National College of Ireland

Enhanced Kubernetes Configuration Generation through Dual-Encoder CodeT5 with Reinforcement Learning and Actionable Feedback

MSc Research Project
Cloud Computing

NagaRavali Ujjineni
Student ID: 24138312

School of Computing
National College of Ireland

Supervisor: Dr Giovani Estrada

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	NagaRavali Ujjineni
Student ID:	24138312
Programme:	Cloud Computing
Year:	2025
Module:	MSc Research Project
Supervisor:	Dr Giovani Estrada
Submission Due Date:	11/12/2025
Project Title:	Enhanced Kubernetes Configuration Generation through Dual-Encoder CodeT5 with Reinforcement Learning and Actionable Feedback
Word Count:	XXX
Page Count:	27

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	19th November 2025

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Enhanced Kubernetes Configuration Generation through Dual-Encoder CodeT5 with Reinforcement Learning and Actionable Feedback

NagaRavali Ujjineni
24138312

Abstract

The multifaceted nature of the Kubernetes YAML configuration system is considered a major cause of cognitive hurdles faced by developers and DevOps engineers, which in turn results in frequent misconfigurations, security vulnerabilities, and deployment failures. Despite the most recent breakthroughs in Large Language Models (LLMs) and the code generation tools based on the transformer architecture having shown good performance in automated code synthesis, the major focus of these systems has mainly been on the production of merely syntactically correct configurations without the provision of quality feedback that can be enacted or the application of the domain-specific best practices. This study tries to fill this void by building an improved CodeT5-based framework that, in addition to generating Kubernetes YAML configurations from natural language specifications, also evaluates the quality and provides specific suggestions for amendments. The following are the innovations we bring forth: first, a dual encoder design is employed that provides better semantic alignment between the linguistic intent and the corresponding Kubernetes configuration. Secondly, reinforcement learning optimisation is utilised based on configuration quality metrics. Lastly, fine-tuning on Kubernetes manifests is achieved through the supplementation of Kubernetes repositories and recognised projects. We identify a CodeT5-as-a-base architecture that is integrated with 100+ real-world Kubernetes manifests. Our proposed enhanced model demonstrates significant improvements in terms of the evaluation framework that incorporates both traditional code generation metrics (CodeBLEU) and novel Kubernetes-specific quality indicator-like configuration complexity scores (CCS).

1 Introduction

The exceptional adoption of cloud computing has completely revolutionised the way organisations take part in the stages of building, deploying, and managing modern applications. The backbone of architectural adjustment is Kubernetes [1]—a free and open-source container management programme that has become the industry benchmark for managing containerised workloads across a variety of infrastructure settings. Kubernetes is a platform that is widely used and has impressive features; however, it also creates remarkable difficulties for those who use it. The process of establishing the platform is mainly done by using YAML [2] documents, which are the declarative specifications that YAML

[1] `https://kubernetes.io/docs/tutorials/`

[2] `https://monokle.io/learn/yaml-basics-a-beginners-guide-to-yaml-manifests`

uses to express the desired state of resources such as Deployments, Services, ConfigMaps and Ingress. Although YAML supports the syntax of human readability, the complexity behind the corresponding code makes the communication obscure. Understanding that the Kubernetes configuration works in a complicated manner is long-term. For example, the process of agreement among dozens of interconnected component resources, where each of them imposes multiple parameters affecting security, networking, resource allocation, and operational behaviour.

The rapidly increasing number of missed deployment cases in cloud configurations is believed to be the dominating factor (Ghorab and Saied; 2025). A recent Gartner's analysis [3] agreed that by 2025, 99% of cloud security failures will happen due to customer misconfigurations. One common type of misconfiguration is negligence in the security context, where resource limits that cause node exhaustion are missing, service networking configurations are wrong, and security policies are not being applied, thus increasing vectors of an invasion. Besides this, the memory "load" enforced on developers by the need to take care of hundreds of options related to configuration, their interdependencies, and the new best practices that emerge subtract seriously from the effectiveness that developers could realise and consequently the reliability of the programmes.

The evolution of Large Language Models (LLMs) and the introduction of transformer-based architectures have brought new avenues of research for the development of automated code generation systems (Raiaan et al.; 2024). Examples of these behavioural changes are GitHub Copilot, Amazon CodeWhisperer, and OpenAI Codex, which have shown off their fabulous amounts of creativity when given codes in various programming languages (Yetiştiren et al.; 2023). In the Infrastructure-as-Code (IaC) area, some specialised systems have been designed to manage Kubernetes configuration generation through the use of other tools (Srivatsa et al.; 2024). CodeT5 (Wang et al.; 2021), a transformer model that is rooted in coding and project development, has been especially robust in tasks like code behaviour, codification, and code flipping out across various specific programming languages.

1.1 Research Gap

There are some situations in which the functionality of LLM-based code generation tools is diminished performance-wise; thus, they cannot be applied to Kubernetes configuration management. First, these programmes chiefly concentrate on just the syntactic correctness, i.e. generating YAML that is parseably produced successfully, besides which they do not focus on the semantic validity or either the usage of the best practices specific to the domain. A syntactic valid Kubernetes manifest is not an exception to expose significant security vulnerabilities, resource contention, network policies violations, or operational requirements. In addition, devices that offer minimal predictive feedback to users are few and far between. When a configuration presented is generated with suboptimal qualities, developers are given poor guidance on the specific improvements, the reasons why they should be implemented, and how this is done practically.

Apart from these issues, it should be mentioned that many systems use a general-purpose coding dataset which makes them non-specific to the configurations of Kubernetes. The special characteristics of Kubernetes configurations that involve the declarative approach, the dependency on resources, the security model, and the operational issues could not be imagined, and these models will not be learning from our experiences.

[3] `https://www.gartner.com/smarterwithgartner/is-the-cloud-secure`

Lastly, traditional metrics used to evaluate code generation systems like BLEU (Bilingual Evaluation Understudy) and exact match, while useful for measuring surface-level similarity, fail to capture configuration quality dimensions that are critical as well.

