MLOps Pipeline Generation for Reinforcement Learning: A Low-Code Approach Using Large Language Models

Stephen John Warnett[a,b,*], Evangelos Ntentos[a], Uwe Zdun[a]

[a] *Research Group Software Architecture, Faculty of Computer Science, University of Vienna, Vienna, Austria,*
[b] *UniVie Doctoral School Computer Science DoCS, Faculty of Computer Science, University of Vienna, Vienna, Austria,*

Abstract

MLOps (Machine Learning Operations) and its application to Reinforcement Learning (RL) involve various challenges when integrating Machine Learning and RL models into production systems, entailing considerable expertise and manual effort, which can be error-prone and obstruct scalability and rapid deployment. We propose a new approach to address these challenges in generating MLOps pipelines. We present a low-code, template-based approach leveraging Large Language Models (LLMs) to automate RL pipeline generation, validation and deployment. In our approach, the Pipes and Filters pattern allows for the fine-grained generation of MLOps pipeline configuration files. Built-in error detection and correction help maintain high-quality output standards.

To empirically evaluate our solution, we assess the correctness of pipelines generated with seven LLMs for three open-source RL projects. Our initial approach achieved an average error rate of 0.187 across all seven LLMs. OpenAI GPT-4o performed the best with an error rate of just 0.09, followed by Qwen2.5 Coder with an error rate of 0.15. We implemented a single round of improvements to our implementation and low-code template. We reevaluated our solution on the best-performing LLM from the initial evaluation, achieving perfect results with an overall error rate of zero for OpenAI GPT-4o. Our findings indicate that pipelines generated by our approach have low error rates, potentially enabling rapid scaling and deployment of reliable MLOps for RL pipelines, particularly for practitioners lacking advanced software engineering or DevOps skills. Our approach contributes towards demonstrating increased reliability and trustworthiness in LLM-based solutions, despite the uncertainty hitherto associated with LLMs.

Keywords: MLOps, RLOps, machine learning, reinforcement learning, low-code development, large language models, automated pipeline generation

1. Introduction

Machine Learning Operations (MLOps) [1] refers to the automated provisioning of Machine Learning (ML) models by implementing repeatable workflows. These workflows help to ensure process reproducibility and can potentially mitigate risks associated with model management and deployment [2, 3]. Despite noteworthy progress in MLOps, critical challenges remain in integrating ML models into production systems. Conventional methods often require considerable manual effort to transition ML scripts from experimental environments, like computational notebooks in Google Colab[1] or Project Jupyter[2], primarily used by data scientists, to production-ready delivery pipelines [4]. This manual hand-off creates inefficiencies and increases the likelihood of errors, obstructing scalability and rapid deployment [5]. Consequently, there is a pressing need for methodologies that can automate and simplify the creation of production pipelines for MLOps, reducing reliance on extensive, specialised DevOps and software engineering expertise [2, 6]. The application of MLOps to Reinforcement Learning (RL) is sometimes called RLOps [7], and although MLOps refers specifically to ML tasks, it equally applies to RL in the form of RLOps [8, 9]. In this paper, we focus on RL pipelines. However, since RLOps is a specialised version of MLOps and has not yet fully gained widespread terminological adoption, and the proposed approach can be equally applied to both MLOps and RLOps, we mostly use the term "MLOps" unless the distinction is relevant.

This paper addresses the challenges described above (manual effort, inefficiencies, error-proneness, scalability, deployment velocity, automation and expertise) by proposing a novel approach to automating MLOps CI/CD (Continuous Integration/Continuous Delivery) pipeline generation by implementing the Pipes and Filters pattern [10] to apply few-shot prompting and prompt chaining [11], error detection and retries to guide Large Language Models (LLMs) in generating GitLab[3] CI/CD pipeline configuration files based on low-code, template-based

*Corresponding author

Email addresses: `stephen.warnett@univie.ac.at` (Stephen John Warnett), `evangelos.ntentos@univie.ac.at` (Evangelos Ntentos), `uwe.zdun@univie.ac.at` (Uwe Zdun)

[1] `https://colab.google`

[2] `https://jupyter.org`

[3] `https://gitlab.com`

MLOps CI/CD pipeline specifications for the RL lifecycle.

Our method employs a low-code approach grounded in template-based, natural language specifications. More specifically, a textual generation template is provided that can be adapted with natural language-based specifications to guide the practitioner, and coding knowledge is needed only to review inputs and generated artefacts.

Our approach is designed to automatically generate, validate, and deploy RL pipelines. It utilises LLMs to facilitate the generation of production-level systems from initial RL scripts and generation requirements specified by the user. These models, specialised through prompt engineering techniques and integrated with supporting components, provide modular and flexible automation suited to different deployment contexts.

The modular architecture of our approach employs the Pipes and Filters pattern, orchestrating the flow between various LLM-based generation modules and supporting components. Each module specialises in discrete tasks, such as creating Dockerfiles, generating CI/CD pipelines, or issue detection. The architectural design provides quality assurance in various tasks, including generation retries, linting, running validation tasks and automated error correction, enhancing the reliability of the generated pipelines.

In addition, the natural language–based specification supports this generation architecture, enabling non–software engineering experts, such as data scientists and domain specialists (e.g. in cyber-physical production systems (CPPSs) [12, 13] or security), to actively contribute to pipeline development - assuming that supporting infrastructure will be provided to them - and offers an alternative to manually writing pipeline specifications and using LLM code completions, since it obviates the need for specialist knowledge of how to structure specific pipeline configuration implementations. Although a formal user study is planned for future work, early feedback from demonstrations with our industry project partner indicates that domain specialists can use our solution to author and refine pipeline descriptions without prior LLM and expert MLOps expertise to generate reliable GitLab pipeline configuration files. Our industry project partner is a large multinational technology company providing production automation solutions for smart factories. Its employees develop RL-based Industry 4.0 [14] CPPSs, including manually-written MLOps pipeline configurations for RL, and manage their own infrastructure.

Our solution accommodates many pipeline architectures, from straightforward CI/CD pipelines dedicated to training and evaluation to more detailed Dockerised pipelines that involve comprehensive testing, error handling, artefact management and deployment. Thus, both standard and advanced practices are catered for.

We evaluate three open-source RL examples and compare the capabilities of seven LLMs to assess our approach. Our empirical evaluation compares the performance of our automated pipeline generation against expected pipelines, focusing on factors such as error rates and execution success.

Specifically, we aim to answer the following research questions:

- **RQ1** How accurately can a low-code, template-based modular flow approach — leveraging LLMs for generation and validation — produce correct MLOps pipeline configurations for RL?
- **RQ2** How accurately do different LLMs generate MLOps pipeline configurations for RL, and what are their limitations?

Our work makes two significant contributions. Firstly, we implemented a modular flow architecture based on the Pipes and Filters pattern. This flexible yet powerful design means that specific components have well-defined responsibilities, making our approach efficient and precise. Our implementation also provides retry, validation, and error correction features, leading to high reliability of generated pipelines. Secondly, our low-code, extensible template-based approach allows for guided, convenient, powerful, user-friendly natural language specifications to describe and design pipeline architecture steps, allowing stakeholders without software engineering or DevOps skills to participate in the development process in a cross-disciplinary fashion. All project artefacts, including the implementation, are available in our Zenodo replication package [15].

The rest of the paper is organised as follows. In Section 2, we describe MLOps CI/CD pipeline architectures, and in Section 3, we discuss related studies and compare them with our work. Section 4 presents the approach we adopted in this study and provides an illustrative example of the approach in action. Section 5 describes our study objects, scenarios and evaluation variables. Section 6 details our initial results, and Section 7 describes the improvements we made to our solution and the results of our reevaluation. Section 8 describes how our solution can be integrated into an industrial setting, and in Section 9 we discuss our results in the context of our research questions. In Section 10, we consider potential threats to the validity of our study. Finally, Section 11 summarises the paper.

2. Background

This section provides background on ML, RL and tasks associated with the RL workflow when creating MLOps pipelines relevant to this study. We also describe aspects of MLOps and MLOps CI/CD pipeline architecture pertinent to this paper.

2.1. Reinforcement Learning

ML and RL [16] use different approaches to their operational methods. ML uses supervised, semi-supervised or unsupervised learning techniques to extract predictions from data, and RL trains agents for sequential decision-making through environmental interactions and reward-based feedback. The process requires agents to describe their problem, states, actions, and the definition of a reward system. Continuous performance enhancement follows an iterative training process, which starts by setting up an environment and then defining a policy for model training algorithms before hyperparameter adjustment and policy evaluation. Differences between ML and RL warrant acknowledgement during the development of MLOps systems and CI/CD pipelines aimed at automation. For example, RL uses complex model architectures and distribution strategies and practices such as model checkpoints, which impose requirements on system architecture and necessitate a checkpoint management system affecting data storage, retrieval and deployment mechanics [8].

Since this study focuses on CI/CD pipelines for RL, and we involve minimal external components for the RL projects we use, the RL details are confined to the source code of the RL scripts executed at various stages of the defined CI/CD pipelines. Hence, we are concerned with a higher level of abstraction than the low-level implementation details – we are more interested in the nature and order of the executed RL pipeline stages. Pipeline architecture is discussed in detail in Section 2.2.

2.2. MLOps CI/CD Pipeline Architecture

MLOps is an all-encompassing approach to the ML workflow [17] and includes such activities as model training, deployment and the management of ML models in production environments [18], defining processes and practices for model lifecycles. It has much in common with DevOps [19, 20] since it concerns process automation and integration, and the use of CI/CD in the build and deployment process.

This study focuses only on CI/CD-relevant [21, 22] aspects of MLOps. CI (Continuous Integration) refers to changes to the codebase being tracked, tested and reported to ensure the correctness of the code and to provide fast feedback whenever a change to the codebase breaks the project. CD (Continuous Delivery) builds upon CI and involves the (semi-)automatic packaging and deployment of software to facilitate frequent, low-friction releases. Other aspects of relevance to CI/CD include software builds, environment provisioning, and various testing processes.

We concentrate on automating the RL models' build, training and deployment process. ML-specific practices, which one would expect to find in CI/CD pipelines for ML [17, 23], are excluded since they are irrelevant to RL. Such practices include, for instance, data preprocessing (data validation, data cleaning, feature engineering and data transformation). Limiting the scope in this way means that the focus of the study leans towards CI/CD pipeline configuration for RL, although the approach can also be applied to ML.

The MLOps CI/CD pipeline architecture contains various well-defined stages, most of which run automatically, accelerating the process of creating and deploying ML models. A typical MLOps CI/CD pipeline architecture for RL is depicted in Figure 1. It consists of five key stages that are of relevance to this study – *testing*, *model training*, *model testing*, *model evaluation* and *model deployment*. The *testing* stage focuses on source code. It may invoke various test types, such as unit tests, integration tests, performance tests, security scans and static source code analysis. Unit tests check the implementation for correctness, whereas integration tests verify the interaction between different components. Performance tests evaluate the system's performance metrics, and security scans identify potential vulnerabilities. Static analysis evaluates code for possible problems during the non-execution of the program.

During the *model training* stage, automatic training of the RL model takes place using specified algorithms. This stage also provides a model artefact for subsequent phases.

The *model testing* stage performs validation of the trained model according to predefined criteria. It also checks whether the model meets the specified criteria before it proceeds to the *model evaluation* phase.

The model approaches deployment readiness as the *model evaluation* stage determines its operational performance. Key activities in this stage may include evaluation via suitable metrics to measure model performance and A/B testing as an algorithm for deciding which version produces maximal performance results. Stress testing can also reveal a model's abilities during critical conditions. Finally, predictive performance checks may check the model's predictive capabilities.

The final MLOps CI/CD pipeline stage – *model deployment* – entails deployment of the model to one or more operational platforms within associated target environments. A staging environment allows users to conduct final testing steps. After satisfying all the testing criteria and verification requirements, the model can be deployed to a production environment. Deployment to a production environment enables clients to use the model. Blue-green deployment allows a new model version to be deployed alongside existing versions. Another deployment option is canary deployment, in which a new model version is deployed to a minimal number of clients before finally being fully rolled out. Deployment to a specific environment can be automatic or executed with minimal manual intervention (typically a single click by a human operator within the CI/CD platform).

The pipeline definitions used in the open-source projects in the study only use a subset of the functions described above. To focus on the core stages and activities within the pipeline architecture, we also omit many of the details described above in Figure 1. We exclude external collaborating components such as source code repositories, metadata stores, model registries and feature stores, and deployment environment practices such as environment provisioning, model performance monitoring and drift detection. Furthermore, pipeline triggers, such as the triggering of a pipeline upon a change to the codebase, are also left out since they are not defined within the CI/CD pipeline configurations but externally. Finally, some solutions may implement pipeline stages or activities irrelevant to this study, such as those for packaging or containerisation of deployment models.

3. Related Work

This section describes related work, covering MLOps and RL in a CI/CD context and using LLMs for code generation.

3.1. Related Work on MLOps and ML in a CI/CD Context

MLOps is widely used as it seeks to enhance the deployment of ML models through automated processes. Vemuri and Thaneeru [24] investigate the incorporation of AI-optimised DevOps within cloud CI/CD environments, outlining the journey from design to implementation. In prior work [23], we analyse architectural design decisions for deploying ML models

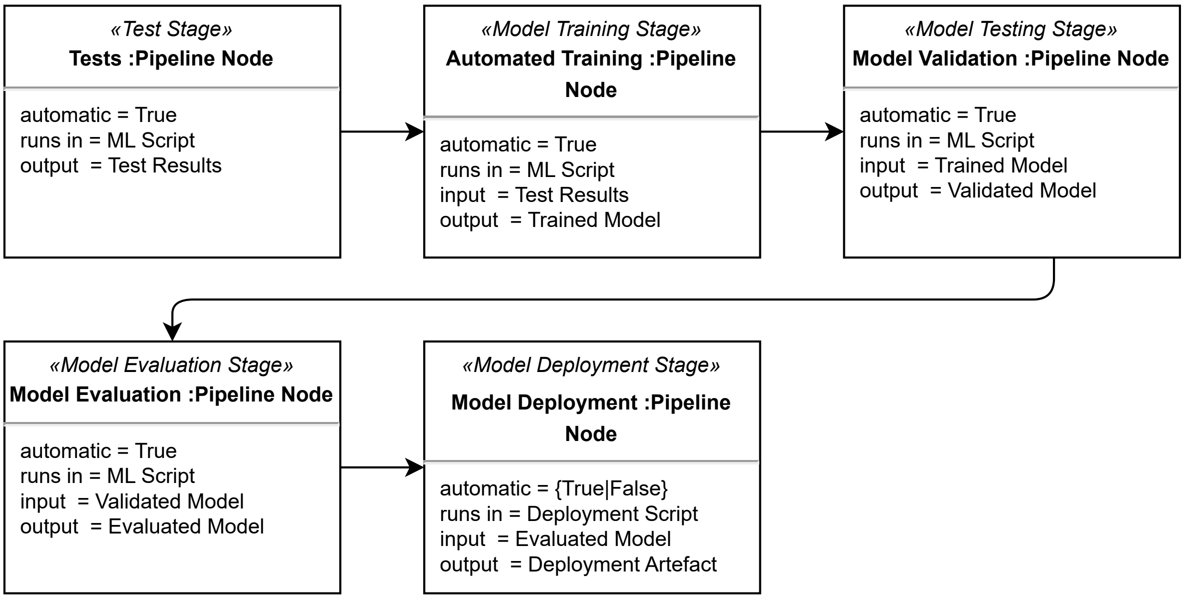

Fig. 1: A typical MLOps CI/CD pipeline architecture.

and emphasise CI/CD automation in such architectures. We also examine architectures that specifically address ML workflows [17]. Zhang et al. [25] conduct another study on architectural decisions for AI-based systems and note that architectural decisions in such systems are primarily technical and closely related to the specific characteristics of those systems. Kumara et al. [26] present a reference architecture for MLOps after reviewing industry literature, tackling the difficulty of choosing suitable tools and design strategies for developing MLOps environments. This work complements these studies by offering a specific low-code, template-based approach for automating RL pipeline generation.

Several studies also address the difficulties of incorporating ML into established DevOps practices. For instance, Sharma et al. [27] examine alternatives for implementing ML techniques within CI/CD pipelines but fail to provide a complete solution. Liang et al. [28] concentrate on automating model training, promoting transparency through version control and integrating ML into conventional CI/CD pipelines whilst stressing the importance of ongoing monitoring and feedback loops to sustain model performance. In contrast, our research introduces an innovative framework for automating RL pipeline generation using LLMs and a low-code methodology, aiming at a broader user base, including those without expert knowledge.

3.2. Related Work on Low-Code Programming

Interest in low-code development methods has increased considerably in recent years [29, 30]. Hirzel [31] attempts to demystify the term "low-code" and discusses how low-code programming reduces reliance on traditional text-based programming languages like C, Java and Python, presenting low-code as a different approach which uses natural and visual languages that resemble user task thinking patterns. Our study uses this concept by employing a low-code, template-based, natural language approach for pipeline specification, leaving the code generation to LLMs.

Kirchhof et al. [32] observe that low-code development platforms (LCDPs) commonly apply one-shot generation. At the same time, this style of code creation leads to problems because the code gets replaced during subsequent generation phases, thus undermining the core concepts of low-code platforms. To reduce this problem, our approach implements few-shot prompting, generated code validation through error detection, and automatic generation retry functions. Our approach eliminates the need for development platforms because it permits easy pipeline definition through simple, natural language-based templates.

Bruhin et al. [33] consider the connection between LCDPs and generative AI. Their research incorporates expert interview findings, demonstrating how generative AI can transform the potential of LCDPs. The study indicates that code generation through this combination can boost productivity and minimise errors. Our approach aims to demonstrate the benefits of combining a low-code approach with generative AI without needing LCDPs. Instead, we opt for a natural language template specification for pipelines.

3.3. Related Work on Using LLMs for Code Generation

Several related works aim to understand and enhance LLM outcomes for code generation tasks, ranging from improvements in code generation capabilities to addressing challenges and limitations, integrating LLMs with other technologies and processes, or focusing on practical applications.

A systematic assessment of LLM code generation challenges is performed in an analysis by Jin et al. [34]. The six-layer vision model they describe divides the code generation operations into four sequential phases known as *input*, *orchestration*, *development* and *validation*. Their study seeks to broaden the scientific understanding of LLM-based code generation programs to accelerate their development. The authors present specific guidelines and suggestions to enhance LLM-based code generation system reliability, robustness and usability.

