
|---|---|---|---|
| Llama | Prompt 1 | End-Slice | 0.9 |
| Mistral-Large | Prompt 1 | End-Slice | 0.9 |
| Codestral | Prompt 1 | End-Slice | 0.9 |
| Llama-Chunk | Prompt 1 | Chunk | 0.9 |
| Mistral-Large-Chunk | Prompt 1 | Chunk | 0.9 |
| Codestral-Chunk | Prompt 1 | Chunk | 0.9 |
| Llama-p2 | Prompt 2 | End-Slice | 0.9 |
| Llama-p3 | Prompt 3 | End-Slice | 0.9 |
| Llama-Chunk-p2 | Prompt 2 | Chunk | 0.9 |
| Llama-Chunk-p3 | Prompt 3 | Chunk | 0.9 |
| Mistral-Large-p2 | Prompt 2 | End-Slice | 0.9 |
| Mistral-Large-p3 | Prompt 3 | End-Slice | 0.9 |
| Mistral-Large-Chunk-p2 | Prompt 2 | Chunk | 0.9 |
| Mistral-Large-Chunk-p3 | Prompt 3 | Chunk | 0.9 |
| Gemini | Prompt 1 | End-Slice | 0.9 |
| Gemini-Temp-0.1 | Prompt 1 | End-Slice | 0.1 |
| Gemini-Max-Temp-0.9 | Prompt 1 | End-Slice | 0.9 |
| Gemini-Max-Temp-0.1 | Prompt 1 | End-Slice | 0.1 |
| Mistral-Large-Temp-0.1 | Prompt 1 | End-Slice | 0.1 |
| Mistral-Large-p3-Temp-0.1 | Prompt 3 | End-Slice | 0.1 |
| Llama-p2-Temp-0.1 | Prompt 2 | End-Slice | 0.1 |
| Mistral-large-p4 | Prompt 4 | End-Slice | 0.9 |
| Mistral-large-Half-Chunk | Prompt 1 | Half-Chunk | 0.9 |
| Gemini-Max-half-Chunk-Temp-0.9 | Prompt 1 | Half-Chunk | 0.9 |
| Gemini-Max-half-Chunk-Temp-0.1 | Prompt 1 | Half-Chunk | 0.1 |
| Gemini-Max-Temp-0.9-p4 | Prompt 4 | End-Slice | 0.9 |
| Gemini-Max-Temp-0.1-p4 | Prompt 4 | End-Slice | 0.9 |
| Mistral-large-half-Chunk-p2 | Prompt 2 | Half-Chunk | 0.9 |
| Mistral-large-Chunk-p4 | Prompt 4 | Chunk | 0.9 |

Table 8.2: Table of prompts

| **Prompt version** | **Prompt** |
|---|---|
| Prompt 1 | You are an AI agent that analyzes build logs from a CI-system thoroughly and helps the engineer figure out why the build failed. Be short and concise, but thorough. Main heading should be `AI Summary`. Divide the summary into `Failure Reason` and if possible `Solution`. |
| Prompt 2 | You are tasked with debugging a build script. Focus on only finding errors in the build script. Create a set of facts with clear explanations on how to solve the errors.<br>1. **Error Messages**: Are error messages clear and descriptive?<br>2. **Reason for build log Failure**: What causes the build to fail? |
| Prompt 3 | Your purpose is to find and solve build script errors. Ignore warnings and other flags except error flags. When you find an error, explain your reasoning and assumptions concisely. Answer in markdown with a list of solutions to each error. |
| Prompt 4 | Analyze the following build log and identify the line where the specific build failure occurred.<br>Highlight the offending line and provide a brief explanation of the error. If applicable, suggest what might have caused the failure or what to check next. Make it easy for a human to locate and verify the failure in the log. |