It has been noticed that the shortcomings mentioned above are being filled by research efforts. It has been suggested (Dubey et al.; 2024) that it is possible to generate intention-based cloud configurations using the LLM as the optimiser, the results being a better alignment between the abstraction of the deployment and the generated manifest. (Pujar et al.; 2023) explored the pioneering route of using LLMs for YAML code generation and setting the baseline for future work on IaC tasks. Other authors (Malul et al.; 2024)) gauged the potential of GPT-4 in Kubernetes configuration tasks with an orientation toward security. However, the studies discussed are mainly concerned with the generation of different artefacts without sufficient quality feedback statements for the end users.

1.2 Research question

The need for tools to automate the deployment and validation of Kubernetes manifest has been identified. In general terms, *it is not clear what modifications can be made to transformer-based language models to not only generate Kubernetes YAML configurations, but also provide actionable quality feedback that helps users learn best practices and reduce configuration errors.* Taking CodeT5 as the baseline encoder-decoder transformer model, three main aspects can be studied:

- What deep learning architecture provides a better alignment of the natural language deployment intent and the generated Kubernetes configurations?
- How effective are reinforcement learning mechanisms for the purpose of optimising the quality of the generated code?
- How can the system provide iterative, actionable, and educational feedback on qualitative configuration issues?

1.3 Research objectives

The goals of this study can be reached through the specific measurable objectives outlined below.

1. **Establish baseline performance (RO1)** A CodeT5-based system will be developed and evaluated as a baseline. A dataset of at least 100 production-grade Kubernetes manifests will be sourced from trusted sources (official Kubernetes example repository, Bitnami Helm charts, CNCF graduated project deployments), and data preprocessing pipelines for natural language intent extraction and YAML normalisation will be implemented. CodeT5 will be fine-tuned in the collected data set, as well as in the established baseline performance using both traditional and new metrics such as CodeBLEU and CCS.
2. **Design and implement enhanced architecture (RO2)** The three major steps that aim for: (a) two-encoder architecture with distinct encoding pathways for natural language intent and Kubernetes configuration pattern recognition, thus allowing semantic alignment between user specifications and generated manifests, (b) the reinforcement learning optimisation module that uses the configuration quality

metrics including syntactic validity and completeness of security context, and resource optimisation as reward functions to fine-tune the model, and (c) the specialised training pipeline based on Kubernetes manifests containing stratified sampling across different resource types and complexity levels.

3. **Develop an actionable feedback mechanism (RO3)** The project will include implementing a comprehensive quality evaluation and feedback generator which: (a) reviews generated configurations based on Kubernetes best practices as set by official industry and technical standards, (b) spotlights quality issues through multiple avenues like security vulnerabilities, resource misconfigurations, and missing or suboptimal operational requirements, (c) provides constructive feedback on the errors made and gives alternative solutions again with the intonation of the reasons that they should be implemented, and (d) the before/after configuration comparisons which underline the impacted specific changes and their side effects.

4. **Design a comprehensive evaluation framework (RO4)** The evaluation methodology will be set that stretches beyond conventional code generation metrics and embraces aspects that are critical for production Kubernetes deployments: (a) code likeness measures, for instance CodeBLEU for assessing generation power; (b) validity cheques concerning Kubernetes including YAML syntax validation, code configuration quality, and confidence scores; (c) complexity and quality indicators including configuration complexity scores, operational readiness assessment (health cheques, logging, monitoring labels) and best practices suggestions; and (d) a comparison of the baseline and the improved models in terms of performance.

1.4 Contributions

This research makes several significant contributions to the intersection of machine learning, software engineering, and cloud-native infrastructure management as presented in Figure 1.

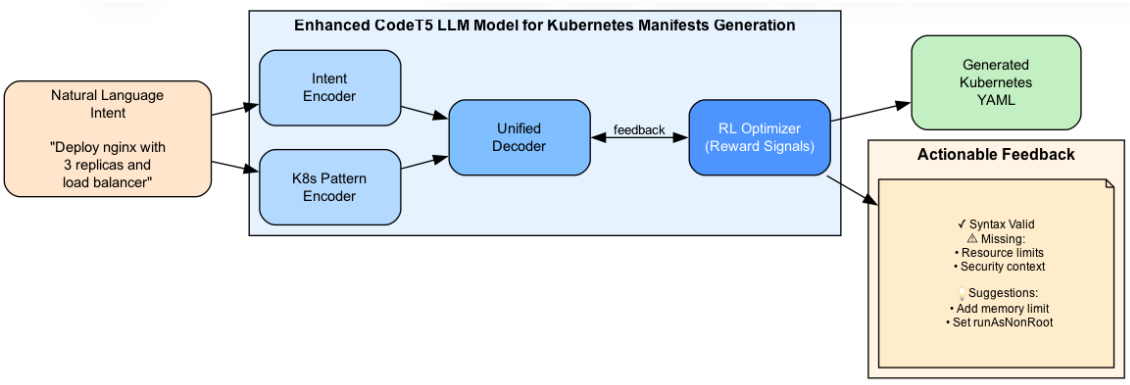

Figure 1: Proposed System Workflow

1. **Architectural Innovation:** We present a dual-encoder architecture that is explicitly targeted at the transition from intent to configuration in the Kubernetes domain. In contrast to standard encoder-decoder models that advance the process of natural language input via the single encoding pathway, our architecture

makes use of different encoders which are specifically optimised for distinct input modalities: one encoder interprets high-level deployment intent, restrictions, and requirements from natural language descriptions, while the other encoder maps Kubernetes configuration patterns, resource relationships, and domain-specific syntax. This meticulously laid out structure is the key to achieving a better level of semantic alignment between the user intent and the configurations produced.

2. **Reinforcement Learning Integration:** We propose a new RL-based optimisation process of IaC generation based on multi-dimensional configuration quality metrics as reward signals. In contrast to common methods of supervised fine-tuning, which only maximise the next-token prediction accuracy, we use domain-specific quality qualifiers such as security compliance, resource optimisation, and operational best practices. The reward dimension incorporates various quality dimensions using the right weight depending on their relative importance in the production environment.