The study by Jiang et al. [35] presents a survey about LLM advancements for code generation tasks. The study develops a taxonomy for recent developments, covering data curation, model performance evaluation and real-world applications. The authors employ established benchmarks for empirical testing, providing insights into the evolution and effectiveness of LLMs in code generation. Their study also identifies critical challenges and promising opportunities concerning the gap between academia and practice and establishes an online repository to document advances in the field continuously.

Li and Murr [36] explore how current GPT-4 models function for generating synthesised programs. They introduce a repository that connects these models to the HumanEval [37] dataset to perform zero-shot Python code generation evaluations. Their study establishes that GPT-4 model variants can effectively solve HumanEval problems and achieve competitive performance using multi-step prompts, demonstrating their potential for program synthesis enhancement.

Li et al. [38] address natural language understanding in code-focused LLMs. Their novel framework comprises two modules – *AttentionExtractor* operates to acquire user requirement statements, and *AttentionCoder* performs code generation from these statements. This framework enables traditional natural language processing tools to integrate with code generation models, which allows them to execute user intents across programming languages.

Our approach complements the above studies by utilising LLMs to improve the flexibility and robustness of MLOps pipeline generation by incorporating natural language processing. LLMs can potentially handle unpredictable or polyglot code situations [39], allowing them to seamlessly interpret and generate code in multiple programming languages or dialects. This adaptability is crucial when meeting project requirements and integrating with existing systems. Furthermore, LLMs allow users to extend low-code specifications with customised functions or unforeseen extensions, facilitating innovation and customisation without requiring in-depth knowledge of software engineering.

Using LLMs for issue detection and code generation introduces accuracy challenges [40, 41], such as incorrect classifications and AI hallucination. The model may generate implausible or irrelevant suggestions due to overfitting, insufficient training data, or a limited token window size [42]. White et al. [43] identify issues using LLMs like ChatGPT to generate application programming interface (API) specifications from natural language requirements and simulate APIs to enable developers to interactively test APIs without deploying a mock server. They suggest generator and prompt engineering patterns to expedite the initial API development by generating synthetic data from natural language descriptions, but do not achieve perfect results.

LLM specialisation and fine-tuning, often combined with model evaluation and testing, are essential for achieving acceptable outcomes [44, 45]. LLMs' accuracy and reliability limitations as coding assistants have been well-documented [46]. They can handle simple programming tasks but struggle with detailed descriptions and have limited attention spans, making their performance highly dependent on the specific inputs and prompts used [47]. Whilst LLMs show promise in automatic program repair [48, 49], they often replicate common programming mistakes found in human-crafted solutions [50]. In security code reviews, LLM-generated responses tend to be verbose, vague and incomplete [51]. Given that generating comprehensive CI/CD pipelines involves complex architectural considerations, similar accuracy and reliability issues are expected.

Our approach heeds the lessons of prior works by focusing on the architecture of the supporting components (as also advocated by Tan et al. [44]) and employing a structured generation flow with focused steps and contextual feedback. Our solution enables detailed, natural language, template-based user input to improve the generation process further. Furthermore, our solution addresses the reliability and trustworthiness issues inherent in LLMs via built-in quality control and retry logic, validated via two rounds of evaluation. To our knowledge, no similar approach has been suggested in prior works.

4. Approach

In this section, we describe the key constituents of our approach: our research process, the proposed workflow, the specification template, the prompting method, an illustrative example of the template-based low-code approach, the generation architecture, and the definition of generation flows within the implementation.

4.1. Research Process

We followed the systematic, structured methodology shown in Figure 2. The first step (**Tool Implementation**) was to implement the approach described in this section. We then searched for open-source RL projects on the Web and in code repositories (**Search Web and Repositories for Candidate Projects**) to find candidates to evaluate our approach. We iteratively searched and applied specific inclusion/exclusion criteria (described in Section 5.1) to determine which projects were suitable for the study (**Apply Inclusion/Exclusion Criteria**). Next, we adapted each project for CI/CD pipeline suitability and wrote Dockerfiles and pipeline specification files for each included project (**Adapt Projects and Write Dockerfiles and Pipeline Specifications**). The pipeline specification files were used as inputs to our implementation, as detailed later in this section, and we also used them to compare with the LLM-generated pipeline configurations when evaluating them. Once we had selected appropriate projects (see Section 5.1 for details) and prepared the files as described in the previous step, we ran our implemented approach on each project, scenario and LLM three times, and recorded the output (**Run Tool on Each Project and Scenario**). We subjected the results generated by the LLM to a comparative analysis against the original pipeline specifications and the expected behaviour of the executed pipelines based on the generated pipeline configurations to understand how our approach performed for different

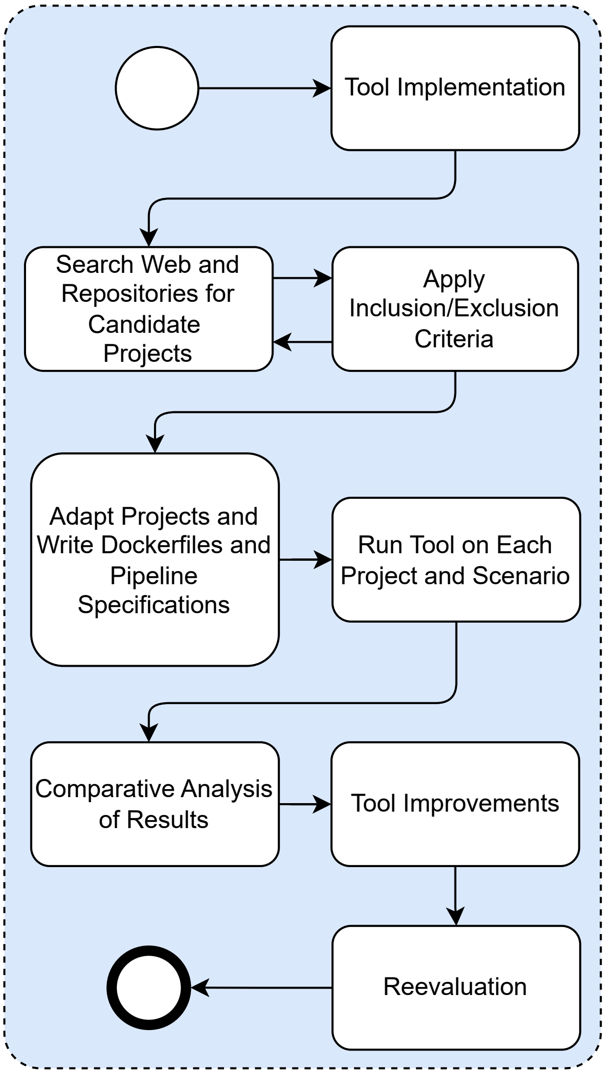

Fig. 2: The research process followed in this paper.

projects, scenarios and LLMs (**Comparative Analysis of Results**). We then improved our implementation (**Tool Improvements**) and ran a reevaluation on the best-performing LLM from the first evaluation (**Reevaluation**). Appendix A contains tables showing the results of the comparative analysis and reevaluation, Section 5 describes the scenarios and evaluation variables, Section 6 describes the initial results, Section 7 highlights the improvements we made and the results of the reevaluation, and Section 9 discusses our results. The rest of this section describes our approach and its implementation as a tool.

4.2. Traditional Approach

Building and deploying ML models requires complex pipelines, which are steps for data processing, training, validation, and deployment. Traditionally, this is a highly manual, error-prone process that requires specialised DevOps/MLOps expertise. Errors in these pipelines directly impact the trustworthiness of the resulting ML system. If the pipeline is flawed, the ML model or RL agent will be trained and deployed with potentially incorrect or biased data, leading to unreliable behaviour.

Considering GitLab CI/CD configuration as an example demonstrates the high complexity of the conventional pipeline setup procedure. Manual GitLab CI/CD pipeline setup involves creating and managing `.gitlab-ci.yml` files, which specify the complete pipeline structure in YAML[4]. Practitioners must clearly define each part of the pipeline, such as stages, jobs, scripts, dependencies, variables and deployment targets. In MLOps applications, these settings are much more complicated as the data preprocessing, model training, validation, and deployment stages need to be orchestrated[5].

The setup process usually starts with defining pipeline stages. Every task in these stages must be configured manually with appropriate Docker images, scripts, artefacts and environment variables. The practitioner should also configure GPU runners for training jobs, set data mount points, and set model storage locations[6].

A standard MLOps pipeline setup in GitLab requires complicated job orchestration that requires ML engineers to liaise with DevOps engineers to set up proper artefact flows between the training and deployment phases. The configuration needs to support large model files that are beyond the scope of Git[7], and thus need to be integrated with external model registries or stored as GitLab artefacts and managed carefully as binary files. Conventional methods require manual definition of resource needs, timeout settings, retry strategies and failure management systems on a per-job basis[8].

The complexity of the configuration is further increased by the necessity to deal with several environments (development, staging and production) with various resource limitations and security needs. Every environment needs a different job configuration, variable management and deployment strategy, which often means hundreds of lines of YAML configuration. The user has to manually specify conditional job execution, dependencies between jobs, and the correct artefact propagation across the pipeline[9].

The classic GitLab CI/CD setup procedure for MLOps introduces several layers of complexity that even seasoned professionals can find challenging to handle. The YAML syntax is also highly formatted and indented, so configuration files are error-prone and hard to debug. The complex pipeline architectures require using advanced YAML features like anchors, aliases, and `extends` keywords, to reduce code duplication, which introduces additional syntactic complexity[10].

The conventional method also demands extensive expertise in GitLab's advanced features, including parent-child pipelines to support complicated MLOps processes, multi-project pipelines to support microservice-based architectures, and conditional pipeline execution depending on code changes.

[4] `https://yaml.org`
[5] `https://docs.gitlab.com/ci`
[6] `https://blogs.nvidia.com/blog/what-is-mlops`
[7] `https://git-scm.com`
[8] `https://about.gitlab.com/blog/continuous-machine-learning-development-with-gitlab-ci`
[9] `https://about.gitlab.com/blog/efficient-devsecops-workflows-with-rules-for-conditional-pipelines`
[10] `https://docs.gitlab.com/ci/yaml/yaml_optimization`

These features can be learned only with a significant investment in GitLab-specific expertise, which may not apply to other CI/CD systems[11] and do not generally fall within the area of knowledge of ML engineers.

Conventional GitLab CI/CD setup methods pose a significant scalability bottleneck when organisations seek to deploy numerous ML models or scale their MLOps capabilities. Every new model or use case also necessitates a new configuration file, duplicating slightly different pipeline definitions. This proliferation of configurations requires high maintenance because updates to common patterns have to be manually applied to many files[12].

Large organisations have specific difficulties in standardising MLOps pipelines across several teams and projects. The conventional methods do not have a sound system for sharing configuration patterns, and as a result, each team tends to have different versions of similar pipelines. This disintegration leads to inconsistency in deployment, security and monitoring within the organisation[13].

The conventional GitLab CI/CD setup procedure generates considerable delays in the development and deployment pace in MLOps processes. The manual process of creating a configuration makes creating new MLOps pipelines a multi-week process instead of a multi-hour process when incorporating complex ML-specific needs. Every configuration change needs thorough testing and verification to make sure that the pipeline runs properly, which slows down the iterative development process[14].

Debugging conventional pipeline setups is a significant challenge, since YAML syntax mistakes, dependency conflicts, and resource provisioning difficulties are not always easy to identify and fix. Its absence of advanced validation tools results in configuration errors only being detected during pipeline execution, which causes failed runs and delays in deployment. The iterative configuration debugging and testing process may slow down pipeline development[15].

Change management also slows down with the increase in pipeline complexity, with traditional configurations necessitating a lot of manual review and testing of each change. The absence of automated configuration validation also implies that even the slightest alterations have a possibility of causing pipeline failures, which requires careful change management procedures that reduce deployment speed. Configuration change integration testing in multiple environments introduces further time overhead in the deployment process [52].

The complexity, expert knowledge dependencies, scaling limits and long deployment cycles undermine pipeline configuration reliability in many interdependent ways. The problem with the traditional MLOps pipeline paradigm, which is burdened with complicated manual setups, expert bottlenecks, scalability issues and slow release cycles, is that it essentially negates the qualities of transparency, accountability, resilience and reliability that define software trustworthiness.

The complexity of CI/CD systems makes them opaque. With pipelines composed of sprawling, hand-crafted YAML definitions and custom scripts, it becomes almost impossible to understand the system's behaviour to anyone but the original authors. This lack of transparency compromises trustworthiness, because stakeholders cannot determine how the pipelines function or how correctly.

4.3. Template-Based Low-Code Approach with Natural Language Elements

Our approach leverages template-based natural language specifications as a low-code approach to generate the GitLab pipeline. The template provides a structured format that can be augmented with natural language text for interpretation by the LLM. Templates enhance input consistency and reduce errors. Using our templates with LLMs improves the accuracy and dependability of the generated results. It combines the advantages of human-readable formats with the precision of machine-generated outputs for greater reliability.

Our research focuses on the trustworthiness of software, particularly in the context of RL models and LLMs used to build these RL systems. LLMs often produce inaccurate output, which is detrimental to building trustworthy software. Using LLMs to generate computer code (such as MLOps pipelines) also carries the risk of runtime errors in the generated configuration file and embedded scripts, leading to unreliable systems.

Our novel approach addresses trustworthiness by using LLMs to automate the creation of MLOps pipelines. Using LLMs, we aim to reduce errors when creating pipelines manually. This contribution is essential for trustworthiness. The "low-code" aspect makes the process accessible to people without in-depth DevOps knowledge.

Using LLMs to generate computer code (such as MLOps pipelines) also carries the risk of runtime and security errors in the generated configuration file and embedded scripts, which can lead to unreliable systems. Our system tackles this factor by including error detection and correction mechanisms to foster trust in our solution. Our approach is based on a low-code and template-based implementation using the Pipes and Filters pattern for modularity to steer and validate the LLM's results, meaning that the pipelines generated by the LLM are not only created and deployed, but also actively checked for correctness before they are used to train and deploy the RL model, contributing to both aspects of trustworthiness. This validation and correction strategy directly increases trustworthiness by minimising errors in the underlying infrastructure and issues with LLM-based code generation.

Table 1 describes the main sections of the template. This structured template acts as a detailed guide for creating the RL pipeline. All pipeline details can be specified using the template. By offering low-code components within a template

[11] `https://about.gitlab.com/blog/parent-child-vs-multi-project-pipelines`

[12] `https://adaptavist.com/blog/top-three-challenges-to-scaling-cicd-in-the-enterprise`

[13] `https://dev.to/zenika/gitlab-ci-10-best-practices-to-avoid-widespread-anti-patterns-2mb5`

[14] `https://about.gitlab.com/blog/build-an-ml-app-pipeline-with-gitlab-model-registry-using-mlflow`

[15] `https://about.gitlab.com/blog/guide-to-ci-cd-pipelines`

Table 1: Descriptions of the Low-Code Template's Sections

Section Header	***Description***
`Usage`	Example of how to run the scripts of the RL example. Based on this, the LLM should determine how to run the stages (assuming reasonable naming of the scripts and their parameters).
`Required Files`	List of essential files needed for the project, such as ML, testing and environment setup scripts. These should be used and adapted for the project's training, evaluation, testing and deployment.
`Requirements`	The Python dependencies required for the project.
`Environment Variables`	Environment variables necessary for the execution of the project, along with their intended default values, e.g. to specify how long the training should run for testing and production training.
`Gitlab CI pipeline stages`	The required stages of the CI/CD pipeline, such as `unit_test`, `train`, `model_test`, `eval` and `deploy`.
`Gitlab CI stage requirements`	Specific requirements for each CI stage, such as artefact generation in the `train` stage and dependencies between stages. Provided as natural language text.
`Gitlab CI pipeline results`	Indicates whether the pipeline generates certain results, such as artefacts or Docker images. The generation of artefacts can lead to substantially different pipelines, e.g. if Docker images, Docker-in-Docker or a Docker-enabled pipeline runner are required.
`Details of the deploy stage`	Information on the deployment process, including the target server, user access credentials and deployment tasks to be executed on the server.

along with natural language specifications, we can restrict the LLM to the necessary tasks and outputs. Using low-code components enables us to effectively direct validation tasks whilst allowing the LLM to comprehend and produce the required configurations and scripts, going beyond a simple template-based method. This strategy optimises the creation of the pipeline and clearly defines all essential components and dependencies.

4.4. Proposed Workflow

Figure 3 shows how a practitioner with a specific project can use our implemented approach. In the first step (**Write Initial Pipeline Specification**), the user writes an initial version of the pipeline specification based on our template, described in Section 4.3. The practitioner runs our implementation on their project and specification file (**Run Tool on Project to Generate and Deploy Pipeline**). During this step, our program creates the GitLab CI/CD pipeline configuration file based on the specification, which is then committed automatically to a GitLab repository. Optionally, the practitioner can then manually inspect the generated pipeline configuration (**Manual Inspection of Generated Pipeline Configuration**) whilst also verifying that the pipeline was deployed and executed and that it functions correctly (**Check Pipeline Deployment and Execution**). If necessary, the user can then make corrections to the pipeline specification (**Minor Corrections to Pipeline Specification if Necessary**) and rerun our program if a new pipeline configuration needs to be generated.

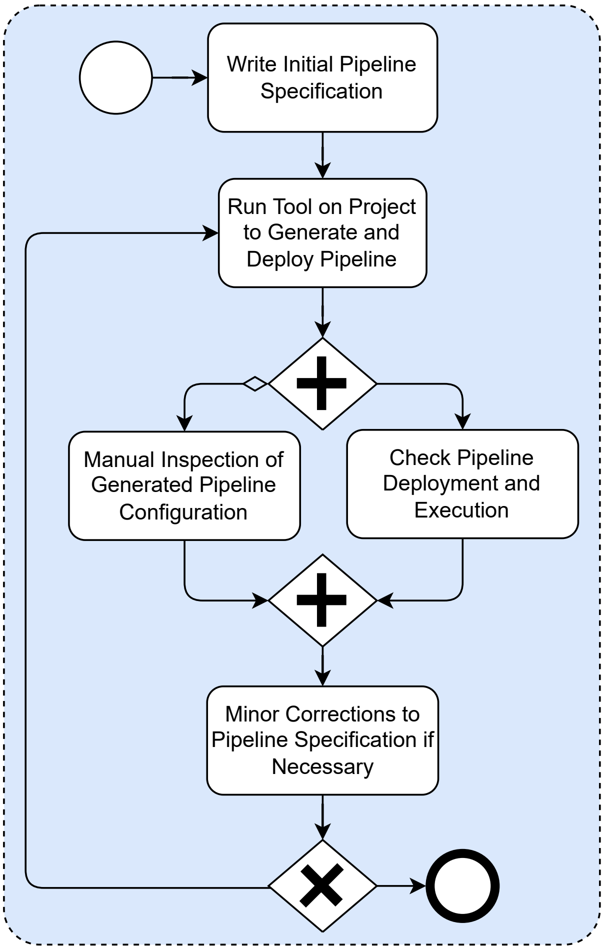

Fig. 3: The proposed approach's workflow.