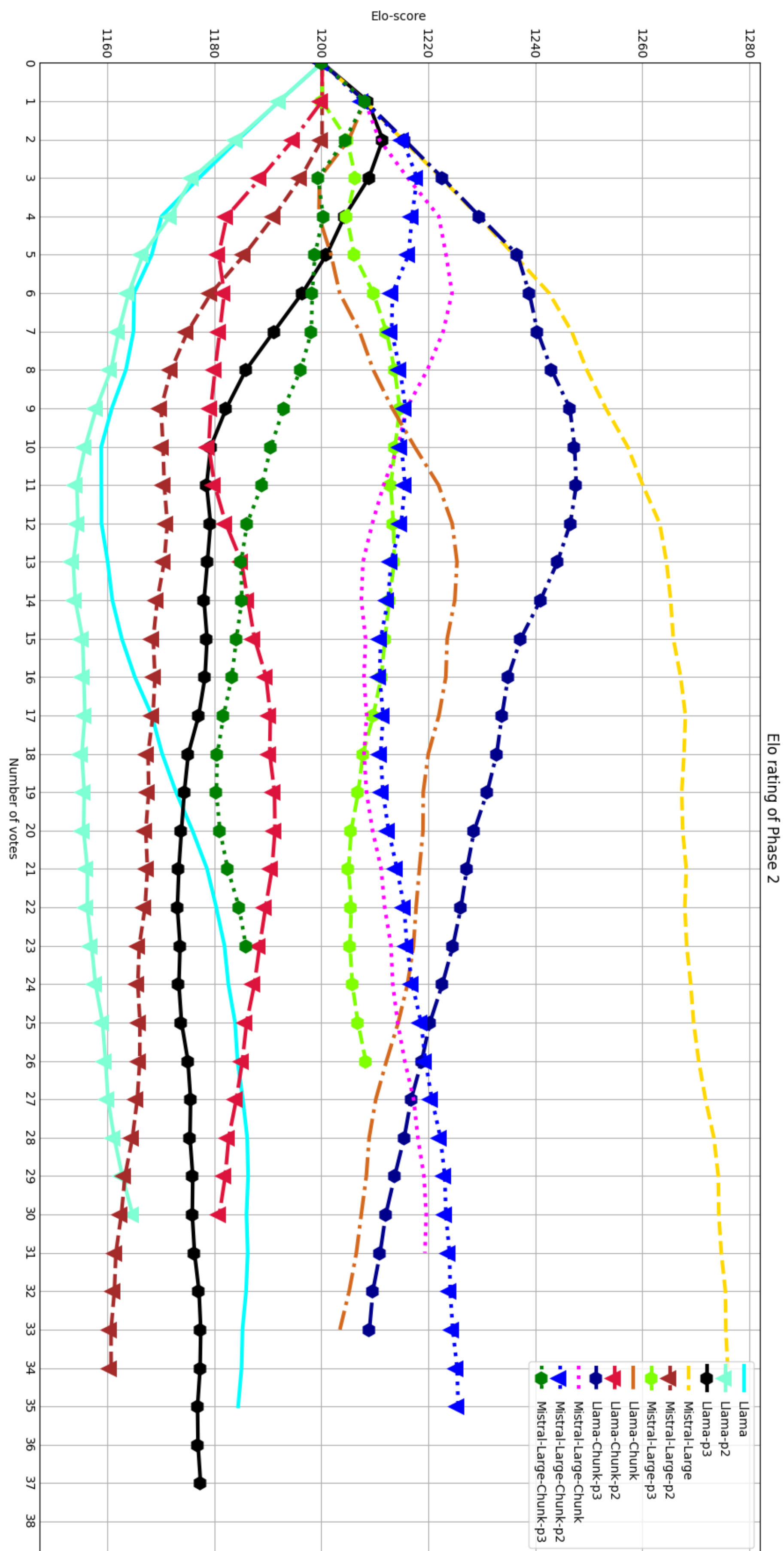



Figure 8.2: Phase 2: Rolling average