3. **Actionable Feedback Mechanism:** As it is, the code generation tools are more of black boxes, whereby they produce the written code without giving any indication as to the quality of the written code or providing any alternatives. Our system, on the other hand, inverts code generation to the point of handling it as a form of learning as it becomes able to provide clear, specific, and realistic advice.

1.5 Outline

Finally, you can now close Section 1 by outlining the structure of the report, for instance: The remainder of the report is organised as follows. Section 2 presents the relevant theory and works closely related to the proposed one. Section 3 describes the details of the proposed approach. In Section 4, we describe the techniques that are used to address the problem, as well as all the proposed test cases. The details of the implementation are described in Section 5, while the evaluation results appear in Section 6. Finally, we provide conclusions and discussions on future research directions in Section 7.

2 Related Work

The following section is a critical review of the academic literature applicable to Kubernetes configuration generation with transformer-based models. We analyse code synthesis transformer architecture, LLM applications to Infrastructure-as-Code and reinforcement learning optimisation methods. Critical review of the literature on available strategies helps us determine the shortcomings that drive our study and determine the distinctiveness of our submissions.

2.1 Transformer Architectures and Code Generation Models

The proposition of automating code synthesis with natural language has created a revolution in programming through the progress in large language models. Some authors Chen et al. (2021) presented Codex, a GPT-based LLM that is fine-tuned on 159 GB of code from GitHub, written in Python, and reached 28.8% pass@1 and 72.3% pass@100 on the HumanEval benchmark, which consists of 164 handwritten programming problems.

The authors created the pass@k metric with the aim of evaluating functional correctness through unit test execution instead of using token-level similarity measures like BLEU, which they showed to correlate poorly with code functionality. Codex, which adopts a 12 billion parameter-only decoder architecture with nucleus sampling for generation, has some shortcomings, including susceptibilities to prompt phrasing and decreased performance on tasks involving multi-step reasoning that is not straightforward. The model was not assessed on generating Kubernetes YAML, which can be categorised as Infrastructure-as-Code tasks at a domain-specific level, which adds an incentive to the special approach of generating declarative configuration languages.

In cross-programme pre-training, which is the practice of pre-training on both programme understanding and generation tasks, ((Ahmad et al.; 2021) brought PLBART, a generic sequence-to-sequence model, to the forefront based on the BART architecture with a 6-layer encoder and a 6-layer decoder (140M parameters). The model was pre-trained on 470M Python and 210M Java functions from GitHub along with 47M Stack Overflow posts using denoising autoencoding objectives, including token masking, deletion, and span infilling. PLBART achieved state-of-the-art results in code summarization tasks with an overall BLEU-4 score of 18.32 after being evaluated in six programming languages, surpassing previous models based only on the encoder, such as CodeBERT. The model was also effective on code translation tasks, specifically between Java and C, grading 83.02% BLEU and a 64.60% exact match for the Java-to-C transition, making it 9.5 points better than CodeBERT. Simultaneously, PLBART's requirement for large-scale parallel data was a bottleneck for flexible fine-tuning on downstream tasks, and its integral encoder-decoder model structure was later found to be suboptimal for task-specific applications compared to dedicated architectures.

The pre-trained language model approach to programming and natural language was first introduced by (Feng et al.; 2020) by means of CodeBERT, which is a bimodal pre-trained model built on the BERT architecture (125M parameters). CodeBERT was trained on 2.1M bimodal code-text pairs from CodeSearchNet in six programming languages using masked language modelling and replaced token detection objectives. On associated code search tasks, CodeBERT was found successful with an average Mean Reciprocal Rank (MRR) of 69.3% for six languages in the context of CodeSearchNet, validating its semantic representation of code as understanding. As far as code-to-text generation is concerned, the model received 17.83 BLEU-4 points on summarization tasks; encoder-only designs, however, have shown much less effectiveness than encoder-decoder or decoder-only models. The CodeBERT model not only laid down solid baselines for code-text alignment but also missed out on incorporating key structural code features like abstract syntax trees and data flow graphs that found their place in later models like GraphCodeBERT.

Regarding the inherent code structure, (Guo et al.; 2020)noted that existing pre-trained models mostly generated code as sequential tokens while disregarding it. To that end, they came up with GraphCodeBERT, which uses data flow graphs to depict the semantic-level code structure. The model consists of a parameter encoder architecture that was pre-trained with masked language modelling on tokens from code and edges in data flow graphs. It also contained data flow edge prediction and node alignment tasks. GraphCodeBERT took 71.3% MRR in CodeSearchNet retrieval tasks, exceeding the 69.3% of CodeBERT, and also had a 97.1% F1 score measuring clone detection tasks. In the tangible context, in the programme repair task, the model was able to get an exact 17.30% match on the small Java dataset and 9.10% on the medium dataset, which are

decent results for structure-aware pre-training. However, despite the GraphCodeBERT encoder capturing the semantic information through data flow, its randomly initialised decoder is unaware of the code structures during the generation tasks, and this is the reason why the model does not work effectively for this type of code synthesis.

The (Ren et al.; 2020) study focusses on traditional evaluation metrics such as BLEU that fail to capture the syntactic and semantic properties that are specific to code, and therefore CodeBLEU was proposed, which is a syntax-aware evaluation metric for code synthesis. The CodeBLEU metric is a blend of weighted n-gram matching (BLEU) with the abstract syntax tree (AST) and the data-flow match scores to evaluate both the structural and semantic correctness of the code that it generated. The evaluation highlighted that CodeBLEU has a better correlation with functional correctness compared to standard BLEU in code generation tasks, which was shown through correlation coefficients of 0.58 vs 0.42. The metric is designed in a way that gives equal weights of 0.25 to n-gram matching, AST matching, data-flow matching, and keyword matching parts. Thus, it gives a multi-dimensional appraisal of code quality. However, CodeBLEU still needs ground-truth references to be provided, and it cannot directly ascertain execution correctness, which basically means it has to resort to complementary metrics like pass@k for a full evaluation of the functionality of the generated code.