The resulting workflow therefore incorporates an iterative feedback loop, allowing the user to incrementally, iteratively and continuously develop the pipeline using our tool and refine the pipeline specification by repeating the process. The optional human-in-the-loop manual inspection and approval helps to enable trust in the results generated by the LLM and accountability on the part of the human operator. In reality, it is unlikely that many iterations would be needed, and in Sections 6 and 7 we provide results showing that if a suitable LLM is used, only very few minor corrections might be required.

4.5. Illustrative Example of the Approach in Action

To illustrate our approach, we present an example of generating and deploying an RL pipeline using a generation flow. An example of a concrete pipeline specification according to the template can be found below.

```
Usage:
python snake_rl.py
    ?train|eval|unit_test|model_test?

Required Files:
snake_env.py, snake_learn.py, snake_play.py,
snake_rl.py, snake_unit_test.py,
snake_test_model.py, snake_eval.py

Requirements:
gymnasium, numpy, opencv-python,
    stable-baselines3, tensorboard

Environment Variables:
ITERATIONS 2
RENDER_ASCII "True"
MODEL_NAME "LATEST"
MODELS_DIR "models/PPO-T4-1"
LOGS_DIR "logs/PPO-T4-1"
NUM_EPISODES 20

Gitlab CI pipeline stages:
unit_test, train, model_test, eval, deploy

Gitlab CI stage requirements:
- train creates artifacts at location MODELS_DIR
- eval job needs train job and its artifacts
- model_test job needs train job and its
    artifacts

Gitlab CI pipeline generates artifacts:
Yes, it creates artifacts for the models in the
    train stage

Gitlab CI pipeline generates docker images:
No

Details of the deploy stage:
- Deployment to a local server "127.0.0.1".
- The server is accessed by the user "user".
- The private key is stored in the Gitlab
    variable SSH_PRIVATE_KEY.
- Deployment tasks:
    - Secure copy the files in MODELS_DIR to a
    directory "deploy_test"
    - Run the model_test again on the server as
    a smoke test
```

The requirements for generating the RL pipeline include a sample command for running the RL example's main script, a list of essential files in the project to be used in the various pipeline stages, the Python dependencies of the project and the necessary environment variables with default values. Next, the GitLab CI/CD stages *unit_test*, *train*, *model_test*, *eval* and *deploy* are required in that order. The generation process should determine how to run the corresponding RL scripts. The *train* stage should provide models as artefacts for evaluation, model testing and deployment. Finally, deployment details specify the server and the private key (provided through a GitLab environment variable). Deployment commands for the *deploy* stage are also specified in natural language text.

Our approach generates a CI pipeline file from a completed template, creates a detailed report of each flow step as a YAML file, assembles the project files in the generation directory and commits the whole project to GitLab. GitLab then runs the pipeline stages and, if successful, will test the code, train the models, test and evaluate the models, deploy the trained models to the deployment server and run the model test on the server as a smoke test.

4.6. Prompting Method

We use *few-shot prompting* [11] for each generation, detection and validation step. This method helps with in-context learning by including examples in the prompt to direct the LLM toward better performance and enables the LLM to produce results that are close to the desired outcome. When retries are needed, we provide the LLM with the previous conversation, including the generation history and any detected errors or warnings, which aids the LLM in refining its output to conform to the desired specification. In this manner, we implement *prompt chaining* [11] as a secondary prompt engineering technique. This structured approach ensures that each component of the generation flow is accurately produced and validated, leading to reliable and efficient pipeline generation.

Figure 4 depicts a conversation template that describes how our approach interacts with an LLM to generate a CI/CD pipeline configuration based on a user-authored low-code specification. Initially, our implementation specifies the **system** role when preparing the API call to the LLM, corresponding to **Role: System** in the **System Message Setup** box in Figure 4. Our implementation also prepares a system message instructing the LLM to create a GitLab CI/CD YAML file, as shown in the snippet below, which corresponds to the **Context Initialisation** in the **System Message Setup** box in Figure 4. Together with the **system** role, the context initialisation is used to set the context for the instructions that are to follow and is as follows:

```
You are a helpful code assistant that can teach
    a junior developer how to code. Your
    language of choice is Python, and you want
    to create a Gitlab CI/CD yml file for your
    project.
Don't explain the code, just generate the code
    block itself.
```

In the **Instruction with Sample Input** box in Figure 4, our implementation specifies the **user** role (**Role: User** in Figure 4). Setting the role to **user** indicates to the LLM that the message contains a request from the user that the LLM should respond to. Our tool also provides an instruction with an example pipeline specification (**Instruction Formulation with Sample Pipeline Specification** in Figure 4) as in the following snippet:

```
Write a Gitlab CI/CD yml file for Python
    machine-learning scripts.
Do not use Docker inside the Gitlab CI/CD
    pipeline. Use the following context:

Usage: python train_rl.py

Required Files:
```

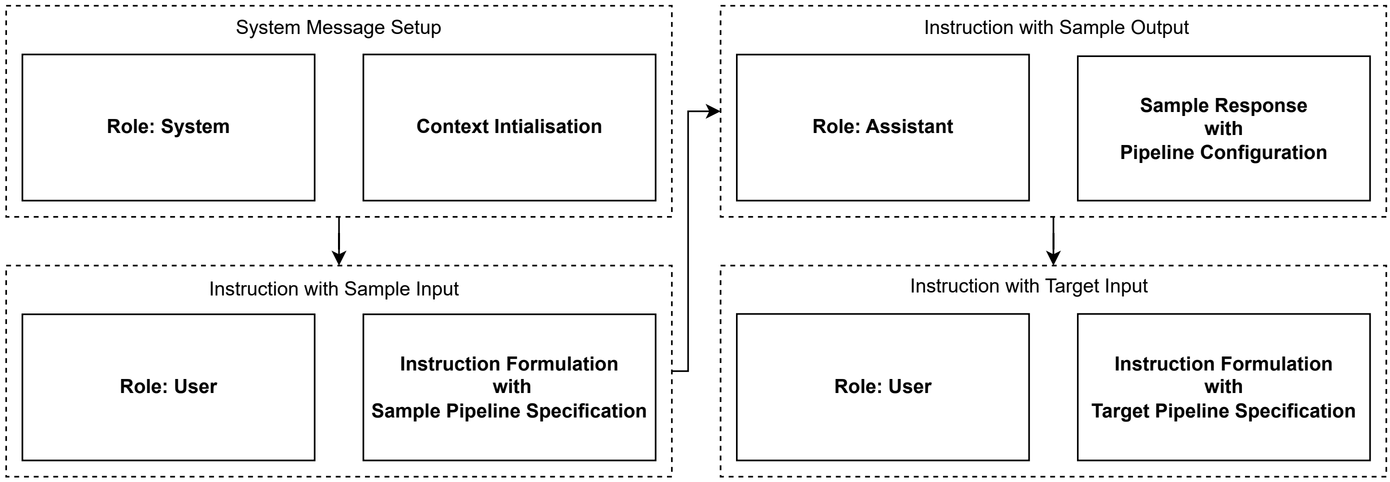

Fig. 4: The approach's conversation template.

```
train_rl.py
eval_rl.py
test_rl.py

Requirements:
stable-baselines3>=2.1.0
supersuit>=3.9.0

Gitlab CI/CD pipeline tasks:
train, eval, test

Gitlab CI/CD pipeline generates artifacts:
Yes, it creates artifacts for the models in the
    train stage

Gitlab CI/CD pipeline generates docker images:
No
```

We next set the role to **assistant** (**Role: Assistant** in the **Instruction with Sample Output** box in Figure 4), which communicates to the LLM that the text we are about to provide is to be considered a previous response from the LLM. Doing so makes the kind of response we expect to receive clear to the LLM. Specifically, we provide a sample output response for the sample input pipeline specification described previously (**Instruction with Sample Input** in Figure 4). This sample response contains a corresponding GitLab CI/CD pipeline configuration and is denoted by **Sample Response with Pipeline Configuration** in Figure 4. The sample response is as follows:

```
image: python:3.9-bullseye

stages:
  - train
  - eval
  - test

before_script:
  - python --version
  - pip install --upgrade pip
  - pip install stable-baselines3>=2.1.0
    supersuit>=3.9.0

train:
  stage: train
  script:
    - python train_rl.py
  artifacts:
    paths:
      - path/to/model/directory/
    expire_in: 1 week

eval:
  stage: eval
  script:
    - python eval_rl.py

test:
  stage: test
  script:
    - python test_rl.py
```

Our implementation selects the **user** role again, indicating that we expect a response from the LLM (**Role: User** in **Instruction with Target Input** in Figure 4). We provide the same instruction as in the **Instruction with Sample Input** part of Figure 4 (`Write a Gitlab CI / CD yml file for Python machine-learning scripts. Do not use Docker inside the Gitlab CI / CD pipeline. Use the following context:`), but instead of providing a sample pipeline specification, this time we give a target pipeline specification for which we want to generation a real pipeline configuration (i.e. the contents of a GitLab CI/CD YAML configuration file). The target specification takes the same format as the sample specification provided in **Instruction Formulation with Sample Pipeline Specification** in Figure 4 and described in Table 1 in Section 4.3. This part corresponds to **Instruction Formulation with Target Pipeline Specification** in **Instruction with Target Input** in Figure 4.

The tool then constructs the conversation by combining the system message, the instruction with the example low-code, template-based pipeline specification, an example GitLab CI/CD YAML file response and the instruction with the target pipeline specification from which to generate the GitLab CI/CD YAML file configuration. This structured approach provides unambiguous instructions and examples to the LLM to increase the chance of receiving an acceptable response.

Our program checks the LLM's response for GitLab CI/CD file linter issues, artefact creation and stage order correctness, and if errors or warnings occur, continues the conversation with the LLM by providing a **role: user** instruction to correct the detected errors and warnings in the generation configuration. It retries this generation process up to a predefined number of times (three, by default) and finally opts for the best response based on the number of errors and warnings encountered for each response. Doing so gives the LLM multiple opportunities to correct its output and allows our tool to select the best response out of the ones that the LLM proposed.

4.7. Generation Architecture

This section details the architecture of our MLOps pipeline generation approach. We describe the primary components, classes, and their roles within the system below. Figure 5 shows the main high-level components of the architecture and Figure 6 shows the base classes containing the implementations for flows, code generation, validation and committing the generated results.

4.7.1. Components

Generator, Detector and LLM Components: `LCFExecutor` in Figure 5 is the central component responsible for executing the generation flow or running a detection flow. It uses instances of `LLM`, which wrap supported LLMs based on an `LLMDefinition`. An `LLMDefinition` specifies the model name, model type and parameters like token limit, minimum response tokens and temperature. The `LLM` component abstracts the functionalities of specific LLMs and facilitates the generation of textual artefacts based on provided specifications.

Pipes and Filters Architecture: The `LCF` component Pipes and Filters orchestrator in Figure 5 is responsible for executing `LCFComponents`, which represent different steps in the Pipes and Filters pattern involved in generation, detection and retrying. The `LCF` orchestrator also provides a `LCFDataTransferObject`, which is passed through the various steps in the Pipes and Filters pattern architecture. It carries the generation report that is gradually assembled during the generation, detection and validation steps, metadata such as project and generation directories and the current conversation. This piping mechanism is based on passing a Data Transfer Object [53] through the implementation's filters. A `GitlabWrapper` component interacts with the GitLab server via the server's API for validating the generated CI/CD pipeline configuration files and uploading the generated pipeline configuration to GitLab.

4.7.2. Classes

Pipes and Filters Classes: Figure 6 shows the `LCFComponent` class, the base class for all classes involved in the generation and validation processes. Thus, its subclasses are the Pipes and Filter pattern architecture filters. `LCF` orchestrates a sequential flow of components, managing a list of `LCFComponent` components to define the sequence of operations, and an instance of `LCF` must be passed to either a generator or a detector for execution.

`LCFParallelFlow` subclasses `LCF`, managing the parallel execution of multiple flow steps and providing mechanisms for parallel validation via its subclass `ParallelDetection`, which is used for parallelising detection flows. `ParallelDetection` itself has two subclasses: `ParallelSuccessfulIfAllSuccessful` validates multiple tasks in parallel and is successful if all steps are successful, and `ParallelSuccessfulIfAnySuccessful` validates multiple tasks in parallel and is successful if any are successful.

`GenerationRetry` is also a subclass of `LCFComponent`. It retries generation steps a specified number of times, recording errors and warnings. It stops at the first attempt without errors or warnings; if no such attempt occurs, it takes the best results after the specified total number of attempts (three, by default).

`GitlabCommitGeneratedProject` commits the generated pipeline and its project to GitLab using the `GitlabWrapper` class. `GitlabWrapper` provides an interface to interact with a GitLab server's API for committing projects and validating GitLab pipelines.

Thus, the generation flow incorporates flexible components for either parallel or concurrent validations (any or all of which must be successful, depending on the user's needs), retrying the generation, and committing and pushing the project to GitLab.

Generation Classes: The `GenerationRetry` class shown in Figure 6 uses the `LCFGeneration` class, which is a base class used for performing generation tasks. It uses a `LCFDataTransferObject`, which in turn utilises the `Conversation` class, to manage interactions with the LLM by abstracting individual conversation instances with messages to and responses from the LLMs, maintaining the context and state of ongoing interactions. The `GenerateGitlabCIYmlFileFromSpecFile` class subclasses `LCFGeneration` and is used to generate GitLab CI/CD YAML files from low-code specifications.

Detection Classes: Figure 6 shows the `LCFDetection` class, another subclass of `GenerationRetry`, which performs the detection used by validation tasks in generation workflows. It ensures that generated artefacts meet specified quality and correctness criteria for generation tasks. Like `LCFGeneration`, `LCFDetection` also makes use of a `LCFDataTransferObject`, which is passed through the Pipes and Filters flow with a `Conversation`.

Validation Classes: Figure 6 also includes various subclasses for performing individual validations. Several classes that subclass `LCFDetection` implement validation logic for use during detection. For instance, `YmlSyntaxIssueDetection` is the base class for detecting issues in YAML files. Its subclass, `GitlabCICheckStageOrder`, ensures the stages in the GitLab CI pipeline are in the correct order, with the `GitlabCICheckStageOrderFromSpecFile` class validating the stage order based on the low-code specification input. `GitlabCINoLinterIssuesDetection` uses a `GitlabWrapper` to validate the pipeline remotely via a GitLab server API call. `LCFCodeBlocksDetection` has a subclass, `GitlabCIArtifactsCreatedDetection`, that detects whether artefacts are created in the generated pipeline configuration.

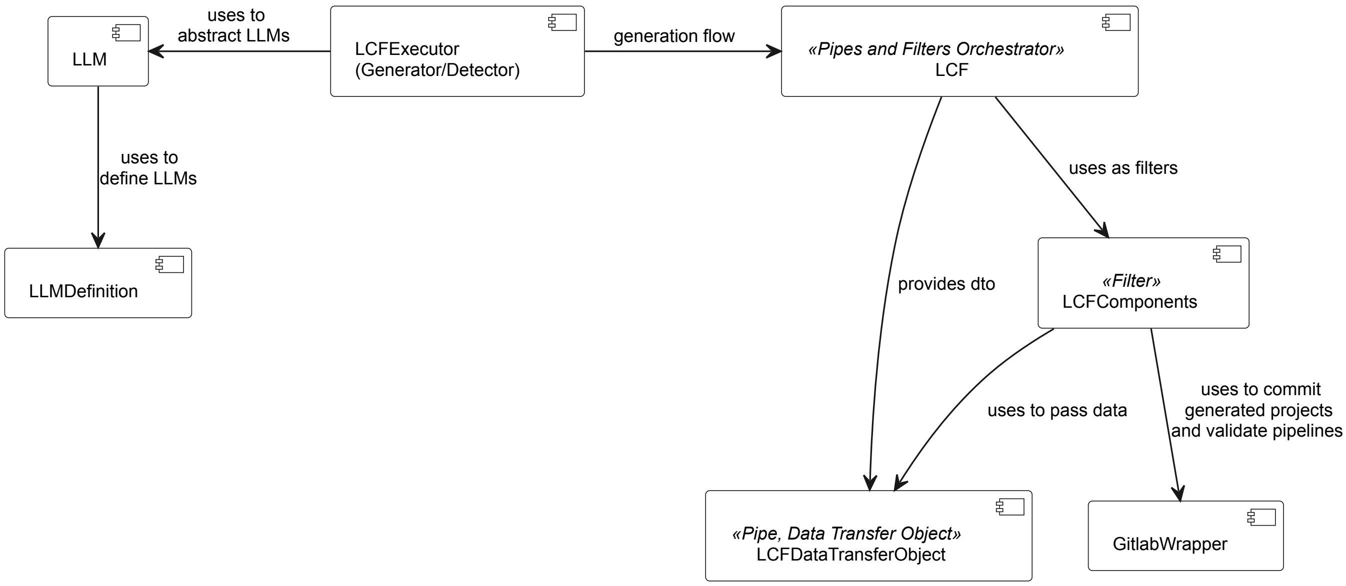

Fig. 5: The main components of the generation architecture.

4.8. Generation Flow Specification Approach

Using LLMs for code generation and validation presents accuracy challenges such as AI hallucinations and incorrect classifications [40–42]. Our approach employs a structured generation flow with focused steps and contextual feedback to address these issues and ensure reliable pipeline generation.