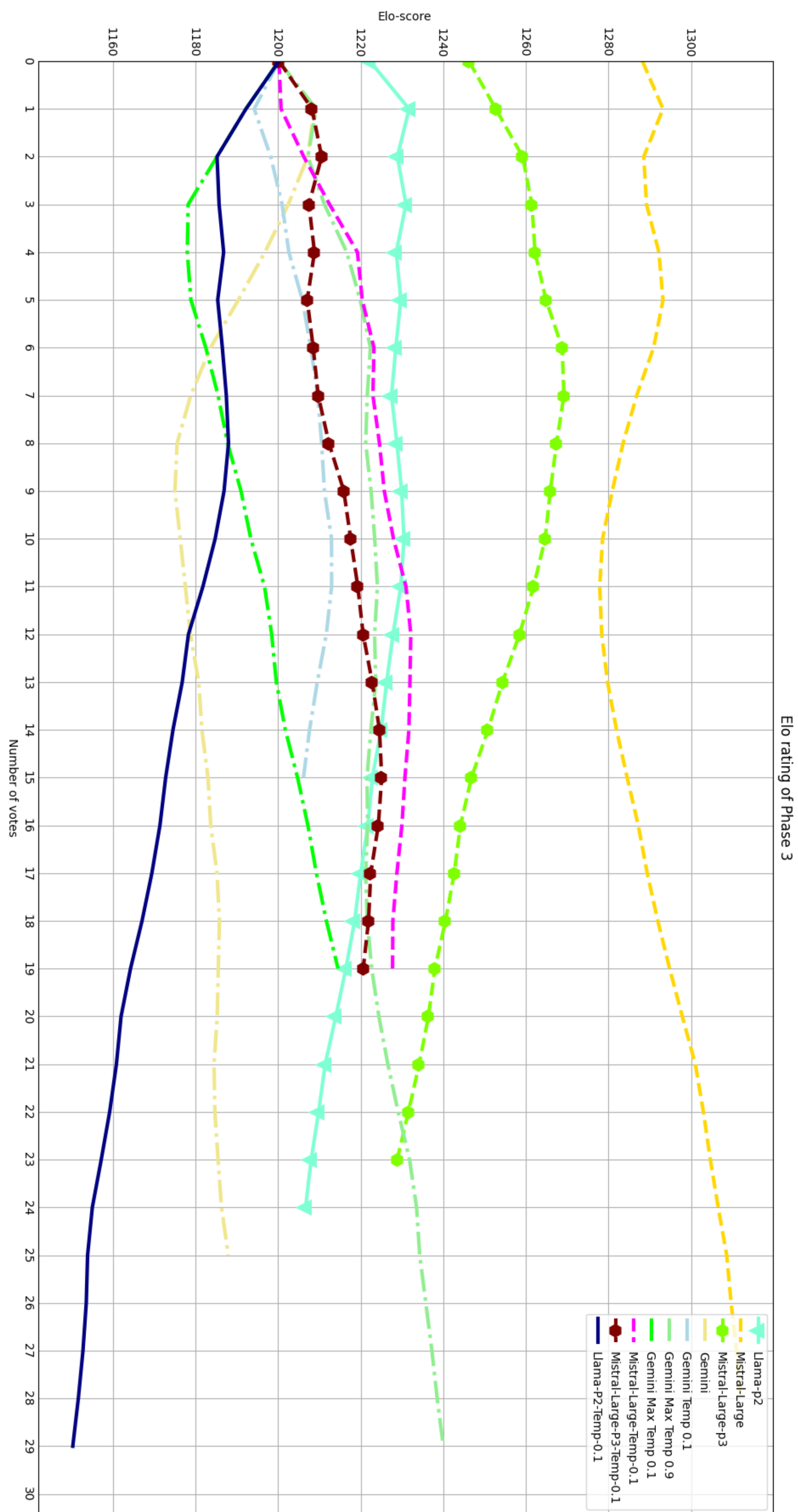



Figure 8.3: Phase 3: Rolling average