Based on the T5 architecture, (Wang et al.; 2021) gave CodeT5, an encoder-decoder model with identifier-aware pre-training objectives, which is made to deal specifically with code understanding and generation. CodeT5-base (220M parameters) implemented masked span prediction, identifier tagging, and masked identifier prediction during pre-training on CodeSearchNet. The language model achieved 19.55 BLEU-4 in code summarization and 64.73% MRR in Java code retrieval. The CodeT5 model introduced identifier-aware denoising, which treats code identifiers differently from other tokens so that it can distinguish and make better use of identifiers during code generation with 38.52 CodeBLEU on text-to-code generation. The CodeT5 model demonstrated its flexibility by multi-task learning, which improved code refinement tasks by 21.61% exact match on medium Java programmes compared to 16.40% for CodeBERT. In spite of the effectiveness of CodeT5 on different tasks, its inherent limitation in the case of compact span denoising tasks was the main reason for it being less effective in the case of auto-regressive generation tasks such as next-line code completion with respect to pure decoder-only models.

An evolution of the CodeT5 framework, CodeT5+, was presented by (Wang et al.; 2023), which is and is a family of encoder-decoder models with enhanced flexibility to operate in encoder-only, decoder-only and full encoder-decoder modes through various pre-training objectives. CodeT5 + added causal language modelling in parallel with span denoising during stage 1 and added contrastive learning and text-code matching tasks during stage 2 on 51.5B tokens across nine programming languages in its bimodal pre-training. The instruction-tuned CodeT5+ 16B has achieved incredible performance with 35.0% pass@1 and 54.5% pass@10 on the HumanEval tasks. This is better than the valued code-cushman-001 from OpenAI and left the model at the top of the list among open-source code models. In the case of the retrieval task, the CodeT5+ 770M model achieved an overall MRR of 77.4% on CodeSearchNet, which is 3.0 points more than UniXcoder with the help of its objective of text-code matching that captures the fine-grained cross-modal alignments. The model showed compute-efficient scalability by obtaining frozen checkpoints from CodeGen for the decoders while training only shallow encoders and cross-attention layers, thus signifying reducing the trainable parameters

while maintaining robust performance in 20+ code-related benchmarks.

2.2 LLMs for Infrastructure-as-Code and Kubernetes Configuration

With emphasis in the ease of use and dependence model control of a framework providing an intelligent orientation (e.g. automating Kubernetes configuration by means of a non-expert natural language) (Dubey et al.; 2024) suggested the approach of the large language model to assist in the deployment of YAML generation. The technology applied was Mistral-7B-Instruct-v0.3 with zero-shot and few-shot prompting strategies, which yielded the 100% success rate of schema validation via kubectl with few-shot prompting, compared to zero-shot prompting 97.18% and human-generated configurations (95.77%). For Kubeval's semantic validation, the framework showed 91.55% success rates with both the few-shot and zero-shot approaches, while human-generated configurations performed at 87.32%. The system framework combined various components such as a configuration repository, an intention extraction module, an intent archive, and a validation engine that enabled the execution of syntax validation, key-value pair verification, kubeval cheques, and kubectl dry-run tests. Besides the achieved strong validation results, the authors pointed out several challenges, such as the variability of output consistency because of LLM non-determinism and the requirement of stronger validation mechanisms to look for inconsistencies. The framework was noted for its high efficiency in simple deployment scenarios; however, it required significantly nested orchestration resource management approaches for complex deployment on a single-container basis.

Recognising that Kubernetes misconfigurations are the main security threat to native cloud environments, (Ghorab and Saied; 2025) presented GenKubeSec, a method that uses large language models that is end-to-end for detection, localisation, reasoning about, and remediation of problems. The work established a common misconfiguration index (UMI) that integrates 169 standardised misconfiguration categories through entity matching across three tools (Checkov, KubeLinter and Terrascan) that are rule-based and industrial-standard. The GenKubeDetect component obtained 0.990 precision (on par with rule-based tools) and 0.999 recall (better than the individual tools) in the 276,520 Kubernetes configuration files under analysis, which indicated the efficacy of fine-tuned LLMs above the pretrained models. The GenKubeResolve component was responsible for location, reasoning, and remediation with 100% accuracy in the sample configurations, which were validated by experts, providing educational value in addition to detection alone. The most serious constraints were the 512-token context window of CodeT5p-770M, which limited the analysis to only 83% of the collected configurations, and the model's trouble in detecting minor misconfigurations of access control (Role, ClusterRole, RBAC) and NetworkPolicy specifications. The strategy of not using API-based models and instead relying on local-run LLMs addressed serious significant issues of security and privacy while at the same time being free of the API call overhead.

Using the community-driven code generation approach, Ansible Wisdom was introduced, a transformer-based system for generating Ansible YAML configurations from natural language prompts (Pujar et al.; 2023). This project is a practical application of language models in automation IT tasks, a section of the LLM technology where the Ansible automation markup language gap existed. The WISDOM-ANSIBLE-MULTI 350M model, which is specifically trained on those files and GitHub BigQuery, got a 54.51 Ansible Aware score and 46.58 BLEU in a few-shot evaluation that surpassed OpenAI

Codex, which had a 48.78 Ansible Aware and 50.40 BLEU with the use of significantly fewer parameters. The model fine-tuned on a total of 112K quality Ansible Galaxy playbooks and task files reached 70.79 Ansible Aware and 66.67 BLEU scores, which is a 15.28-point improvement compared to the CodeGen-Multi baseline. The evaluation metrics were introduced in the study, which are specific to the Ansible programming language: Ansible Aware (which is a measure of module, keyword, and parameter accuracy) and Schema Correct (which looks for compliance with the Ansible schema without a reference comparison). The research reformulated code generation as code completion by utilising Ansible's name field containing natural language task descriptions, demonstrating superior performance over prefix-based prompting (70.79 vs 45.87 Ansible Aware). On the positive side, their methodology of the code design model for the Ansible domain performed well under strict guidelines. However, they have not had the opportunity to prove their worth in open-format configuration situations and in cases with multiple file dependencies.