The generation flow specification for our approach is defined using a simple Python DSL. This specification guides the LLM through highly specific and focused generation, detection and validation steps to ensure high accuracy in the generated outcomes. An example of such a generation flow specification is shown below:

```
LCF(
  components=[
    GenerationRetry(
      GenerateGitlabCIYmlFileFromSpecFile,
        ParallelSuccessfulIfAllSuccessful(
          [
            GitlabCINoLinterIssuesDetection,
            GitlabCIArtifactsCreatedDetection,
            GitlabCICheckStageOrderFromSpecFile
          ]
        )
      ),
      GitlabCommitGeneratedProject,
  ],
)
```

In the flow above, we specify that we wish to generate a GitLab CI/CD YAML file from a specification file (i.e. a low-code, template-based specification that describes the CI/CD stages and steps). We then wish to run various error detection measures in parallel: linting the generated GitLab pipeline configuration file, checking that the pipeline configuration creates artefacts, and that the pipeline has the correct order of stages. Finally, assuming all three detections passed, we wish to commit and push the project with its generated pipeline configuration to GitLab, where the pipeline is executed according to the generated pipeline configuration.

5. Evaluation

We selected three open-source RL projects (see Section 5.1) and developed pipelines for each project, which were applied in three different scenarios (see Section 5.2) and multiple iterations. Each scenario has increasingly complex requirements to simulate real-world pipeline practices. This approach allows us to thoroughly evaluate our method's versatility, adaptability and scalability in addressing different pipeline requirements. By focusing on various scenarios for each project, rather than simply increasing the number of projects, we capture a wider range of potential use cases and challenges and ensure that our approach is robust and effective in meeting standard and unforeseen requirements in real-world applications.

5.1. Study Objects

Ideally, we would select a set of open-source RL projects, each of which already includes an MLOps pipeline. If these pipelines collectively contained all the pipeline stages our approach can generate, we could use them as ground truth to assess the generation results. After an intensive search on GitHub and the Web, we found no open-source projects that fulfilled these requirements.

We then searched on GitHub, Google and Bing, with search terms including "(reinforcement learning | RL | multi-agent reinforcement learning | MARL | (Tensoflow & RL) | (Keras & RL) | RLLib | StableBaselines) & (MLOps | DevOps | CI/CD | Continuous Integration | Continuous Deployment | Continuous Delivery)". We found that there were not many repositories that combined these two fields.

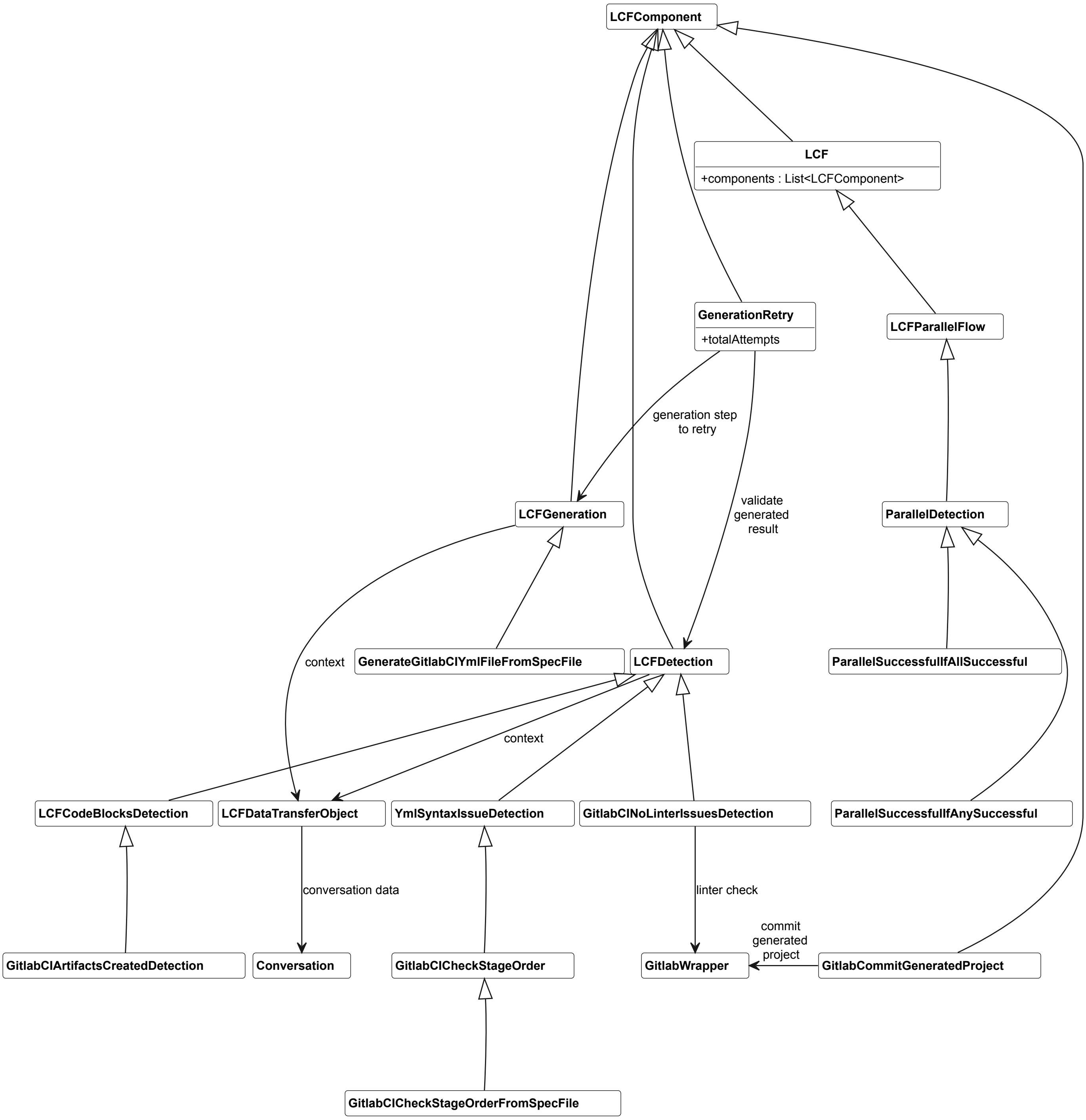

Fig. 6: The base classes for flows, generation, detection and commits in `LCFComponent`.

As a second-best option to create the ground truth, we selected three popular RL projects and realised MLOps pipelines and various tests for them. We used the first part of our search string only (before the "&") and the following inclusion/exclusion criteria:

1. Projects must primarily focus on RL.
2. Projects have well-documented and clearly explained code, reflecting the typical setup data scientists use for RL projects.
3. Projects follow different approaches to constructing RL projects.
4. All scenarios described in Section 5.2 are fully applicable.

We deliberately opted for several small rather than many large projects, as the specific RL implementation plays a secondary role in our evaluation. Instead, integrating these projects into each of the evaluation scenarios presented in Section 5.2 increases the complexity of the evaluation cases. Our approach focuses on the MLOps components and treats the RL code primarily as a black box integrated into the pipeline stages – we are interested principally in MLOps at the level of pipeline architectures in this study and not the specifics of the implementations. Many larger, existing RL projects failed to meet inclusion/exclusion criterion number 4 ("All scenarios described in Section 5.2 are fully applicable"), meaning that only relatively trivial MLOps pipeline generation tasks would have been tested. Table 2 summarises the projects we included in our study. The full results are reported in the tables in Appendix A.

The three projects were selected because they offer sufficient diversity in RL approaches whilst maintaining the rigour needed to perform a thorough evaluation using the evaluation scenarios. This focused approach allows us to conduct comprehensive, scenario-complete evaluations that would not be possible with a larger set of projects that only partially meet our evaluation criteria. The breadth vs. depth trade-off aligns more with our research objectives; we are trying to validate MLOps generation capabilities, not survey all existing RL project structures. To show scalability and applicability beyond the three projects, we demonstrate how our MLOps pipeline generation can be integrated with larger production-scale RL deployments in an industry application described in Section 8. This setting is a much more complex RL implementation than our evaluation projects, with multi-agent systems, distributed training architectures and enterprise-grade deployment requirements.

Table 2: Descriptions of the Open-Source Projects

Name	*Description*
KAZ	Stable Baselines 3 realisation of RL training, prediction, visual play and evaluation for the Petting Zoo Knights, Archers and Zombies environment. It uses the Proximal Policy Optimisation algorithm, parallel environments, and model saving/loading, and it can be customised using many parameters. Original implementation: `https://github.com/Farama-Foundation/PettingZoo/tree/master/pettingzoo/butterfly/knights_archers_zombies`
SNA	Stable Baselines 3 realisation of RL training, prediction, visual play and monitoring for a custom snake game environment. It uses the Proximal Policy Optimisation algorithm, model saving/loading, checkpoints, TensorBoard and a custom environment. Original implementation: `https://youtube.com/watch?v=XbWhJdQgi7E&list=PLQVvvaa0QuDf0O2DWwLZBfJeYY-JOeZB1`
SHO	Keras/TensorFlow-based realisation of RL training, prediction and evaluation for a custom shower simulation environment. It uses the Deep Q-Network algorithm with a Boltzmann Q policy, a sequential model and a custom environment. Original implementation: `https://github.com/nicknochnack/OpenAI-Reinforcement-Learning-with-Custom-Environment`

5.2. Scenarios

We define three evaluation scenarios, each subsequent scenario building on the previous one, for representing different MLOps pipeline practices, as well as the different kinds of generation requirements (precisely specified and additional natural language-based specifications):

1. The existing basic functionality of the projects used as objects of our study is realised as an MLOps pipeline. This pipeline usually consists of training and evaluation stages with several specific Python requirements, requiring setting environment variables and generating an RL model as an artefact in the training stage. Each generation requirement is precisely specified using a specification file such as the one in Section 4.5. Additionally, under the section `Gitlab CI stage requirements` in the specification, one natural language requirement is added that the training stage (`train`) creates artefacts in a models directory and another one that the evaluation stage (`eval`) requires these artefacts (only if an evaluation stage is defined).
2. In addition to all generation requirements of the basic scenario, we require a deployment stage (`deploy`) and Docker images as container artefacts. We further detail this requirement under `Details of the deploy stage` in two natural language requirements: firstly, the deployment to a Docker container provided as an artefact by the `deploy` stage is required. Secondly, we require the `deploy` stage to use Docker in Docker (DIND) to create the Docker images. We specify as further natural language requirements under `Gitlab CI stage requirements` that the evaluation job needs the training job and its artefacts. We also require an environment variable *MODEL_DIR* specifying the model directory.
3. In this scenario, we use all generation requirements of the basic scenario and the natural language `Gitlab CI stage requirements` from the second scenario. We re-

quire five stages: `unit_test`, `train`, `model_test`, `eval` and `deploy`. We manually wrote unit and model tests for each project. These are placed in additional files that need to be executed by the pipeline in the respective stages. Under `Gitlab CI stage requirements`, we add another natural language requirement: the `model_test` job needs the `train` job and its artefacts. No Docker image creation is required. Deployment host and user names are provided under `Details of the deploy stage`. Further, it is stated that the private key to be used is stored in the GitLab variable *SSH_PRIVATE_KEY* and that secure copy is to be used as a deployment task to copy the artefact in *MODEL_DIR* to a directory on the host.

We executed each scenario three times for each project, totalling twenty-seven evaluations per LLM. Our choice of evaluation scenarios was driven by the need to comprehensively evaluate MLOps pipeline generation over the most critical and commonly needed pipeline stages in RL deployments. The three scenarios described in this section were carefully designed, based on our understanding of RL and informed by discussions with our industry partner, to capture fundamental MLOps practices when building CI/CD pipelines. These scenarios, and their assorted evaluation variables, include the key pipeline components - model-training orchestration, hyperparameter-optimisation workflows, and deployment automation - where MLOps tools provide the most value.

We considered expanding and customising the scenarios to cover a broader scope of RL projects, but decided that doing so would dilute the evaluation's validity and depth. Such evaluation scenarios would not have tested our solution's ability to produce advanced pipeline stages that reflect the real complexity of production RL deployments. We focused on depth and real-world applicability over sample size, so our evaluation represents real MLOps challenges and not examples customised to existing projects.

5.3. Evaluation Variables

We use twenty evaluation variables to analyse the generated MLOps pipelines in our evaluation process; all variables have Boolean values. Some variables have the value "NA" if not required by the evaluation scenario's generation requirements.

We evaluated the following generation-specific variables by inspecting the generated pipeline configuration files:

- *GPI*: The generated script uses a correct Python image.
- *GAS*: The generated script contains all required stages.
- *GSO*: The required stages are in the correct order.
- *GAR*: All Python requirements and the necessary Linux packages are correctly installed for all required stages.
- *GEN*: All specified environment variables are correctly set.
- *GMS*: All model artefacts are correctly created.
- *GCD*: Required Docker images are correctly created.
- *GDA*: All Docker artefacts are correctly created.

We evaluated the following variables by inspecting the code to check whether the natural language text requirements under `Gitlab CI stage requirements` and `Details of the deploy stage` are correctly realised. Note that many other details could be provided in natural language; in our evaluation, to systematise our study, we limit ourselves to the following cases:

- *NTA*: The `train` job creates artefacts in a specific directory.
- *NET*: The `eval` job needs the `train` job and its artefacts.
- *NMT*: The `model_test` job needs the `train` job and its artefacts.
- *NSK*: An SSH key setup script is correctly created to reach the deployment server.
- *NSC*: Secure copy is used to perform the deployment with the correct user/key, host and directory.
- *NDA*: The `deploy` stage provides Docker results as an artefact.
- *NDI*: Docker in Docker (DIND) is realised, and the correct image is selected.

Finally, we evaluated the following runtime variables to determine whether a pipeline job ran successfully or not.

- *RUT*: Unit tests ran successfully.
- *RTR*: Model training ran successfully.
- *RTE*: Model tests ran successfully.
- *REV*: Model evaluation ran successfully.
- *RDP*: Model deployment ran successfully.

We executed each scenario three times for each study object and each LLM. We then calculated the error rate for each evaluation variable *var* as, with $n = 3$ (three iterations) as given by Equation 1:

$$\text{ErrorRate}^{var} = \frac{1}{n}\sum_{i=1}^{n} \text{ErrorRate}_i^{var} \tag{1}$$

We also calculated the overall error rate over all precisely specified generation variables (*Overall Error GS*), all variables relating to natural language-based requirements (*Overall Error NL*), all variables associated with running the jobs (*Overall Error RJ*) and all variables (*Overall Error*).

5.4. Models

We chose seven popular and current LLMs for evaluating our approach: OpenAI GPT-3.5 Turbo, OpenAI GPT-4o, Qwen2.5 72B, Qwen2.5 Coder 32B, Llama 3.3 70B, Code Llama 70B and DeepSeek R1 70B. We deployed and ran the two OpenAI models on Microsoft Azure[16]. We deployed and ran the five freely available models in Ollama[17] on a Cisco UCS C245 M6 server with an NVIDIA A100 80GB GPU, two AMD EPYC 7543 32-core processors and 512GB of RAM.

The five Ollama models are of particular interest because they represent a growing cohort of small, performant LLMs which demonstrate reasonable capabilities in terms of solving tasks but are freely available and deployable locally thanks to their relatively low memory footprint and flexible parameter size, which can be selected based on local hardware capabilities. These locally-executable LLM types expand the range of applied model types beyond that of the GPT family.

The LLMs were selected from the standard benchmark EvalPlus [54][18], which is based on peer-reviewed studies and provides a ranked list of code-generation LLMs, including versions of the ones we use in this study. Where a newer or higher-parameter version of a model than the one in the EvalPlus leaderboard was available, then, within the constraints of our hardware, we selected that version. We only selected popular models, i.e. models with over one million pulls on `https://ollama.com`. Furthermore, similarly to how we chose GPT-3.5 Turbo and then an enhanced version in GPT-4o, we selected base models (Qwen2.5 72B and Llama 3.3 70B) and corresponding coding-specific related models (Qwen2.5 Coder 32B and CodeLlama 70B). We also included DeepSeek R1 70B due to its novelty and popularity.

The LLMs are widely used in related but differing tasks and have varying characteristics, such as speed, contextual awareness and training data. Even though it may appear obvious that some LLMs are likely to perform much better than others, this can only be confirmed by evaluating a specific application. Evaluating a selection of LLMs enables us to compare how different LLMs impact the results of our approach.

GPT-3.5 Turbo[19] is a widely adopted, efficient model from OpenAI. GPT-3.5 Turbo delivers an effective baseline for assessing our approach because it strikes the right equilibrium between operational efficiency and performance quality. GPT-3.5 Turbo is a standard reference model to check progress in new technologies or customised coding solutions. The model builds upon previous versions by making advancements, although it sometimes exhibits diminished mathematical capabilities. We included GPT-3.5 Turbo as a model for assessment since recent benchmarks demonstrate its comparable performance to the top non-commercial LLM models but with substantially reduced costs compared to GPT-4o.

GPT-4o[20] from OpenAI delivers enhanced speed and efficiency features and is one of the best models in the EvalPlus performance benchmark. The model exhibits powerful general capabilities, which make it ideal for performing state-of-the-art code generation. The proficient capabilities of GPT-4o extend to basic coding and general awareness tasks, but this advanced performance level comes at a greater cost than other models. Our evaluation also adopts GPT-4o because of its superior capability to understand and generate natural language, forming our methodology's core foundation.

Qwen2.5 72B[21], created by Alibaba Cloud, has over 4.1 million pulls as of 2025-02-14 and a size of 47GB. We use Qwen2.5 in the evaluation to assess the effectiveness of our approach with a model that demonstrates superior multilingual skills, enhanced coding capabilities, advanced instruction following and structured text data understanding and generation support.

Qwen2.5 Coder 32B[22] by Alibaba Cloud has over 2.2 million pulls as of 2025-02-14 and a size of 20GB. This model has been ranked highly on various popular code generation benchmarks, and its performance is comparable with GPT-4o. Qwen2.5 Coder 32B can also fix code errors, which is relevant to our approach's error-handling aspect. The model supports over 40 programming languages.

Llama 3.3 70B[23] is a leading open-source model from Meta. It is widely used, with over 1.3 million pulls as of 2025-02-14 and is 43GB in size. Llama 3.3 70B performs similarly to Llama 3.1 405B but with fewer parameters. It demonstrates particular strength when tasks require extended context, like document analysis and conversations spanning multiple exchanges, making it a suitable model for evaluating our approach.

Code Llama 70B[24] from Meta, with over 1.7 million pulls as of 2025-02-14 and a size of 39GB, is well-suited to this study since it has been specifically trained for code generation tasks. The model supports most popular programming languages, including Python, C++, Java, PHP, TypeScript, JavaScript, C# and Bash.