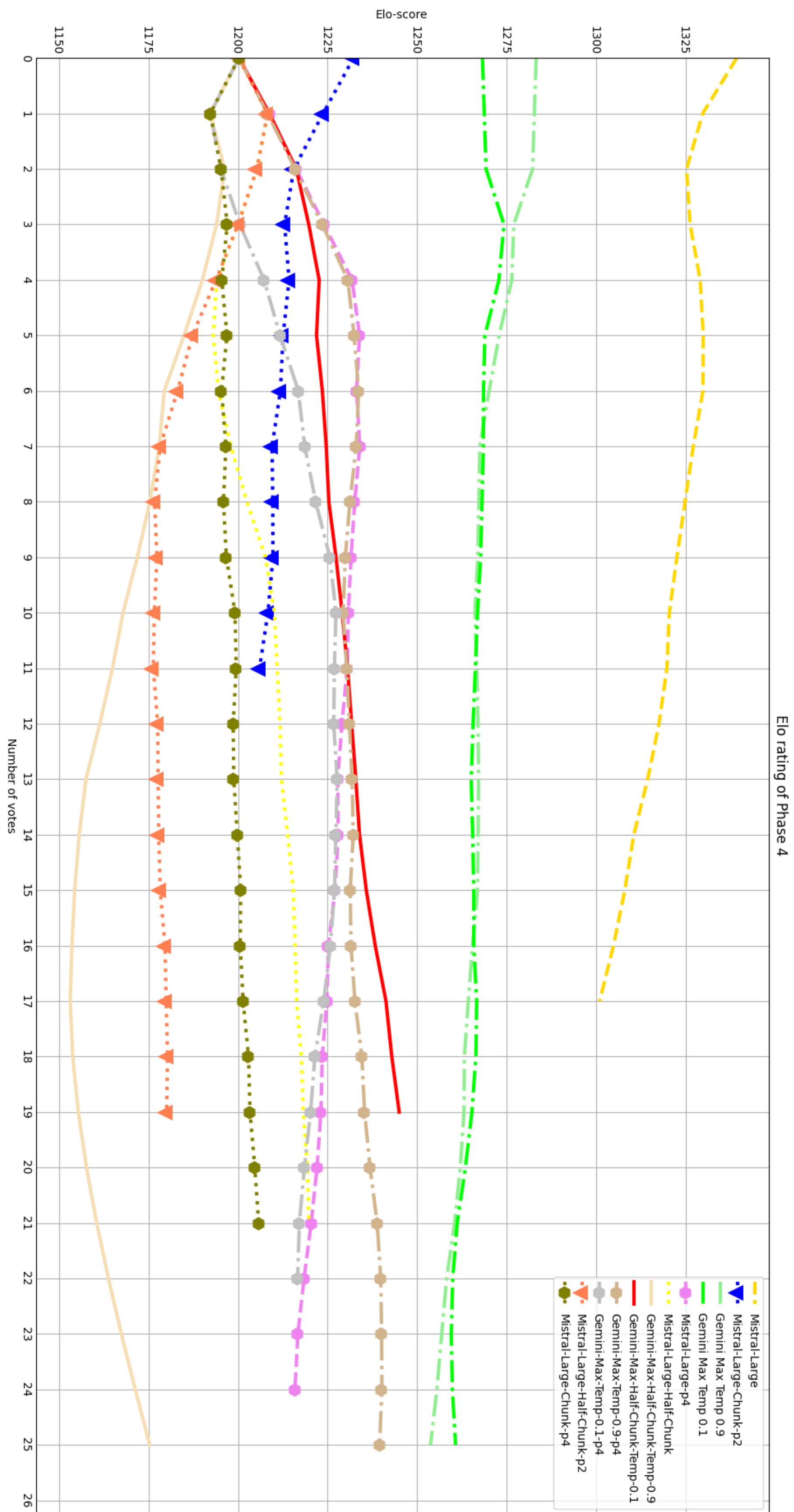



Figure 8.4: Phase 4: Rolling average