After working on effective misconfiguration detection research, (Malul et al.; 2024) introduces the GenKubeSec initiative as a comprehensive LLM-based system for the containment of Kubernetes configuration errors, including not only their detection but also their localisation, reasoning, and remedial actions. The system has two parts; it consists of GenKubeDetect (the multi-label classifier is fine-tuned CodeT5p-770M), and the performance results are 0.990 precision and 0.999 recall in comparison with the toolset. Checkov, KubeLinter and Terrascan, which was verified by manual expert analysis, disclosed that 83.44% of false positives were actually true positives. It is a manifestation of superior generalisation. GenKubeResolve (an adapted Mistral LLM with few-shot learning and prompt engineering) allowed localisation, reasoning, and actionable remediation to be discussed at the level of item lines in 30 configurations drawn from a random sample, which were validated with 100% accuracy by experts. The unified misconfiguration index (UMI) standardised 169 unique misconfiguration types through GPT-4-based entity matching across multiple detection tool taxonomies, addressing critical label inconsistency issues in the field. Data preprocessing that is new and applicable includes label encoding ("app:52"), reducing token counts for efficient training, object-level segmentation, allowing the analysis of configurations that exceed token limits, and injection of synthetic misconfigurations through mutation operations (insertion, deletion, and modification) to class imbalance. The only adjustment that made the system react to the new misconfiguration was the label of the examples of the misconfigurations because they were the only ones that needed change, but on average it was just 2-5, and after successful training the system precision went to 0.8+. Thus, the approach is not only supple and open-ended, but also very constructive in its deployment.

2.3 Reinforcement Learning Optimization and Quality Improvement

The research by (Bai et al.; 2022) attempted to address the issue of training AI assistants to be effective and minimize the human oversight required in the process. They proposed the Constitutional AI (CAI), which is the training of language models by means of self-generated critiques as well as AI-provided feedback in accordance with the principles laid down in a constitution. The essence of the method is integrating supervised learning performed in the critique-revision cycles and reinforcement learning from AI feedback (RLAIF), wherein a 52B model provides critical comments and revises its own responses

by following 16 constitutional principles, thereby obtaining the Elo scores 50-100 points higher than the standard RLHF models. Conversely, e.g., in the supervised phase, models would generate critiques and revisions needed for harmful content removal from the initial responses, thus achieving progressive improvements in harmlessness with each revision cycle, while the RL phase uses AI-generated preference labels for teaching reward models without human harmfulness annotations. It was verified that the reason-chain of thought functioning actually contributed to both the process of generating critiques and the evaluation of preferences. This in turn improved the efficiency of the models, so much so that the correct performance, the models being helpful-harmless-honest, reached as high as 70-75% compared to only 50-60% of their smaller relatives. The primary drawbacks are the possible over-optimisation issues, which can lead to excessive caution or preachy answers; the model's habit of including boilerplate security language after long stretches of RL training; and limited transferability since the principles were not developed through stakeholders' engagement but were selected ad hoc for the research purposes. The potential to use multi-objective reward signals as showing how to balance conflicting criteria is the project that config generation will benefit from directly. However, the challenge of adapting constitutional principles for technical correctness versus safety alignment represents a problem of a different order.

A new solution to conventional supervised fine-tuning problems that do not capture execution feedback signals through the CodeRL project was introduced by (Le et al.; 2022). The project is a fusion of pre-trained language models and deep reinforcement learning and leverages unit test outcomes as reward signals through an actor-critic framework. The approach regards CodeT5-770M as an actor policy that outputs code samples and that the critic model (CodeT5-small) estimates appropriate functionality and also gives token-level value predictions which, on the one hand, guide actor optimisation and, on the other hand, are the main source of the achieved results of pass@1 2.57%, pass@5 6.21% and pass@1000 19.36% on the APPS benchmark. The model assigns rewards according to the outcome of compilation and execution; for instance, +1.0 if all tests pass, -0.3 for failed tests, -0.6 for runtime errors and -1.0 for syntax errors. The probabilities predicted by critics provide intermediate token-level feedback that helps to reduce the variance. During the inference process, the Critic Sampling procedure cheques the generated programmes against the example unit tests, removes the low-scoring subsequence, and then uses the ones with high scores to condition the increased regeneration of improved programmes. In fact, the project was only able to match AlphaCode's performance at k=50,000 with k=1000 samples just by using the critic guidance. It also has the additive advantages of being normalised, so the baseline pays off, and the critic-based intermediate returns are optimal; thus, critic guidance contributes to the results with pass@1 improvements of 0.25-0.35 across difficulty levels. The training process lacked its stability due to the feedback mechanism of the actor-critic loop, which needed a careful warm start with imitation learning, the modest improvement (0.69% pass@1) of competing-level problems versus introductory tasks (6.77% pass@1), and the lack of evaluation in declarative configuration languages like YAML, where execution feedback mechanisms are fundamentally different from imperative programming.

Extending offline RL approaches to online frameworks with richer feedback signal, (Liu et al.; 2023) introduced RLTF (Reinforcement Learning from Unit Test Feedback), an online RL system utilising multi-granularity feedback for code generation. RLTF's structure accommodates dynamically updating online buffers congregating fresh training samples, which are created instantly while the model trains, reaching improvements of

1.45% pass@1, 3.78% pass@5, and 19.92% pass@1000 on APPS benchmarks and outperforming that of CodeRL's offline approach by 0.15-0.46%. The method encompasses three types of feedback granularity: coarser-grained feedback, which assigns episode-level rewards based on the overall test outcomes; fine-grained feedback that categorises errors into Uglobal (logic issues affecting the entire code), Uline (specific line errors like IndexError), and Uignore (ambiguous errors), as well as penalising specific code segments providing corrective actions; and also adaptive feedback that involves graduated rewards proportionate to test case pass rates. Until recently, the prevailing fine-grained feedback designs have given the largest performance boost, while the online framework advantages were more pronounced combined with RLTF versus offline settings, with temperature 1.0 sampling during training enabling broader exploration than conservative 0.2 sampling. The approach has proven to be robust with various base models, where CodeGen-2.7B reached a pass@1 of 2.04% (up from 1.64% without RLTF) and the zero-shot transfer to MBPP yielded a pass@1 rate of 30.4%, which is higher than GPT's 22.4%. The limitations of the project are first, additional categories of Python sub-error types have been introduced, but explaining them limits the applicability to other programming languages; second, semantic errors have been the least affected in the improvement process compared to syntax and runtime ones; and finally, the overhead of 12-hour-plus GPUs for 8 V100 GPUs and 3 extra machines for sample generation have directly impacted training costs.