DeepSeek R1 70B [55][25] is a new LLM released in January 2025 with over 15.6 million pulls as of 2025-02-14 and a size of 43GB. We chose this LLM due to its novelty and the high benchmark ranking of its predecessor (DeepSeek V3 from November 2024) and opted for a parameter size of 70B, which is a distilled model based on Llama3.3:70B. Early benchmarks suggest it should be a high-performing model and evaluation results by DeepSeek and EvalPlus suggest that its base model (DeepSeek V3) performs comparably to OpenAI O1[26]

[16]`https://azure.com`

[17]`https://ollama.com`

[18]`https://evalplus.github.io/leaderboard.html`. Also see `https://trustbit.tech/en/llm-leaderboard-juli-2024`, `https://klu.ai/llm-leaderboard`, `https://aimlapi.com/comparisons/chatgpt-4o-vs-o1-mini` and `https://the-decoder.com/gpt-4-crushes-other-llms-according-to-new-benchmark-suite` and links on EvalPlus for further examples of benchmarks.

[19]`https://platform.openai.com/docs/models/gpt-3.5-turbo`

[20]`https://platform.openai.com/docs/models/gpt-4o`

[21]`https://ollama.com/library/qwen:110b`

[22]`https://ollama.com/library/qwen2.5-coder:32b`

[23]`https://ollama.com/library/llama3.3:70b`

[24]`https://ollama.com/library/codellama:70b`

[25]`https://ollama.com/library/DeepSeek-r1`

[26]`https://openai.com/o1`

and OpenAI O1 Mini[27], which came first and second place on the EvalPlus leaderboard respectively.

6. Initial Results

In this section, we interpret the evaluation results listed in Tables A.6, A.7, A.8, A.9, A.10, A.11 and A.12 in Appendix A. Please note that, based on these results, we also implemented some improvements to the project and ran a new evaluation, which we report on in Section 7, the results of which are shown in Table A.13. A discussion of the implications in the context of the research questions can be found in Section 9.

6.1. Best and Worst-Performing LLMs

Table 3 summarises the best-performing LLMs by task type error rate, project and evaluation scenario, and overall error rates. The rows signify three different sets of error rates depending on the task type: *Overall Error GS*, which represents generation tasks' overall error rate, *Overall Error NL*, which denotes the overall error rate for natural language tasks, and *Overall Error RJ*, which encapsulates the overall error rate for runtime job tasks. The final row, *Overall Error*, is an aggregation of the aforementioned error rates over the task types (generation, natural language, and runtime) for each column. Except for the final one, the columns represent the error rates for the open-source projects *KAZ*, *SNA*, and *SHO* described in Table 2. Combined with the evaluation scenarios 1, 2 and 3 described in Section 5.2, represented as numerical suffixes. For example, *KAZ1*, *KAZ2* and *KAZ3* refer to the *KAZ* project and the three evaluation scenarios. The final column *Overall Error* gives the overall error rate across all projects and scenarios for a given task type.

For example, the row and column pair corresponding to row *Overall Error NL* and column *KAZ3* indicates that GPT-4o had the lowest overall error rate for natural language tasks for project *KAZ* and scenario 3. Row *Overall Error* and column *SHO1* show that Qwen2.5 Coder and Llama 3.3 achieved the lowest overall error rates across all task types for project *SHO* and scenario 1. The cell matching row *Overall Error RJ* and column *Overall Error* indicates that GPT-4o had the lowest overall error rate for runtime tasks across all projects and scenarios.

Overall, GPT-4o exhibited the lowest error rate (*Overall Error*) followed by Qwen2.5 Coder. GPT-4o also performed best for generation tasks (*Overall Error GS*), natural language tasks (*Overall Error NL*) and runtime tasks (*Overall Error RJ*), followed by Qwen2.5 Coder, Llama 3.3 and GPT-3.5 Turbo, respectively. Therefore, GPT-4o was the best overall performer in every category, with Qwen2.5 Coder coming second overall and performing particularly strongly in code generation tasks.

The worst-performing LLM overall was Code Llama, followed by DeepSeek. Code Llama performed worst for code generation tasks, followed by GPT-3.5 Turbo. Natural language tasks were performed worst by GPT-3.5 Turbo, followed by Code Llama. Llama 3.3 performed runtime tasks the worst, followed by DeepSeek. Hence, Code Llama performed poorly at code generation and natural language tasks.

In conclusion, our approach used GPT-4o and Qwen2.5 Coder to successfully generate GitLab CI/CD pipeline configuration files with low error rates. Note that the error rates reported in the tables in Appendix A were easily improved further by resolving the common errors described in Section 6.4 - see Section 7 for details.

It should also be noted that the results we report are likely an underestimation of the capabilities and potential of the LLMs in this application and an overestimation of the error rates, since if a pipeline stage failed, we also marked all subsequent pipeline stages as having failed. In reality, some of these pipeline stages would have succeeded, assuming the error in the previous stage had not occurred. Also, it was often the case that multiple errors were related and had a single cause. For instance, secure copy during deployment might have failed due to the SSH key not being set up correctly, or a runtime error could have resulted from an incorrect value in the pipeline configuration caused by a generation error. Both scenarios would increase the error rate for multiple evaluation variables but only require a single fix.

6.2. Evaluation Variables

Across all seven LLMs, some common patterns emerged in the results when handling the tasks associated with the evaluation variables, particularly when considering what subtasks the LLMs were capable of performing reliably and where they often made mistakes.

Generally, and as might be expected, simpler tasks yielded lower error rates, whilst more complex ones resulted in higher error rates. The tasks that most of the LLMs performed most correctly were the generative tasks and a large proportion of the natural language processing tasks. On the other hand, runtime tasks were more challenging for all of the LLMs. In Section 6.4, we discuss common errors in output that occurred, which were easily resolved by providing slightly more (and more specific) information to the LLM.

Generative tasks that the models mostly completed with low error rates included, for instance, using the correct Python image in the generation pipeline configuration file, including all the required stages in the proper order, and installing necessary Python and Linux requirements. Some deviation in behaviour occurred when specifying environment variables and the correct Docker artefacts.

Several LLMs had difficulties with natural language tasks like setting up the SSH key for model deployment, using secure copy to perform the model deployment, realising the use of Docker in Docker, selecting the correct Docker image, and providing the generated Docker image as a result. Problematic runtime tasks for several models were running unit tests successfully, running model tests successfully and deploying the model successfully. Failure of any of these stages would result in the subsequent stages being skipped and marked as failing. On the other hand, some natural language tasks, like making

[27] `https://openai.com/index/openai-o1-mini-advancing-cost-efficient-reasoning`

Table 3: Summary of the Best-Performing LLMs by Lowest or Equal-Lowest Error Rate (Initial Results)

	KAZ1	***KAZ2***	***KAZ3***	***SNA1***	***SNA2***	***SNA3***	***SHO1***	***SHO2***	***SHO3***	***Overall Error***
Overall Error GS	GPT-3.5 GPT-4o Qwen2.5 Qwen2.5 Coder Llama 3.3	GPT-4o Qwen2.5	GPT-4o Qwen2.5 Coder Llama 3.3	GPT-4o Qwen2.5 Qwen2.5 Coder Code Llama DeepSeek	GPT-4o	GPT-4o Qwen2.5 Coder DeepSeek	GPT-4o Qwen2.5 Qwen2.5 Coder Llama 3.3	GPT-4o Qwen2.5 Llama 3.3	GPT-4o Llama 3.3 DeepSeek	GPT-4o
Overall Error NL	GPT-3.5 GPT-4o Qwen2.5 Llama 3.3 Code Llama DeepSeek	GPT-4o	GPT-4o	All Equal	GPT-4o	Llama 3.3	DeepSeek	GPT-4o	GPT-4o Llama 3.3	GPT-4o
Overall Error RJ	GPT-3.5 GPT-4o Qwen2.5 Qwen2.5 Coder Llama 3.3	Qwen2.5	Llama 3.3	All Equal	GPT-4o	GPT-3.5	GPT-4o Qwen2.5 Qwen2.5 Coder Llama 3.3	Qwen2.5 Coder	GPT-4o	GPT-4o
Overall Error	GPT-3.5 GPT-4o Qwen2.5 Llama 3.3	GPT-4o Qwen2.5	Llama 3.3	GPT-4o Qwen2.5 Qwen2.5 Coder Code Llama DeepSeek	GPT-4o	GPT-4o	Qwen2.5 Coder Llama 3.3	GPT-4o Qwen2.5 Coder	GPT-4o	GPT-4o

sure the `train` job specifies a directory for artefact creation and specifying that the `eval` and `model_test` jobs require the `train` job, were generally unproblematic.

6.3. Evaluation Scenarios

For the individual projects and scenarios, GPT-4o exhibited the lowest or equal-lowest error rate for most. Even in more complex cases, GPT-4o kept error rates low. These results indicate improved performance and reliability in generating MLOps pipelines compared to the other LLMs.

6.4. Common Errors Committed by the LLMs

We identified and categorised the most common errors and mistakes made by the LLMs according to their corresponding task category. It should be noted that, due to the unpredictable nature of LLMs, these errors cannot be foreseen when first implementing a solution such as ours. The errors only become apparent when running the tool and analysing what went wrong. These errors were straightforward to resolve in a single iteration – see Section 7 – by introducing simple improvements to our low-code specification and conversation templates based on our findings below, leading to even lower error rates.

6.4.1. Generation Errors

GPI: The generated script uses a correct Python image. Occasionally, either no Python image or an incorrect image was configured. Sometimes, a Python image was specified, but only for specific stages. These mistakes resulted in runtime errors during the training or evaluation stages because the `python` command was not found, or it was not possible to run the `pip` command due to the Docker environment being externally managed. These errors affected Qwen2.5 Coder when running scenario 2 on the *SHO* project (*SHO2*) and Llama 3.3 for *KAZ2* and *SNA2*. Please note that we did not always specify the use of a Python image, let alone a specific version, so a simple fix would be to always select a particular version, which should have a global scope.

GAR: All Python requirements and the necessary Linux packages are correctly installed for all required stages. On one occasion (Code Llama and *SHO2*), the LLM used the `apt` command instead of `pip` to install the Python prerequisites. It installed the incorrect version of `protobuf`, leading to runtime errors during pipeline execution. The solution is instructing the LLM to always install Linux requirements with `apt` and Python requirements with `pip`.

GEN: All specified environment variables are correctly set. LLMs sometimes set environment variables using the `export` command rather than a global `variables:` section of the pipeline configuration, or they would set environments only within the scope of a specific stage. This error arose with Code Llama (*KAZ3* and *SHO2*), Qwen2.5 (*SNA3*) and DeepSeek (*KAZ3*). It resulted in pipeline stages (e.g. `deploy`) failing due to these environment variables being undefined for those stages. The solution is to specify that all environment variables should be set globally in the `variables:` section of the pipeline configuration.

GDA: All Docker artefacts are correctly created. Sometimes, the generated pipeline configuration didn't specify the model artefact for the `train` stage. The solution is to state that the model should be provided in that stage in the `artifacts:` section of the stage. Errors were mainly found with GPT-3.5 (*SNA2*), Code Llama (*KAZ2*, *SNA2* and *SHO2*) and DeepSeek (*SHO2*).

6.4.2. Natural Language Errors

NET and NMT: The `eval` and `model_test` jobs need the `train` job and its artefacts. In several cases, either the `eval` (*NET*) or `model_test` (*NMT*) stages didn't specify needing the `train` stage. For *NET*, the issue arose with every model for *SHO1*, whereas for *NMT*, mainly GPT-3.5 was affected (*KAZ3*, *SNA3* and *SHO3*). This omission wasn't a problem in practice since artefacts from prior stages were passed by default to the next stage in GitLab.

NSK: An SSH key setup script is correctly created to reach the deployment server. This problem occurred relatively frequently and resulted in runtime errors during the `deploy` stage when using secure copy. Worst affected were GPT-3.5 (*KAZ3*), Qwen2.5 Coder (*SHO3*) and DeepSeek (*KAZ3* and *SHO3*). The LLMs tried various ways to create the SSH key, including storing it in an authentication agent instead of a file, not creating the `.ssh` directory, trying to write the private key to the `/root` directory, not setting correct permissions on the `~/.ssh/id_rsa` file, not adding the host to `known_keys` or completely omitting SSH setup. The solution would be to provide the explicit SSH script (optionally as a self-contained file) for the pipeline stage to execute, since the steps would rarely change.

NSC: Secure copy is used to perform the deployment with the correct user/key, host and directory. Another frequent issue was deploying the model using the `scp` (secure copy) command. Sometimes, LLMs would use `rsync` instead or use an incorrect target path or hostname. The LLMs would sometimes not ensure the target path existed on the server and create it if necessary. Finally, the secure copy would also fail if the SSH setup were absent or had failed. As for *NSK*, the solution would be to provide the steps as an executable script or be more specific about the values and commands to use. Most impacted were GPT-3.5 with a 100% failure rate for each of *KAZ3*, *SNA3* and *SHO3*, Qwen2.5 (*SNA3*), Code Llama (*SNA3*) and DeepSeek (*SHO3*).

NDA: The `deploy` stage provides Docker results as an artefact. Infrequently, the Dockerfile would be configured as the artefact for the `deploy` stage instead of the generated Docker image. Sometimes, the generated pipeline configuration didn't specify any Docker image artefact. The solution to both problems is to state that the generated Docker image should be provided in that stage in the `artifacts:` section of the stage. Most LLMs were affected, in particular GPT-3.5 (*SNA2*), Qwen2.5 Coder (*KAZ2* and *SNA2*), Llama 3.3(*SNA2*), Code Llama (*KAZ2*, *SNA2* and *SHO2*) and DeepSeek (*KAZ2*, *SNA2* and *SHO2*).

NDI: Docker in Docker (DIND) is realised, and the correct image is selected. A frequently occurring mistake was using the incorrect DIND image or not specifying DIND at all. The result during pipeline runtime was that specific commands were not found (e.g. `pip` and `docker`). Again, most LLMs were impacted: Qwen2.5 (*KAZ2*, *SNA2* and *SHO2*), Qwen2.5 Coder (*SNA2*), Llama 3.3(with a 100% error rate for *KAZ2*, *SNA2* and *SHO2*), Code Llama (*SNA2*) and DeepSeek (*KAZ2* and *SNA2*). On one occasion, a specified version and entry point prevented the Docker daemon from starting (GPT-3.5 running *KAZ2*). The solution is to be explicit about which image version to use and specify that `docker:dind` should be used under `service:` in the `deploy` stage.

6.4.3. Runtime Errors

RUT: Unit tests ran successfully. Sometimes, the LLM erroneously assumed that unit tests are executed via `pytest` (GPT-4o for *SHO3*, Qwen2.5 for *KAZ3* and *SHO3*, Code Llama for *KAZ3* and DeepSeek for *SHO3*). On another occasion, the LLM would get unit tests, model tests and model evaluation mixed up and attempt to run scripts meant for different stages in this stage (GPT-3.5 for *SHO3*). These issues can be avoided by providing a clear example of how to run the tests to the LLM.

RTR: Model training ran successfully. Other than problems resulting from no Python image being specified during generation, there were no problems with the model training stage. The solution is to specify the precise Python image to use, as for *GPI*.

RTE: Model tests ran successfully. Like *RTR*, errors were mainly caused by issues with the Python image (see *GPI*). More precise instructions to the LLM would help in these cases.

REV: Model evaluation ran successfully. The most frequently encountered issue with this task was environment variables not being set correctly, resulting in the model from the `train` stage not being found. The solution for the generation task is the same as the one described for *GEN*, i.e. to set the environment variables globally.

RDP: Model deployment ran successfully. Other than the reasons for failure described earlier, sometimes it appeared that the LLM didn't understand what "deployment" means in the pipeline context. As a result, the pipeline would be configured to push the Docker image to *registry.gitlab.com* (Qwen2.5 Coder and Code Llama for *KAZ2*, and DeepSeek for *SNA2*) or would assume that the model was destined for Kubernetes[28] (DeepSeek for *SHO2*), and would use the `kaniko-builder` command. Another example of the LLM misinterpreting the specification led to unexpected commands, such as changing the directory (`cd ..`), e.g. with Qwen2.5 running *SHO2*.

7. Improvements and Reevaluation

Based on our findings described in Section 6.4, where we examine common errors committed by the LLMs, we made several improvements to the low-code, natural language template and the text used when prompting as part of our conversation template, with a view to running another evaluation after implementing the changes. We describe the changes in this section. Later in this section, we report on the improved results from the reevaluation.

7.1. Improvements to the Low-Code, Natural-Language Template

Our overall approach remains as in Section 4.3, with some enhancements. These changes represent a maturation of the CI/CD specification template, transitioning from a basic functional setup to a more professional, maintainable and standardised approach that more closely follows GitLab CI best practices whilst ensuring consistent execution environments,

[28] `https://kubernetes.io`

artefact management, resource allocation, more precise documentation and more reliable deployment, addressing our prior findings. Table 4 summarises the changes we made to the low-code template, which are discussed below. The modifications specify enhanced script usage instructions to the LLM by providing more precise documentation for each command. The improved specification template replaces ambiguous single-script invocations, such as `python snake_rl.py ?train|eval|unit_test|model_test?` with four explicitly labelled commands for `train`, `eval`, `unit_test` and `model_test`, each invoking a dedicated script. This change makes the usage of each command immediately apparent. The file paths are also made more explicit by adding `./` prefixes, which follow better scripting practices and remove any ambiguity about file locations.

Table 4: Comparison of Changes to the Low-Code Template

Aspect	*Original*	*Change*
Usage	Unlabelled commands.	Labelled commands with file paths.
Runtime Environment	Unspecified Python image.	Named Python image for all stages.
Dependency Management	Unclear for which stages dependencies should be installed and when.	Introduction of a `before_script` to specify dependencies separate from execution logic.
Dependency Management	Program for installing dependencies unspecified.	Specify `pip` for dependency installation.
GitLab Runners	No GitLab runners specified.	Introduction of optional runner tags.
Terminology	Inconsistent.	Correct use of `job` and `stage`.
Terminology	"Git**l**ab", "**d**ocker".	"Git**L**ab", "**D**ocker".
Configuration	Scope of environment variables undefined.	Environment variables set globally (for all stages).
Docker-in-Docker (DIND)	Mention using DIND.	Precise Docker-in-Docker (DIND) specification for the `deploy` stage.
Deployment Stage Setup	Python dependencies installed.	Override `before_script` to skip Python dependencies installation.
Deployment Strategy	Deployment to a Docker container provided as an artefact.	Save a Docker image and provide it as an artefact.
Deployment Strategy	Description of the deployment steps.	Deploy steps are outsourced to an external script.
Artefact Management	No explicit instructions.	The model artefact for the "train" stage should be provided in the "artifacts:" section of the "train" stage.