The misalignment of the next-token prediction objective estimator with the user intent was the main driver of the invention of InstructGPT by (Ouyang et al.; 2022), who developed and fine-tuned GPT-3 with reinforcement learning from human feedback attachment to the helpfulness, harmlessness, and honesty aspects. This three-step solution consists of controlling the fine-tuning in 13K labeler demonstrations, training a 6B reward model in 33K pairwise comparisons of model outputs, and optimising a policy by using PPO with per-token KL penalties ($\beta = 0.02$) to prevent over-optimisation of the reward model, where 1.3B InstructGPT was preferred to 175B GPT-3 outputs 85±3% of the time even though it had 100x fewer parameters. Performance improvements were great, such as a 21% decrease in hallucinations on closed-domain tasks compared to 41% for GPT-3, more honest and informative responses in TruthfulQA and, when asked respectfully, a 25% reduction in toxic output in RealToxicityPrompts, which were measured both by human evaluations and Perspective API. To prevent performance regressions of the public NLP datasets, which were observed during pure PPO training, the variant PPO-ptx was mixed with pretraining distribution updates, consoled them with HellaSwag, and maintained preference improvements, but did DROP or SQuADv2 outpace GPT-3? The 60 petaflops-days cost of training for 175B PPO-ptx compared to 3,640 for GPT-3 pretraining implies that alignment techniques can offer cost-effective alternatives over raw scaling, and it was found that the labeler agreement rates of $72.6 \pm 1.5\%$ provide inter-annotation reliability as the norm. One serious problem involves the lack of diversity in the English training set (∼96% of the data), dependence on the preferences of 40 contractors whose values

(Use your citations in Table 1) **ToDo**

2.4 Research Gap Analysis and Contributions

The reviewed literature shows the considerable advancement in transformer-based code generation models, with work like CodeT5+ that secures the first place in general programming tasks. The recent programmes on Infrastructure-as-Code have been very

Table 1: Comparison of key papers with our work addressing Kubernetes configuration generation and quality assessment

Paper	**Key Contribution**	**Limitations**	**How Our Work Addresses**
Ghorab & Saied (2025)	LLM-based misconfiguration detection with 0.999 recall using CodeT5p-770M; Unified Misconfiguration Index with 169 categories	Detection-only with no generation; 512-token limit restricts 17% of configurations; No actionable feedback or learning component	Generation with multi-dimensional quality assessment; Educational feedback with explanations and recommendations
Malul et al. (2024) - GenKubeSec	Two-component system achieving 0.990 precision; Provides localization, reasoning, and remediation with 100% expert validation	Focuses on remediation rather than natural language generation; No architectural innovations; Limited to existing patterns	Novel dual-encoder architecture separating intent and pattern encoding; Proactive generation from scratch
Dubey et al. (2024)	Intent-based framework using Mistral-7B-Instruct; 100% schema validation with 91.55% semantic validation rate	No quality feedback mechanism; Limited to single-container deployments; Evaluation restricted to validation metrics only	Multi-dimensional evaluation including security and resource efficiency; Comprehensive feedback across multiple dimensions
Pujar et al. (2023) - Ansible Wisdom	Transformer-based YAML generation achieving 70.79 Ansible Aware and 66.67 BLEU; Fine-tuned on 112K files	General Ansible focus, not Kubernetes-specific; No feedback mechanism; Missing security and operational assessment	Kubernetes domain specialization; Production-grade dataset with 100+ curated configurations
Wang et al. (2023) - CodeT5+	Multi-stage pre-training on 51.5B tokens achieving 35.0% pass@1 on HumanEval; Flexible encoder-decoder architecture	General-purpose without IaC specialization; No configuration quality feedback; Focused on functional correctness	Domain-specific fine-tuning on Kubernetes YAML; Configuration-specific metrics including CodeBLEU and Complexity Score
Le et al. (2022) - CodeRL	Actor-critic framework achieving 19.36% pass@1000 on APPS; Critic-based sampling with token-level value estimates	Designed for imperative programming; Unit test rewards unsuitable for declarative configs; No feedback generation	RL-based quality optimization with custom rewards; Configuration-aware rewards for CIS compliance and best practices
Chen et al. (2021) - Codex	Pioneering pass@k metric; 72.3% pass@100 on HumanEval; 12B parameter model on 159GB GitHub code	Not evaluated on IaC tasks; Functional correctness focus inappropriate for YAML; No feedback generation	Declarative language focus with specialized YAML evaluation; Security, efficiency, and operational readiness metrics

results-orientated, as evidenced by the fact that GenKubeSec achieving 0.999 of Kubernetes misconfiguration detection and Ansible Wisdom computed a score of 70.79 in Ansible Aware on YAML generation. The execution of RL strategies in the field of quality assurance has become a confirmation of the feasibility of their use, as shown in the 19.92% pass@1000 RLTF performance. Nevertheless, these particular progressions are

still overshadowed by the absence of results on the utilisation of these methods to improve domain-specific configuration generation with quality feedback mechanisms. The key differences between published literature and the present research report are:

- The present Kubernetes configuration generation systems such as the one built by (Dubey et al.; 2024) are capable of high performance through robust validation (schema validation at 100% with few-shot prompting), but do not provide any educational feedback to the user with respect to the quality of their schema or suggest improvement strategies which hinder their value for developer training and iterative refinements.

- The measures for traditional code generation such as BLEU, CodeBLEU, and pass@k are primarily designed to reflect syntactic correctness and functional accuracy on the part of the imperative code; in the process, they overlook the distinct quality dimensions that are highly relevant for a practical case like Kubernetes deployment (e.g., adherence to cloud-native best practices).

- Although general code models apply dual-encoder or RL techniques, no research has examined the development of design architectures that specifically encode natural language deployment intent and Kubernetes configuration patterns to which declarative YAML generation will benefit from the improved semantic alignment.

- Infrastructure-as-Code generation systems are black boxes with the current operations; they just give you the generated configurations but never mention your quality issues, security implications, or suggest concrete improvements which are unlike the educational AI assistants that balance the two aspects.

While the existing work has significant evidence of LLM being utilised in general-purpose code generation and Kubernetes misconfiguration detection, little is known about how architectural enhancements (dual-encoder designs, RL with domain-specific rewards) and specialised training on Kubernetes manifests can improve configuration quality across multiple dimensions (security, resource efficiency, operational readiness) while simultaneously providing actionable, educational feedback that helps users learn cloud-native best practices and reduce configuration errors in production deployments.

Through three primary contributions directly related to the research gaps addressed and aligned with the objectives stated in Section 1.2. This research is designed to cover a complex intervention implemented at different levels. The first research objective (RO1) is to establish CodeT5 as a baseline for performance in relation to the generation of Kubernetes YAML using 100 production-grade manifests. Then, RO2 denotes the development of a dual-encoder design with RL optimisation that is enhanced by the use of configuration quality metrics as reward signals Finally, RO3 illustrates the implementation of a feedback system that benchmarks the output of configurations against given, actionable recommendations with before-and-after illustrations, and (RO4) entails expounding on a more exhaustive evaluation framework that does not only involve traditional metrics (BLEU, CodeBLEU) but also production-relevant metrics such as security assessment, complexity scores, and operational readiness.

Hints

✘ Do *not* copy/paste sections from your research proposal (RIC CA2) blindly. It is unlikely that the old content is suitable at this stage of your work (flagged as

self-plagiarism!).

✘ Do NOT start sentences with citations. The citation should be at the end of the sentence (George et al.; 2025). Or, the citation should appear (George et al.; 2025) in the middle of a sentence.

✓ Only include references *from reliable academic sources* (IEEE, ScienceDirect, Springer). Do not be distracted by the application area, software libraries, hardware, etc. Focus on the bottom line of your *research question*.

✓ Based on your literature review, highlight the most similar or closely related paper to your own research project.

You are expected to provide a critical/analytic synthesis of recent academic research (especially surveys and review papers of the last 5-10 years). Comment on the strength and weakness/limitation of work in reviewed papers.

Note: LaTeXtemplate

If you decide to use Microsoft Word, pay attention to the correct citation style. Also, the **References** section, at the end of the document, is not numbered. A common mistake is to write incomplete references, place them at the beginning of sentences, or write trivial sentences to use them (Kumar et al.; 2023; Nguyen et al.; 2020). (Wang et al.; 2021)

- Change your name, studentID, and document title in the file called researchProject.tex, line 56, see placeholders "Forename" and "Surname". Do not modify the template itself.
- It is worth mentioning that your citations must be included in the *refs.bib* file. It is in *bibtex* format, open it up to see what it looks like, and populate it with your own references.
- A nice way to leave notes and reminders for yourself: **(Use *todo* to document ToDo reminders)**

Note: Figures

Use figures to make your content more appealing, but always cite the source of your figures. Make sure you always describe what the figure is about in your document and figure captions, see, for instance, Figure 2.

Figure 2: Notice figure captions appear **below** the figure. It should summarise a result or major finding – guide the reader to that interesting bit, without having to read the entire report! Cite the source, if appropriate.

A very easy way to drop marks in your report is to include low-quality screenshots with unreadable text. Prepare professional high-resolution images. Save them and then include them in your report.

Note: Key Paper(s)

After writing the previous subsections, it might be convenient to summarise your **key papers** into a table. In Table 1 an example is provided with four **key papers**. Type the table, do *not* insert screenshots. As you will see below, it will help you to contrast the **key papers** with your own work.

Table 2: Notice table captions appear **above** the table. It should summarise a result or major finding – What makes your own work different to other papers?

Paper	**Description & Drawbacks**	**Accuracy %**
George et al. (2025)	proposed X;	
	used very limited data	93.65
Kumar et al. (2023)	introduced Y;	
	did not show accuracy on validation set	90.01
Nguyen et al. (2020)	said to improve Z;	
	did not provide details to validate claims	92.50
Repina et al. (2025)	proposed a novel W;	
	only tested binary classifiers	98.99

Provide a brief and crucial difference(s) with your own work. Say what your work *is about*, and what *is not*. The literature review will guide the reader towards the drawbacks you worked on. For example, Table 1 suggests that you:

- will use large scale data;
- will use a validation set;
- will provide enough implementation details; and
- will focus on multi-class problems.

While the baseline included support for C# and Java, the work here presented has been extended to Python. The baseline paper and code has an incorrect Blue score metric and a better one has been included in this work.

Finally, the literature review must end up with a paragraph saying why previous solutions were not adequate, and thus justifies the need for further research (yours!). Typically, your one-liner **research gap** will go about: "X thing has never been explored", "little is known about X", "nobody knows how to do X", etc. It must be aligned with your research question(s) and objective(s), written in Sections 1.2 and 1.3.

Note: Citations

The correct style for your references is the one from this template. In other words, do not change the template. If you are going to use Word, follow the style closely. When you cite somebody else's work, keep in mind the following:

✓ References in tables, e.g. Table 1, look good using the command “cite”. The result appears as: Repina et al. (2025).

✓ If you insert the citation within a paragraph, the command is “citep”. The result will appear as: (Repina et al.; 2025).

✓ Let the reader know where the information came from. In other words, all claims must be supported with a proper reference.

✘ Do *not* start your sentence with a citation. The citation should be placed in the middle or end of your sentence.