An essential addition in the improved version is the Python Docker image specification, for instance `python:3.8-bullseye`, for all pipeline stages. This addition ensures consistent runtime environments across the entire CI/CD pipeline, eliminating potential issues caused by environment variations. The original version left the runtime environment unspecified, which led to inconsistent behaviour across different pipeline stage runs.

The enhanced solution introduces structured dependency management by requiring Python packages to be installed in a dedicated `before_script` section using `pip`. This approach separates dependency installation from the main execution logic, making the pipeline more maintainable and following GitLab CI best practices. The original version didn't specify where or how these requirements should be installed.

The improved implementation requires all stages to use the `gpu-runner` tag, ensuring consistent resource allocation across the pipeline. Whilst not specifically addressing output correctness, this change can expedite pipeline execution and move the implementation closer to production readiness.

We also refined the use of terminology. References to `job` and `stage` now align with proper GitLab CI terminology. Capitalisation of `GitLab` and `Docker` has also been corrected.

The enhanced version emphasises centralised configuration practices by mandating that all environment variables be defined globally in the pipeline's `variables:` section. This practice promotes consistency and easier maintenance compared to stage-specific variable definitions.

The new implementation provides a detailed Docker-in-Docker (DIND) configuration for the deploy stage. Whilst the original version mentioned using DIND generically, the enhanced version specifies exact requirements: using `docker:latest` as the base image, and setting `docker:dind` as a service. We also specify an empty `before_script:` to avoid conflicts with Docker operations. This level of specificity prevents common DIND configuration issues.

Deployment has been refined from container deployment to image artefact creation. The original template described `deployment to a Docker container that is provided as an artifact`, which was somewhat ambiguous. The enhanced solution clarifies that the deploy stage should `save a Docker image and provide it as an artifact`, making the deployment output more explicit and reusable. The deployment approach has also undergone a fundamental transformation from the LLM having to guess at the deployment commands to use based on ambiguous instructions to script-based execution. Typically, the deployment script would be provided by the operations infrastructure team. The original implementation specified SSH deployment tasks, including server, user credentials, SSH key management, and file transfer operations, but not the commands. The improved solution outsources these operations into an external `deploy.sh` script that must be made executable and executed, simplifying the pipeline configuration whilst improving modularity and maintainability.

The enhanced tool makes artefact configuration more explicit and comprehensive. Both configurations produce model artefacts in `models/`, but the enhanced template requires defining these outputs in the `artifacts:` section of the train stage. This declaration ensures that downstream processes or future jobs can reliably locate and use the trained model files.

7.2. Improvements to the Prompting Method

We made several changes to the prompting method. The overall structure of the conversation template and the prompting method remain the same as described in Section 4.6, but we streamlined the content to achieve more accurate output. The key differences between the original and new implementations are listed in Table 5 and described below.

Table 5: Comparison of Changes to the Conversation Text

Aspect	***Original***	***Change***
Question Wording	Write a Gitlab CI/CD yml file for **Python machine-learning scripts**.	Write a GitLab CI/CD YAML file for **Python reinforcement learning scripts**.
Sample Configuration and Output	No GitLab runner `tags` section in any jobs.	Jobs can include a `tags` section specifying a GitLab runner.
System Message Wording	"You are **a helpful code assistant that can teach a junior developer how to code**."	"You are an **expert programmer**."
Instruction Wording	"Don't explain the **code, just generate the code block itself**."	"Don't explain the **file; just generate it**."
Terminology & Capitalisation	"Git**l**ab", "**d**ocker", "**yml**".	"Git**L**ab", "**D**ocker", "**YAML**".

The new implementation represents a more mature, production-ready version with GPU support and more accurate terminology to guide the chosen LLM in generating more correct output. Overall, our changes to the file represent a maturation of the tool's capabilities: it is more targeted in its application, supports GPU-enabled execution, and speaks with a clearer, more confident tone – all whilst maintaining the essential stages, scripts, and artefact handling of the original conversation template.

The revised question prompt shifts the focus from broad "Python machine-learning scripts" to the narrower domain of "Python reinforcement learning scripts" signalling a more precise use case. This update clarifies the nature of the task and aligns it with RL projects rather than generic ML tasks. This text can, of course, be modified depending on the type of target project (ML vs RL). By honing in on RL, the template better reflects that subfield's specialised requirements and typical conventions.

A significant enhancement in the new version is the introduction of GPU-runner tags, if desired, across all job stages. In the updated sample specification, a `gpu-runner` tag is specified for every stage except any hypothetical deploy stage, and each job block in the generated YAML now includes a `tags` section listing `gpu-runner`. This change can, of course, be customised to the user's infrastructure. For instance, the `tags` specification may be omitted entirely, the runner may be named differently than `gpu-runner`, and the stages for which the tags are specified or omitted may be modified. This modification, whilst not targeting LLM correctness, ensures that jobs will be executed on runners equipped with GPU resources, which is critical for compute-intensive RL training and evaluation tasks in production.

We also refined the system and instruction messages to present a more authoritative and professional voice. What was previously phrased as a `helpful code assistant that can teach a junior developer how to code` in the system message is now an `expert programmer`. The instruction `Don't explain the code, just generate the code block itself` in the instruction message has been streamlined to `Don't explain the file; just generate it`. These adjustments shorten the phrasing and elevate the perceived expertise of the assistant, making it more likely to create concise, expert-level assistance.

Terminology in the system messages and specifications has been standardised for consistency and clarity. References to `Gitlab` have been updated to `GitLab` in keeping with the proper product name, and the capitalisation of `Docker` has been corrected. The description of the CI/CD configuration has also shifted from `yml file` to `YAML file` where appropriate, reflecting the more formal nomenclature used in professional documentation and to make the expected markup format clear. These changes aim to increase the correctness of the LLMs' generated output. The terms were also updated in the rest of the codebase where appropriate.

7.3. Reevaluation

We ran the refined implementation on the best-performing LLM from the first evaluation round, GPT-4o, and evaluated it as previously described in Section 5. The results are summarised in Table A.13. In short, we achieved perfect results for this LLM and the new implementation. GPT-4o scored an overall error of zero for every combination of evaluation variable, project and scenario. Based on this success, we did not see the need to continue evaluating the remaining LLMs.

8. Integration of our Approach in an Industry Setting

This section describes the approach taken to apply our solution in an industrial CPPS. This integration is currently being developed in collaboration with an industry partner in a research and development project for the next generation of intelligent CPPSs. The tool finds particular applicability in scenarios where pipeline configurations must be continuously created and edited at scale, i.e. for many factories. Figure 7 depicts a typical CPPS setup.

In Figure 7, our tool (**LCF** - Low Code Flow) enables RL engineers, software developers and MLOps practitioners (represented collectively as the RL Engineer) to generate GitLab CI/CD pipeline configurations based on our low-code template using an LLM. The **RL Engineer** is the human agent of the system architecture and is focused on designing, sustaining, and optimising industrial-level solutions based on RL.

The **Software Engineering Environment** is where the **RL Engineer** carries out their main work activities. This environment comprises key elements which facilitate successful development processes. The **IDE** (Integrated Development Environment) gives the **RL Engineer** a complete set of development tools tailored to RL development, such as debugging support of complex multi-agent systems and visualisation support to understand agent behaviour. It can simplify the development process and cut the complexity of creating the RL system. The **RL Engineer** may make use of LLMs, chatbots or integrated AI tools like Copilot[29] - **LLM/Chatbot/Copilot** in Figure 7. These tools assist the **RL Engineer** with smart help such as coding suggestions, auto-documenting, and problem-solving. More importantly for the context of this paper, the **RL Engineer** can also use our program - Low Code Flow (**LCF**), which utilises an underlying LLM.

The **Code Repository** has version control and collaboration infrastructure so that all RL algorithm implementations, configuration files and system integration code are correctly tracked and managed. This aspect matters since, during the development of RL, a lot of experimentation and iteration typically occur, and robust version control is crucial to ensuring reproducible outcomes. As the **RL Engineer** commits code changes to the repository, the CI/CD pipeline automatically triggers, illustrating automated integration between the development and operational deployment processes. The difference with our solution is that **LCF** can create and edit an **RLOps CI/CD Pipeline** and commit it via the tool to the **Code Repository**, which also triggers a CI/CD pipeline run.

The **RLOps CI/CD Pipeline** within the **RLOps Environment** that gets triggered as described is the automation engine that coordinates the building, testing, validation and deployment of RL models. It aims to ensure that new model versions are evaluated systematically before deployment and that system reliability is ensured with the possibility of ongoing improvement. The **Model Repository** is a centralised model versioning and storage system that tracks metadata regarding model performance, model training parameters, and deployment history. This aspect is essential as RL models need a large amount of training, and their performance is often highly dependent on the training circumstances, so it is essential that production systems correctly version and manage metadata.

[29]`https://copilot.microsoft.com`

The **CPPS Factory Environment** in Figure 7 is the physical manufacturing environment where RL algorithms are implemented on real-life production processes. An **External Trigger**, such as changes in real-world model factory performance according to predefined performance metrics, can trigger a new deployment within the **CPPS Factory Environment**. The **Model Selector** may be used to deliver models to production factories. It offers intelligent model selection features, which allow the dynamic selection of suitable models from the **Model Repository** according to the existing operational conditions, production needs or performance indicators. It enables adaptive manufacturing systems to adjust their intelligence automatically in response to changing conditions without requiring human intervention. The **Line Controller** is the main orchestration element of manufacturing processes that organises the work of several production units and coordinates the operations, deploying models to any number of Cyber-Physical Production Units (**CPPU**s), each of which may have a different RL strategy at its core and be associated with robots with varying capabilities and competencies. **CPPU**s are extendable manufacturing units, which are physical equipment with cyber functionality. These units are necessary because they are the real locations where AIs make decisions that affect physical production.

The **Digital Twin** is a virtual representation of physical manufacturing assets in real time, which can be simulated, predicted, and optimised without affecting real production. It is important because it enables RL agents to experiment with strategies and estimate their outcomes with great certainty before changing the physical world.

Equipment Control directly communicates with actual manufacturing machinery and converts high-level commands issued by the RL agents into explicit equipment instructions. This element is necessary to guarantee safe and effective physical realisation of AI-generated strategies.

The **RL/MARL Agent** (Multi-Agent Reinforcement Learning Agent) is the heart of AI that can learn and optimise manufacturing processes using either an RL or MARL framework. This agent is the final frontier of the whole system architecture, in which the models are put into action to make real-time decisions that directly affect production efficiency, quality and adaptability.

9. Discussion

This section discusses our results in the context of the research questions defined in Section 1.

RQ1 *How accurately can a low-code, template-based modular flow approach — leveraging LLMs for generation and validation — produce correct MLOps pipeline configurations for RL?* Our empirical validation confirms that our low-code, template-based approach can generate MLOps pipelines with low error rates, contingent on the specific LLM selected for the task, which is discussed in further detail below under RQ2. Initial overall error rates across all evaluation variables, projects and scenarios ranged from 0.09 (GPT-4o) to 0.3 (Code Llama).

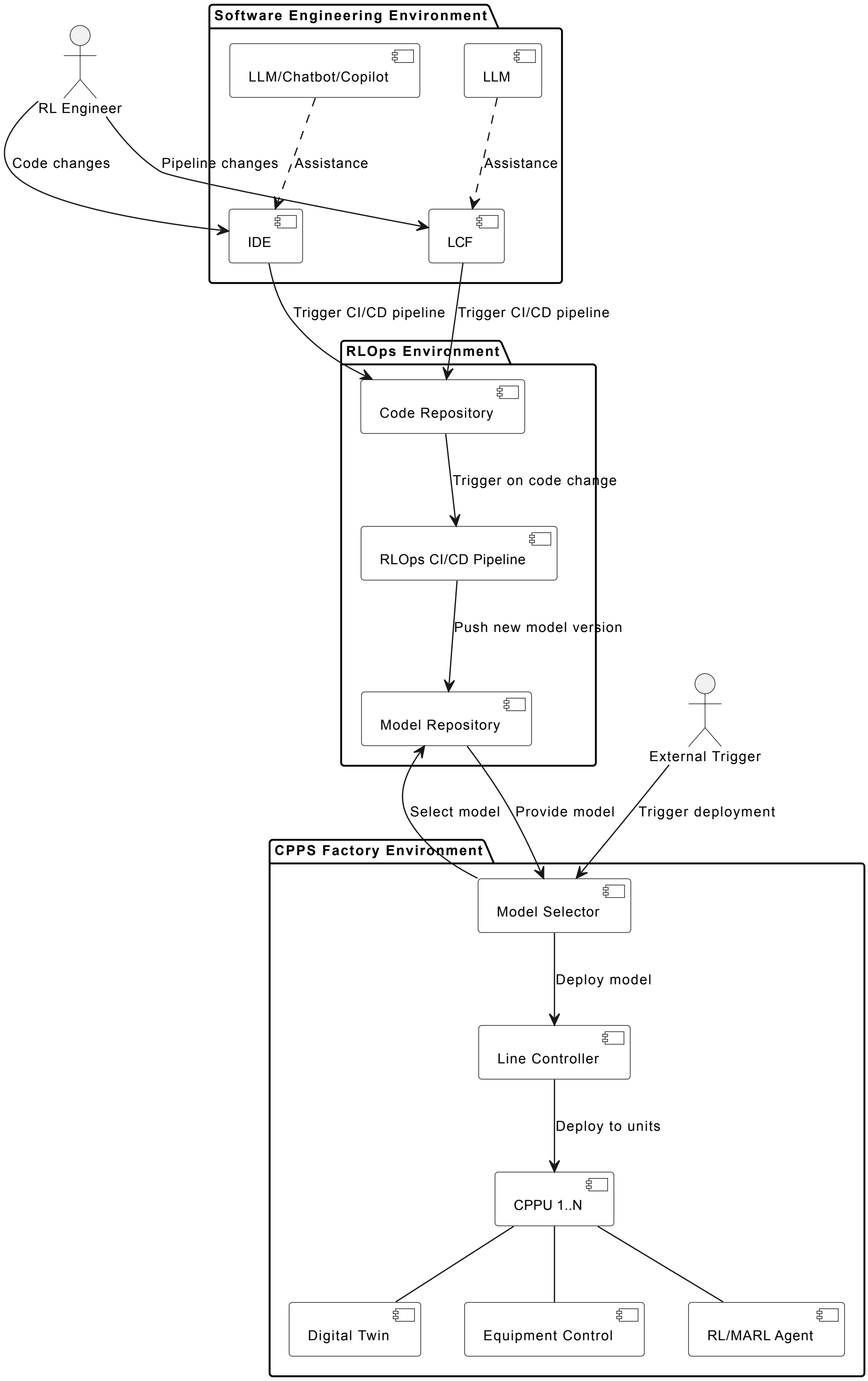

Fig. 7: Integration of our Approach in an Industry Setting.

Regarding task categories, our approach achieved a best error rate of 0 for generation tasks, 0.09 for natural language tasks, and 0.26 for runtime tasks, all achieved by GPT-4o. We also showed that error rates can be improved by making only minor corrections to the generated pipeline configuration, subsequently achieving perfect results for GPT-4o. Typically, these minor amendments are carried out by the individuals closest to the pipeline's domain: MLOps specialists, domain experts, ML engineers or in leaner teams, experienced data scientists who wear multiple hats. Since our approach flags errors via the LLMs' validation feedback or through straightforward GitLab failure messages, remedying issues often amounts to little more than updating some text in the low-code template and pipeline specification instances, or being more precise in the conversation template. Ample examples of correcting the errors noted in Section 6.4 are documented in Section 7.

Such tweaks as those we implemented impose a minimal burden on non-expert users, as the requisite details (for instance, using approved container images) are usually supplied alongside the pipeline templates by the underlying infrastructure team. Errors of this nature typically arise because the user neglected to specify the correct image in the first place. If, however, the LLM itself generates a pipeline configuration that doesn't succeed at runtime, for instance because it erroneously used an incorrect command to install Python dependencies (like `apt` instead of `pip`), then the user can add natural language text to the pipeline specification file that `Python requirements should be installed using pip` and rerun the tool. This continuous refinement can also aid in reducing future error rates.

Combined with selected LLMs' abilities to handle natural language inputs, our approach is potentially a good option for users who may not have deep DevOps knowledge, helping to make pipeline development more accessible. Our templates support a wide range of pipeline features. With their rigid structure, they are designed to guide practitioners, making it easy for users to specify their needs in natural language whilst also being detailed enough for LLMs to produce correct output, reducing the likelihood of a mismatch between expected and generated output.

We see a practical application of our approach: it can potentially reduce development time as users work with templates developed for pipeline descriptions, decreasing time spent learning how to write and debug GitLab pipeline configurations directly. Our solution could increase accessibility for non-experts as it allows users to participate in pipeline production without needing extensive software engineering or DevOps knowledge. The templates may enable rapid development, particularly in complex or dynamic settings with many pipelines or continuous pipeline creation or adaptation. Indeed, we plan to test the potential for increased production speed and applicability in a user study in future work (see Section 11).

Our findings provide concrete contributions. We discovered some common errors that different LLMs make when generating pipelines, which can guide practitioners on improving communication with LLMs when implementing an approach like ours. A key takeaway is that the limitations of the poorly performing LLMs appear due to ambiguity or lack of detail in parts of the pipeline specifications. Being more precise in the template and providing more GitLab-specific knowledge to the LLM can improve results further – we achieved perfect results for GPT-4o after doing so. A further contribution of this study is our comprehensive set of evaluation variables, and their resulting error rate metric can be used to assess the correctness of any MLOps or MLOps pipeline or automated approach with minimal modification, which could be used in similar studies or applications.

RQ2 *How accurately do different LLMs generate MLOps pipeline configurations for RL, and what are their limitations?* Comparative analysis of our initial results demonstrates that OpenAI's GPT-4o model performed best across all task types, with an overall error rate of just 0.09, and was the overall best-performing LLM. This result represents high consistency, reliability and accuracy in generating correct outputs. GPT-4o outperformed the other LLMs across most evaluation variables, with lower or equal overall error rates in most cases.

Qwen2.5 Coder also performed well, achieving an overall error rate of 0.15, with a generation task error rate similar to that of GPT-4o (0.04 and 0 respectively). Qwen2.5 Coder performed much worse than GPT-4o for natural language (0.17 vs 0.09 respectively) and runtime tasks (0.36 vs 0.26 respectively). Many LLMs had zero errors for multiple evaluation variables across all projects and scenarios. In particular, most LLMs achieved an error rate of zero for the generation tasks corresponding to *GAS* ("the generated script contains all required stages"), *GSO* ("the required stages are in the correct order") and *GAR* ("all Python requirements and the necessary Linux packages are correctly installed for all required stages"). GPT-4o consistently achieved zero-error results for a broader range of evaluation variables, suggesting users can expect more reliable output from a pipeline built with GPT-4o using our approach. We noted a low variability in error rates for GPT-4o, which was much more consistent across different projects, scenarios, and tasks, and performed remarkably well. All models struggled with the deployment task corresponding to *RDP* ("model deployment ran successfully"), which proved challenging at runtime, and a lowest error rate of only 0.72 was achieved (by Qwen2.5 Coder).

That GPT-4o demonstrated an overall error rate of only 0.09, beating more specialised coding models like Qwen2.5 Coder, Code Llama and DeepSeek R1 (with overall errors of 0.15, 0.3 and 0.22 respectively), is both surprising and indicative of the advancement in model design and training techniques. Despite their specialisation in code generation, these LLMs could not compare to GPT-4o in terms of consistency and accuracy. Even broader models such as GPT-3.5 Turbo, Qwen2.5 and Llama 3.3 (overall error rates being 0.19, 0.18 and 0.18 respectively) could not match GPT-4o, indicating that the architectural improvements and multimodality of GPT-4o provide a practical benefit to tackle various tasks without sacrificing depth and accuracy. The fact that GPT-4o performed consistently well on different projects and scenarios indicates its stability, and it is an ideal option in pipelines where consistency of output is paramount. This consistency is compounded by GPT-4o's perfect performance in the enhanced version of our tool after making minimal changes.

Our results raise the question of what specialised LLMs can reasonably be expected to accomplish compared to more general LLMs. It could be expected that a model designed specifically to code – like Code Llama or DeepSeek R1 – would perform well on generation or runtime benchmarks that involved programming tasks. Still, these models performed worse overall on deployment and runtime tasks. The fact that these coding-oriented LLMs underperform shows that specialisation does not necessarily imply lower error rates in all task dimensions. Instead, the balanced and subsequent perfect performance of GPT-4o on natural language, generation and execution tasks implies that multimodal training and inference optimisations can be more reliably integrated to produce an overall performance that is greater than the sum of its parts. To organisations considering model selection, these results point to the importance of flexibility and robustness over domain specialisation, and that even the most accurate coding LLMs may need supplementary validation mechanisms in production, such as the ones we implemented, to counter edge-case failures.

Our comparison shows that LLMs vary in performance, and standard LLMs can produce impressive results. A key observation is that code-specific LLMs should not automatically be assumed to perform better than non-code-specific LLMs for code-generation applications such as ours. Combining the strengths of multiple LLMs, fine-tuning, and more advanced prompt engineering and error correction techniques, focusing on the most challenging tasks, may yield even better results. Alternatively, by being more precise in our pipeline specification and conversation template, results were improved upon even further.

10. Threats to Validity

In this section, we address threats to validity according to Wohlin et al. [56].

Internal Validity. The correctness of the produced pipelines depends on how comprehensive and precise the templates are. LLMs' capabilities to accurately understand and create code from template-based and natural language instructions can vary. Incorrectly interpreting templates, particularly natural language inputs, can lead to mistakes or less-than-ideal configurations, which in turn may cause problems in the generated pipelines. In cases where we implemented specific checks for the contexts of the generated configurations, this issue was often considerably reduced.

The way prompts are designed also heavily influences the correctness of LLM-generated pipelines. If prompt phrasing is unintentionally biased toward certain LLMs, some models may perform better due to better alignment with the prompt style rather than their actual capabilities. We sought to mitigate this risk by using carefully formulated prompts, applying few-shot prompting and prompt chaining, and error detection and retries to guide the LLMs. We manually inspected LLM output over numerous scenarios, evaluation variables, projects and LLMs. We did not detect any incompatibility or bias that was obviously linked to the prompt style. We refined the prompts during the initial development phase (before beginning the evaluation) until we started receiving reasonable outputs from the LLMs.

External Validity. We assess our approach based on a small number of RL cases, which may not apply to other fields or project types. Due to the scarcity of open-source MLOps projects utilising RL, we had to independently develop the ground-truth pipelines for our case study projects. Although we adhered to practices observed in the industry (based on the literature [8, 17, 23] and industry projects), we may have overlooked some essential MLOps practices. Various systems might need considerable modifications to the specification and generation classes. Whilst these factors could restrict the immediate use of our methodology, the fundamental approach should still be relevant in similar situations. The effectiveness of our approach is contingent on the capabilities of the underlying LLMs. Differences in the quality of the model or the inputs given to it can affect the standard of the generated pipelines. We tackled this concern by conducting tests on various cases, evaluation scenarios and LLMs.

Construct Validity. The metrics used to assess the generated pipelines might not comprehensively reflect all aspects of their performance, and all evaluation metrics carry equal weight in our assessment. In contrast, some factors may hold greater significance in practice than others. In this case, one could re-evaluate our findings using a set of weighted metrics.

The effectiveness of the natural language-based specification components relies on the accuracy of the user's inputs. Therefore, slight variations in human inputs can result in inconsistencies in the generated outputs, which could affect the validity of the results. Likewise, differently structured prompts could yield different generated outcomes. We ran our evaluation multiple times for multiple projects and LLMs to account for this eventuality.

Conclusion Validity. The results of our experiments rely on a small number of examples and cases. A broader and more varied range of case study projects might strengthen the validation of our approach. Any biases present in the training data of the LLMs we used could have affected the generated pipelines. We addressed this concern by conducting tests with multiple LLMs.

11. Conclusion and Future Work

In this paper, we present a novel approach for automating the generation of MLOps pipelines for RL applications, employing a low-code, template-based, modular approach that leverages LLMs to enable users with minimal coding expertise to generate MLOps pipelines for testing, model training and validation, and model deployment. Our evaluation indicates low error rates for pipeline generation, natural language comprehension and runtime tasks, with GPT-4o performing best out of seven LLMs and even achieving a perfect overall error rate of zero after simple refinement of our tool. The results indicate how LLMs, with the correct approach incorporating a validation and correction strategy, can foster trustworthiness in software by minimising errors in the underlying infrastructure and issues with LLM-based code generation.

To further validate our solution, assess its usefulness, and inform future enhancements, we plan to conduct a quantitative user study comparing the use of our approach with the manual creation of pipeline configurations as part of an Industry 4.0 case study in collaboration with domain experts. We suggest that using our proposed system could drastically reduce coding requirements, accelerate pipeline development, and bring MLOps closer to data scientists and other domain experts who may lack advanced software engineering or DevOps skills, thereby democratising MLOps and simplifying pipeline creation for people with limited technical know-how.

Other potential directions include fine-tuning LLMs and developing more advanced error detection and correction mechanisms in the pipeline generation process to support a broader range of LLM types. These improvements could target weaker tasks (such as model deployment). Another enhancement could be incorporating a feedback mechanism that learns from deployed pipelines to improve future pipeline generation.

Finally, this study focused on the applicability of our approach and the correctness of our pipeline. A future study could examine LLM pipeline generation efficiency, including aspects such as the time taken to generate a pipeline and the number of tokens consumed, since these are also factors in practice.

CRediT Authorship Contribution Statement

As suggested in the JSS author policies and guidelines, the CRediT Authorship Contribution Statement will be included in the final version of the paper.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data Availability

All relevant source code and artefacts are available in our Zenodo replication package [15].

Acknowledgements

This research was funded in whole or in part by the FFG (Austrian Research Promotion Agency) project MODIS (no. FO999895431) and the Austrian Science Fund (FWF) project CQ4CD, Grant-DOI: 10.55776/I6510. For open access purposes, the authors have applied a CC BY public copyright license to any author-accepted manuscript version arising from this submission.

References

[1] M. Treveil, N. Omont, C. Stenac, K. Lefevre, D. Phan, J. Zentici, A. Lavoillotte, M. Miyazaki, and L. Heidmann, *Introducing MLOps*. O'Reilly Media, 2020.

[2] Y. Zhou, Y. Yu, and B. Ding, "Towards MLOps: A case study of ML pipeline platform," in *2020 International conference on artificial intelligence and computer engineering (ICAICE)*. IEEE, 2020, pp. 494–500.

[3] M. M. John, H. H. Olsson, and J. Bosch, "Towards MLOps: A framework and maturity model," in *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2021, pp. 1–8.

[4] E. Bisong *et al.*, *Building machine learning and deep learning models on Google cloud platform*. Springer, 2019.

[5] M. Steidl, M. Felderer, and R. Ramler, "The pipeline for the continuous development of artificial intelligence models—Current state of research and practice," *Journal of Systems and Software*, vol. 199, p. 111615, 2023.

[6] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich, "Data management challenges in production machine learning," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 1723–1726.

[7] P. Li, J. Thomas, X. Wang, A. Khalil, A. Ahmad, R. Inacio, S. Kapoor, A. Parekh, A. Doufexi, A. Shojaeifard *et al.*, "RLOps: Development life-cycle of reinforcement learning aided open RAN," *IEEE Access*, vol. 10, pp. 113 808–113 826, 2022.

[8] E. Ntentos, S. J. Warnett, and U. Zdun, "Supporting Architectural Decision Making on Training Strategies in Reinforcement Learning Architectures," in *2024 IEEE 21st International Conference on Software Architecture (ICSA)*, 2024, pp. 90–100.

[9] S. J. Warnett and U. Zdun, "Bridging the Gap Between MLOps and RLOps: An Industry 4.0 Case Study on Architectural Design Decisions in Practice," in *2025 IEEE 22nd International Conference on Software Architecture (ICSA)*, 2025, pp. 232–242.

[10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture*. John Wiley & Sons Ltd, 1996.

[11] DAIR.AI, "Prompt Engineering Guide," https://www.promptingguide.ai/techniques, 2024.

[12] L. Monostori, B. Kádár, T. Bauernhansl, S. Kondoh, S. Kumara, G. Reinhart, O. Sauer, G. Schuh, W. Sihn, and K. Ueda, "Cyber-physical systems in manufacturing," *CIRP Annals*, vol. 65, no. 2, pp. 621–641, 2016.

[13] S. J. Oks, M. Jalowski, M. Lechner, S. Mirschberger, M. Merklein, B. Vogel-Heuser, and K. M. Möslein, "Cyber-Physical Systems in the Context of Industry 4.0: A Review, Categorization and Outlook," *Information Systems Frontiers*, 2022.

[14] H. Singh and B. Singh, "Industry 4.0 technologies integration with lean production tools: a review," *The TQM Journal*, 02 2024.

[15] S. J. Warnett and U. Zdun, "MLOps Pipeline Generation for Reinforcement Learning: A Low-Code Approach Using Large Language Models: Replication Package," Dec. 2025. [Online]. Available: https://doi.org/10.5281/zenodo.18054482

[16] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[17] S. J. Warnett and U. Zdun, "Architectural design decisions for the machine learning workflow," *Computer*, vol. 55, no. 3, pp. 40–51, 2022.

[18] D. Kreuzberger, N. Kühl, and S. Hirschl, "Machine Learning Operations (MLOps): Overview, Definition, and Architecture," *IEEE Access*, vol. 11, pp. 31 866–31 879, 2023.

[19] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*, 1st ed. Addison-Wesley Professional, 2015.

[20] T. Mboweni, T. Masombuka, and C. Dongmo, "A Systematic Review of Machine Learning DevOps," in *2022 International Conference on Electrical, Computer and Energy Technologies (ICECET)*, 2022, pp. 1–6.

[21] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, 1st ed. Addison-Wesley Professional, 2010.

[22] M. Shahin, M. Zahedi, M. A. Babar, and L. Zhu, "An empirical study of architecting for continuous delivery and deployment," *Empirical Softw. Engg.*, vol. 24, no. 3, p. 1061–1108, jun 2019. [Online]. Available: https://doi.org/10.1007/s10664-018-9651-4

[23] S. J. Warnett and U. Zdun, "Architectural design decisions for machine learning deployment," in *2022 IEEE 19th International Conference on Software Architecture (ICSA)*. IEEE, 2022, pp. 90–100.

[24] N. Vemuri, N. Thaneeru, and V. M. Tatikonda, "AI-Optimized DevOps for Streamlined Cloud CI/CD," *International Journal of Innovative Science and Research Technology*, vol. 9, no. 7, pp. 10–5281, 2024.
[25] B. Zhang, T. Liu, P. Liang, C. Wang, M. Shahin, and J. Yu, "Architecture decisions in AI-based systems development: an empirical study," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 616–626.
[26] I. Kumara, R. Arts, D. Di Nucci, W. J. Van Den Heuvel, and D. A. Tamburri, "Requirements and reference architecture for MLOps: insights from industry," *Authorea Preprints*, 2023.
[27] P. Sharma and M. S. Kulkarni, "A study on Unlocking the potential of different AI in Continuous Integration and Continuous Delivery (CI/CD)," in *2024 4th International Conference on Innovative Practices in Technology and Management (ICIPTM)*. IEEE, 2024, pp. 1–6.
[28] P. Liang, B. Song, X. Zhan, Z. Chen, and J. Yuan, "Automating the Training and Deployment of Models in MLOps by Integrating Systems with Machine Learning," *arXiv preprint arXiv:2405.09819*, 2024.
[29] S. A. A. Naqvi, M. P. Zimmer, P. Drews, K. Lemmer, and R. Basole, "The Low-Code Phenomenon: Mapping the Intellectual Structure of Research," in *Hawaii International Conference on System Sciences*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:266757600
[30] H. El Kamouchi, M. Kissi, and O. El Beggar, "Low-code/No-code Development: A systematic literature review," in *2023 14th International Conference on Intelligent Systems: Theories and Applications (SITA)*, 2023, pp. 1–8.
[31] M. Hirzel, "Low-Code Programming Models," *Commun. ACM*, vol. 66, no. 10, p. 76–85, Sep. 2023. [Online]. Available: https://doi.org/10.1145/3587691
[32] J. C. Kirchhof, N. Jansen, B. Rumpe, and A. Wortmann, "Navigating the Low-Code Landscape: A Comparison of Development Platforms," in *2023 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, 2023, pp. 854–862.
[33] O. Bruhin, E. Dickhaut, E. Elshan, and M. M. Li, "The Rise of Generative AI in Low Code Development Platforms - An Analysis and Future Directions," in *Hawaii International Conference on System Sciences*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:266757710
[34] H. Jin, H. Chen, Q. Lu, and L. Zhu, "Towards Advancing Code Generation with Large Language Models: A Research Roadmap," *arXiv preprint arXiv:2501.11354*, 2025.
[35] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A Survey on Large Language Models for Code Generation," *ACM Trans. Softw. Eng. Methodol.*, Jul. 2025, just Accepted. [Online]. Available: https://doi.org/10.1145/3747588
[36] D. Li and L. Murr, "HumanEval on latest GPT models–2024," *arXiv preprint arXiv:2402.14852*, 2024.
[37] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," 2021. [Online]. Available: https://arxiv.org/abs/2107.03374
[38] W. Li, D. Zan, B. Guan, A. Yu, X. Chen, and Y. Wang, "Improving Natural Language Capability of Code Large Language Model," *arXiv preprint arXiv:2401.14242*, 2024.
[39] Z. Zheng, K. Ning, Y. Wang, J. Zhang, D. Zheng, M. Ye, and J. Chen, "A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends," *ArXiv*, vol. abs/2311.10372, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:265281389
[40] C. Parnin, G. Soares, R. Pandita, S. Gulwani, J. Rich, and A. Z. Henley, "Building Your Own Product Copilot: Challenges, Opportunities, and Needs," *arXiv preprint arXiv:2312.14231*, 2023.
[41] K. Jesse, T. Ahmed, P. T. Devanbu, and E. Morgan, "Large Language Models and Simple, Stupid Bugs," in *Proc. 20th International Conference on Mining Software Repositories (MSR)*. IEEE/ACM, May 2023, pp. 563–575. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MSR59073.2023.00082
[42] Z. Xu, S. Jain, and M. Kankanhalli, "Hallucination is inevitable: An innate limitation of large language models," *arXiv preprint arXiv:2401.11817*, 2024.
[43] J. White, S. Hays, Q. Fu, J. Spencer-Smith, and D. C. Schmidt, *ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design*. Cham: Springer Nature Switzerland, 2024, pp. 71–108. [Online]. Available: https://doi.org/10.1007/978-3-031-55642-5_4
[44] D. Tan and J. Wang, "Engineering Practices for LLM Application Development," https://martinfowler.com/articles/engineering-practices-llm.html, 2024.
[45] N. M. do Nascimento, P. S. Alencar, and D. D. Cowan, "Artificial Intelligence vs. Software Engineers: An Empirical Study on Performance and Efficiency using ChatGPT," in *CASCON*, 2023, pp. 24–33.
[46] B. Meyer, "AI Does Not Help Programmers," *BLOG@CACM, https://cacm.acm.org/blogs/blog-cacm/273577-ai-does-not-help-programmers/fulltext*, 2023.
[47] H. Tian, W. Lu, T. O. Li, X. Tang, S.-C. Cheung, J. Klein, and T. F. Bissyandé, "Is ChatGPT the Ultimate Programming Assistant–How far is it?" *arXiv preprint arXiv:2304.11938*, 2023.
[48] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated Repair of Programs from Large Language Models," in *Proc. 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, p. 1469–1481. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00128
[49] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy, "InferFix: End-to-End Program Repair with LLMs," in *Proc. 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023, 2023, p. 1646–1656.
[50] L. Belzner, T. Gabor, and M. Wirsing, "Large language model assisted software engineering: prospects, challenges, and a case study," in *Proc. International Conference on Bridging the Gap between AI and Reality*. Springer, 2023, pp. 355–374.
[51] J. Yu, P. Liang, Y. Fu, A. Tahir, M. Shahin, C. Wang, and Y. Cai, "An Insight into Security Code Review with LLMs: Capabilities, Obstacles and Influential Factors," 2024. [Online]. Available: https://arxiv.org/abs/2401.16310
[52] A. Damarapati, "CI/CD Best Practices: Building Reliable Pipelines," *European Journal of Computer Science and Information Technology*, vol. 13, no. 10, pp. 1–10, 2025. [Online]. Available: https://doi.org/10.37745/ejcsit.2013/vol13n10110
[53] M. Fowler, *Patterns of enterprise application architecture*. Addison-Wesley, 2012.
[54] J. Liu, C. S. Xia, Y. Wang, and L. ZHANG, "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation," in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: https://openreview.net/forum?id=1qvx610Cu7
[55] DeepSeek-AI, D. Guo, D. Yang *et al.*, "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning," 2025. [Online]. Available: https://arxiv.org/abs/2501.12948
[56] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

Stephen John Warnett is a researcher with the Research Group Software Architecture at the Faculty of Computer Science, University of Vienna, Austria. His research interests include the intersection of software engineering, software architecture and artificial intelligence. Stephen received a Bachelor of Science degree with First Class Honours in Computer Science from the University of Edinburgh, Scotland, and a Master of Science degree with Distinction in Engineering from the University of Applied Sciences Technikum Wien, Austria.

Evangelos Ntentos is a PostDoc researcher with a Doctoral Degree in Software Architecture from the University of Vienna. He has been involved in various research projects focusing on DevOps and MLOps practices and architecture evaluation. His teaching experience spans several years as a teaching assistant in the Research Group Software Architecture at the University of Vienna. Evangelos has published in peer-reviewed conferences and journals on architectural design decisions, architecture evaluation in microservices, MLOps and Infrastructure as Code.

Uwe Zdun is a full professor of software architecture at the Faculty of Computer Science, University of Vienna. His research focuses on software design and architecture, distributed systems engineering (service-based, cloud, IoT, and microservices systems), SW engineering for ML, ML for SW engineering, DevOps and continuous delivery, and empirical software engineering. Uwe has published over 300 peer-reviewed articles and co-authored 4 professional books. He is Associate Editor of the Journal of Systems and Software (JSS) published by Elsevier, Associate Editor of the Computing journal published by Springer, and Associate Editor for design and architecture for the IEEE Software magazine.

Appendix A. Evaluation Results Tables

This appendix contains the tables with the evaluation results for each model, the scenarios and the evaluation variables. The rows of each table specify an evaluation variable or task type as described in Section 5, such as *GPI* ("The generated script uses a correct Python image"). The final rows correspond to the sets of evaluation variable types, for example, *Overall Error GS* represents all generation tasks. The final row, *Overall Error*, represents all evaluation variables. The columns, except the final column, contain the error rates for the open-source projects *KAZ*, *SNA*, and *SHO* described in Table 2, and the evaluation scenarios 1, 2 and 3 described in Section 5.2, as numerical suffixes. For instance, *SNA1*, *SNA2* and *SNA3* indicate the *SNA* project and each of the three evaluation scenarios. The last column, *Overall Error*, shows the overall error rate across all projects and scenarios for a given task type. As another example, in Table A.6, the cell represented by row *GEN* and column *KAZ2* shows an error rate of 0.33 for the evaluation variable *GEN* ("All specified environment variables are correctly set"). The final column of the same row indicated an overall error rate of 0.22 across all projects and scenarios. The cell for column *SHO1* and row *Overall Error GS* indicates an overall error of 0.06 for that project and scenario for generation tasks, and the final row in that column shows an overall error of 0.17 for project and scenario *SHO1* across all evaluation variables.

Table A.6: GPT-3.5 Turbo 0613 (Version 2023-06-13) Initial Results: Error Rates

	KAZ1	*KAZ2*	*KAZ3*	*SNA1*	*SNA2*	*SNA3*	*SHO1*	*SHO2*	*SHO3*	*Overall Error*
GPI	0.00	0.67	0.00	0.00	0.33	0.00	0.00	0.00	0.00	0.11
GAS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAR	0.00	0.00	0.00	0.00	0.33	0.00	0.00	0.67	0.00	0.11
GEN	0.00	0.33	0.33	0.33	0.00	0.33	0.33	0.00	0.33	0.22
GMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GCD	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
GDA	NA	0.33	NA	NA	0.67	NA	NA	0.33	NA	0.44
NTA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NET	0.00	0.00	0.00	NA	0.00	0.00	1.00	0.00	0.00	0.11
NMT	NA	NA	1.00	NA	NA	1.00	NA	NA	0.67	0.89
NSK	NA	NA	1.00	NA	NA	0.33	NA	NA	0.33	0.56
NSC	NA	NA	1.00	NA	NA	1.00	NA	NA	1.00	1.00
NDA	NA	0.33	NA	NA	0.67	NA	NA	0.33	NA	0.44
NDI	NA	0.67	NA	NA	0.33	NA	NA	0.33	NA	0.44
RUT	NA	NA	0.00	NA	NA	0.00	NA	NA	0.67	0.22
RTR	0.00	0.67	0.00	0.00	0.33	0.00	0.00	0.67	0.00	0.19
RTE	NA	NA	0.00	NA	NA	0.00	NA	NA	0.67	0.22
REV	0.00	0.67	0.00	NA	0.33	0.33	0.33	0.00	0.00	0.19
RDP	NA	1.00	1.00	NA	0.67	1.00	NA	0.33	1.00	0.83
Overall Error GS	0.00	0.17	0.06	0.06	0.17	0.06	0.06	0.13	0.06	0.08
Overall Error NL	0.00	0.25	0.60	0.00	0.25	0.47	0.50	0.17	0.40	0.29
Overall Error RJ	0.00	0.78	0.20	0.00	0.44	0.27	0.17	0.33	0.47	0.30
Overall Error	0.00	0.31	0.27	0.04	0.24	0.25	0.17	0.18	0.29	0.19

Table A.7: GPT-4o (Version 2024-08-06) Initial Results: Error Rates

	KAZ1	*KAZ2*	*KAZ3*	*SNA1*	*SNA2*	*SNA3*	*SHO1*	*SHO2*	*SHO3*	*Overall Error*
GPI	0.00	0.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04
GAS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GEN	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GCD	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
GDA	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
NTA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NET	0.00	0.00	0.00	NA	0.00	0.00	1.00	0.00	0.00	0.11
NMT	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NSK	NA	NA	0.00	NA	NA	0.33	NA	NA	0.00	0.11
NSC	NA	NA	0.00	NA	NA	0.33	NA	NA	0.00	0.11
NDA	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
NDI	NA	0.00	NA	NA	0.33	NA	NA	0.33	NA	0.22
RUT	NA	NA	0.33	NA	NA	0.00	NA	NA	0.33	0.22
RTR	0.00	0.33	0.33	0.00	0.00	0.00	0.00	0.00	0.33	0.11
RTE	NA	NA	0.33	NA	NA	1.00	NA	NA	0.33	0.56
REV	0.00	0.33	0.33	NA	0.00	0.33	0.00	0.00	0.33	0.15
RDP	NA	1.00	0.33	NA	1.00	1.00	NA	1.00	0.33	0.78
Overall Error GS	0.00	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Overall Error NL	0.00	0.00	0.00	0.00	0.08	0.13	0.50	0.08	0.00	0.09
Overall Error RJ	0.00	0.56	0.33	0.00	0.33	0.47	0.00	0.33	0.33	0.26
Overall Error	0.00	0.13	0.10	0.00	0.09	0.19	0.10	0.09	0.10	0.09

Table A.8: Qwen2.5 72B Initial Results: Error Rates

	KAZ1	*KAZ2*	*KAZ3*	*SNA1*	*SNA2*	*SNA3*	*SHO1*	*SHO2*	*SHO3*	*Overall Error*
GPI	0.00	0.00	0.00	0.00	0.33	0.00	0.00	0.00	0.00	0.04
GAS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GEN	0.00	0.33	0.67	0.00	0.00	0.33	0.00	0.00	1.00	0.26
GMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GCD	NA	0.00	NA	NA	0.33	NA	NA	0.00	NA	0.11
GDA	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
NTA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NET	0.00	0.00	0.00	NA	0.00	0.00	1.00	0.00	0.00	0.11
NMT	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NSK	NA	NA	0.00	NA	NA	0.33	NA	NA	0.00	0.11
NSC	NA	NA	0.33	NA	NA	0.67	NA	NA	0.33	0.44
NDA	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
NDI	NA	0.67	NA	NA	1.00	NA	NA	0.67	NA	0.78
RUT	NA	NA	1.00	NA	NA	0.33	NA	NA	1.00	0.78
RTR	0.00	0.00	1.00	0.00	0.33	0.33	0.00	0.00	1.00	0.30
RTE	NA	NA	1.00	NA	NA	0.33	NA	NA	1.00	0.78
REV	0.00	0.33	1.00	NA	0.33	0.67	0.00	0.00	1.00	0.37
RDP	NA	0.67	1.00	NA	1.00	1.00	NA	1.00	1.00	0.94
Overall Error GS	0.00	0.04	0.11	0.00	0.08	0.06	0.00	0.00	0.17	0.05
Overall Error NL	0.00	0.17	0.07	0.00	0.25	0.20	0.50	0.17	0.07	0.16
Overall Error RJ	0.00	0.33	1.00	0.00	0.56	0.53	0.00	0.33	1.00	0.42
Overall Error	0.00	0.13	0.38	0.00	0.22	0.25	0.10	0.11	0.40	0.18

Table A.9: Qwen2.5 Coder 32B Initial Results: Error Rates

	KAZ1	*KAZ2*	*KAZ3*	*SNA1*	*SNA2*	*SNA3*	*SHO1*	*SHO2*	*SHO3*	*Overall Error*
GPI	0.00	0.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04
GAS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GEN	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.11
GMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GCD	NA	0.00	NA	NA	0.67	NA	NA	0.00	NA	0.22
GDA	NA	0.33	NA	NA	0.00	NA	NA	0.33	NA	0.22
NTA	0.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04
NET	0.00	0.00	0.00	NA	0.00	0.00	0.67	0.00	0.00	0.07
NMT	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NSK	NA	NA	0.00	NA	NA	0.33	NA	NA	0.67	0.33
NSC	NA	NA	0.33	NA	NA	0.00	NA	NA	0.00	0.11
NDA	NA	0.67	NA	NA	0.67	NA	NA	0.33	NA	0.56
NDI	NA	0.33	NA	NA	0.67	NA	NA	0.33	NA	0.44
RUT	NA	NA	0.00	NA	NA	0.00	NA	NA	1.00	0.33
RTR	0.00	0.33	0.00	0.00	1.00	0.00	0.00	0.00	1.00	0.26
RTE	NA	NA	0.00	NA	NA	1.00	NA	NA	1.00	0.67
REV	0.00	0.33	0.00	NA	1.00	1.00	0.00	0.00	1.00	0.37
RDP	NA	0.67	0.33	NA	1.00	1.00	NA	0.33	1.00	0.72
Overall Error GS	0.00	0.08	0.00	0.00	0.08	0.00	0.00	0.04	0.17	0.04
Overall Error NL	0.17	0.25	0.07	0.00	0.33	0.07	0.33	0.17	0.13	0.17
Overall Error RJ	0.00	0.44	0.07	0.00	1.00	0.60	0.00	0.11	1.00	0.36
Overall Error	0.03	0.20	0.04	0.00	0.33	0.21	0.07	0.09	0.42	0.15

Table A.10: Llama 3.3 70B Initial Results: Error Rates

	KAZ1	*KAZ2*	*KAZ3*	*SNA1*	*SNA2*	*SNA3*	*SHO1*	*SHO2*	*SHO3*	*Overall Error*
GPI	0.00	1.00	0.00	0.00	1.00	1.00	0.00	0.00	0.00	0.33
GAS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GEN	0.00	0.00	0.00	0.33	0.33	0.33	0.00	0.00	0.00	0.11
GMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GCD	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
GDA	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
NTA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NET	0.00	0.00	0.00	NA	0.00	0.00	0.67	0.00	0.00	0.07
NMT	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NSK	NA	NA	0.33	NA	NA	0.00	NA	NA	0.00	0.11
NSC	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NDA	NA	0.00	NA	NA	0.67	NA	NA	0.00	NA	0.22
NDI	NA	1.00	NA	NA	1.00	NA	NA	1.00	NA	1.00
RUT	NA	NA	0.00	NA	NA	1.00	NA	NA	0.67	0.56
RTR	0.00	1.00	0.00	0.00	1.00	1.00	0.00	0.00	0.67	0.41
RTE	NA	NA	0.00	NA	NA	1.00	NA	NA	0.67	0.56
REV	0.00	1.00	0.00	NA	1.00	1.00	0.00	0.00	0.67	0.41
RDP	NA	1.00	0.00	NA	1.00	1.00	NA	1.00	0.67	0.78
Overall Error GS	0.00	0.13	0.00	0.06	0.17	0.22	0.00	0.00	0.00	0.06
Overall Error NL	0.00	0.25	0.07	0.00	0.42	0.00	0.33	0.25	0.00	0.15
Overall Error RJ	0.00	1.00	0.00	0.00	1.00	1.00	0.00	0.33	0.67	0.44
Overall Error	0.00	0.33	0.02	0.04	0.40	0.40	0.07	0.13	0.21	0.18

Table A.11: Code Llama 70B Initial Results: Error Rates

	KAZ1	*KAZ2*	*KAZ3*	*SNA1*	*SNA2*	*SNA3*	*SHO1*	*SHO2*	*SHO3*	*Overall Error*
GPI	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAS	0.00	0.00	0.00	0.00	0.00	0.00	0.67	0.00	0.00	0.07
GSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.33	0.04
GAR	0.00	0.00	0.00	0.00	0.00	0.67	0.00	0.33	0.00	0.11
GEN	1.00	0.67	0.67	0.00	0.33	0.67	0.00	0.67	0.67	0.52
GMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GCD	NA	0.67	NA	NA	0.33	NA	NA	0.00	NA	0.33
GDA	NA	1.00	NA	NA	1.00	NA	NA	1.00	NA	1.00
NTA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NET	0.00	0.00	0.00	NA	0.33	0.33	1.00	0.00	0.33	0.22
NMT	NA	NA	0.00	NA	NA	0.33	NA	NA	0.00	0.11
NSK	NA	NA	0.33	NA	NA	0.33	NA	NA	0.33	0.33
NSC	NA	NA	0.00	NA	NA	1.00	NA	NA	0.33	0.44
NDA	NA	1.00	NA	NA	1.00	NA	NA	1.00	NA	1.00
NDI	NA	0.33	NA	NA	0.67	NA	NA	0.33	NA	0.44
RUT	NA	NA	0.67	NA	NA	0.67	NA	NA	0.33	0.56
RTR	0.00	0.67	0.67	0.00	0.67	0.67	0.00	0.33	0.33	0.37
RTE	NA	NA	0.67	NA	NA	0.67	NA	NA	0.33	0.56
REV	1.00	1.00	0.67	NA	1.00	1.00	1.00	1.00	0.33	0.78
RDP	NA	1.00	0.67	NA	1.00	1.00	NA	1.00	1.00	0.94
Overall Error GS	0.17	0.29	0.11	0.00	0.21	0.22	0.11	0.25	0.17	0.17
Overall Error NL	0.00	0.33	0.07	0.00	0.50	0.40	0.50	0.33	0.20	0.26
Overall Error RJ	0.50	0.89	0.67	0.00	0.89	0.80	0.50	0.78	0.47	0.61
Overall Error	0.20	0.42	0.27	0.00	0.42	0.46	0.27	0.38	0.27	0.30

Table A.12: DeepSeek R1 70B (Version 2025-01-20) Initial Results: Error Rates

	KAZ1	***KAZ2***	***KAZ3***	***SNA1***	***SNA2***	***SNA3***	***SHO1***	***SHO2***	***SHO3***	***Overall Error***
GPI	0.00	0.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.04
GAS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GEN	0.67	0.33	0.33	0.00	0.33	0.00	0.67	0.00	0.00	0.26
GMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GCD	NA	0.00	NA	NA	0.33	NA	NA	0.00	NA	0.11
GDA	NA	0.33	NA	NA	0.33	NA	NA	0.67	NA	0.44
NTA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NET	0.00	0.00	0.00	NA	0.00	0.00	0.33	0.00	0.00	0.04
NMT	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NSK	NA	NA	0.67	NA	NA	0.33	NA	NA	0.67	0.56
NSC	NA	NA	0.33	NA	NA	0.33	NA	NA	0.67	0.44
NDA	NA	0.67	NA	NA	1.00	NA	NA	0.67	NA	0.78
NDI	NA	1.00	NA	NA	1.00	NA	NA	0.33	NA	0.78
RUT	NA	NA	0.33	NA	NA	0.33	NA	NA	1.00	0.56
RTR	0.00	0.33	0.33	0.00	0.67	0.67	0.00	0.00	1.00	0.33
RTE	NA	NA	0.33	NA	NA	0.67	NA	NA	1.00	0.67
REV	0.67	0.67	0.33	NA	0.67	0.67	0.67	0.00	1.00	0.52
RDP	NA	1.00	1.00	NA	1.00	1.00	NA	1.00	1.00	1.00
Overall Error GS	0.11	0.13	0.06	0.00	0.13	0.00	0.11	0.08	0.00	0.07
Overall Error NL	0.00	0.42	0.20	0.00	0.50	0.13	0.17	0.25	0.27	0.21
Overall Error RJ	0.33	0.67	0.47	0.00	0.78	0.67	0.33	0.33	1.00	0.51
Overall Error	0.13	0.31	0.23	0.00	0.36	0.25	0.17	0.18	0.40	0.22

Table A.13: GPT-4o (Version 2024-08-06) Improved Results: Error Rates

	KAZ1	***KAZ2***	***KAZ3***	***SNA1***	***SNA2***	***SNA3***	***SHO1***	***SHO2***	***SHO3***	***Overall Error***
GPI	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GSO	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GAR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GEN	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GMS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
GCD	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
GDA	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
NTA	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
NET	0.00	0.00	0.00	NA	0.00	0.00	0.00	0.00	0.00	0.00
NMT	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NSK	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NSC	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
NDA	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
NDI	NA	0.00	NA	NA	0.00	NA	NA	0.00	NA	0.00
RUT	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
RTR	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
RTE	NA	NA	0.00	NA	NA	0.00	NA	NA	0.00	0.00
REV	0.00	0.00	0.00	NA	0.00	0.00	0.00	0.00	0.00	0.00
RDP	NA	0.00	0.00	NA	0.00	0.00	NA	0.00	0.00	0.00
Overall Error GS	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Overall Error NL	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Overall Error RJ	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
Overall Error	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00