3 Methodology

The research methodology has been based on Knowledge Discovery in Databases (KDD), which is a linear process involving selection, preprocessing, transformation, data mining, and interpretation of the patterns from data to extract the patterns that can be acted upon. The KDD approach is the best fit for this particular research work since it mainly focuses on pattern detection and knowledge extraction and not on the iterative business understanding cycles, which is exactly our goal of discovering optimal configurations and quality assessment patterns in Kubernetes manifests. Our adapted KDD-based research pipeline, consisting of five phases from data collection through knowledge discovery, is illustrated in Figure 3.

3.1 Data Collection and Acquisition

for this research. Data collection ventured into three essential sources: Kubernetes repositories on GitHub, including Kubernetes/examples and the Kubernetes-sigs projects; the most widely adopted Helm charts by Bitnami and ArtifactHub; and the production configurations of Cloud Native Computing Foundation (CNCF) graduated projects like Istio, ArgoCD, and Knative. To limit the selection of sources the criteria included increased GitHub stars, activity in the past six months and observable signs of production deployment to ensure applicability in the real world and quality standards.

The GitHub API was used to fetch YAML manifest files from the identified repositories systematically. The data collection emphasized a range of Kubernetes resource types involving deployments, services, config maps, stateful sets, and daemon sets in order to provide a comprehensive overview of common infrastructure patterns. The dataset creation was focused on including more than 100 production-grade manifests, distributed among three levels of complexity: simple configurations with single resources and minimal specifications, medium configurations with interconnected resources, and complex configurations with various patterns and large customization. This stratified sampling makes sure that the model is trained on multiple configuration scenarios, like the one done by (Pujar et al.; 2023), in which they used an LLM to make automated YAML documents.

3.2 Data Processing Pipeline

The manifests collected were subjected to high-level preprocessing to confirm data quality and model effectiveness. The initial validation was performed using PyYAML for

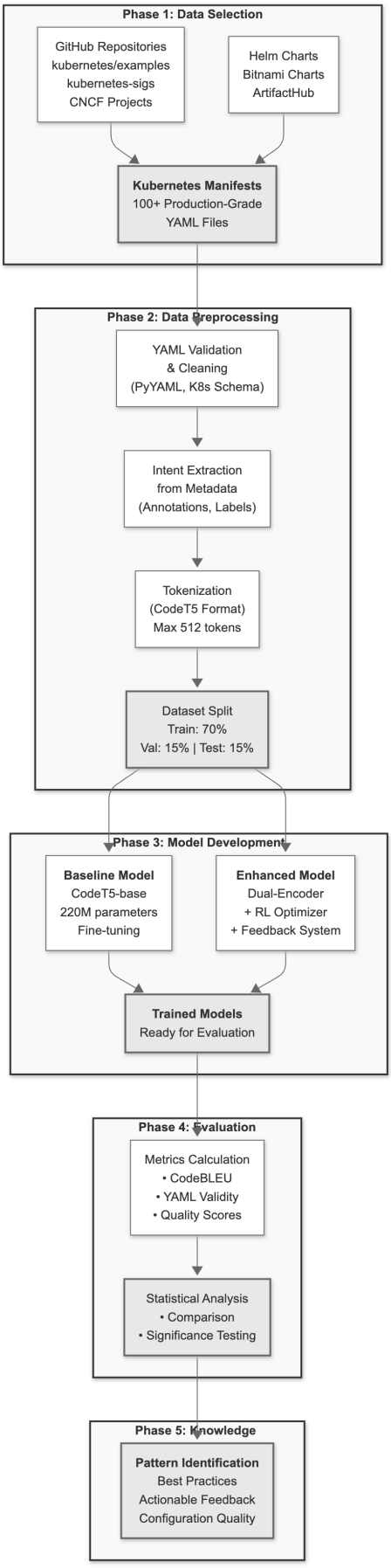

Figure 3: Adapted KDD-based Research Pipeline

syntax verification and Kubernetes-Python client libraries for schema validation against Kubernetes API specifications. The manifests that had syntax errors, deprecated API versions, or incomplete resource definitions were captured and excluded from the dataset. This validation exercise removed approximately 15% of the original collection, ensuring that only deployable configurations passed into the training pipeline.

The intent extraction was a crucial preprocessing step since it provided natural language specifications that would be modeled. The extraction process examined multiple components of the manifest, e.g., metadata annotations, resource naming conventions, label semantics, and structural patterns, to produce the natural language descriptions. For instance, a deployment manifest called "nginx-production" with three replicas and resource limits would generate the intent description "Deploy nginx application in production environment with three replicas and defined resource constraints." The similar approach was also followed in (Malul et al.; 2024) in GenKubeSec, which illustrated how crucial context extraction is to configuration comprehension.

The preparation of tokenization consisted of preparing intents and manifests extracted and transforming them to the configuration of CodeT5 input. It was also a modification of CodeT5 designed to operate on structured code and natural language. The pretrained tokenizer CodeT5 is intended to deal with both natural languages and structured code and focuses more on identifiers and structural tokens. The maximum sequence length was configured to 512 tokens, which can fit the majority of Kubernetes settings and control computational limits. The manifests which did exceed this limit were intelligently truncated to leave resource specifications vital and annotated optional resources deleted where appropriate. The resulting processed dataset was then stratified and sampled into training (70%), validation (15%), and test (15%) sets to ensure that the piece of resources and complexity levels were proportionately represented in all splits.

3.3 Model Development Methodology

The model development follows a two-stage process that includes establishing a baseline performance benchmark and implementing architectural enhancements as presented in Figure 4 . The baseline model uses CodeT5-base (220M parameters), a model that has shown strong performance on code generation tasks through a special pre-training approach that is aware of the identifiers and has a unified encoder-decoder architecture (Wang et al.; 2021). The fine-tuning of the baseline model used the Kubernetes-specific dataset and was based on the AdamW optimizer, with a learning rate of 5e-5 and a batch size of 8, and was trained for 10 epochs with early stopping based on validation loss measure. The fine-tuning objective is to minimize the cross-entropy loss between the predicted tokens and the ground-truth YAML sequences through the association of the model with Kubernetes-specific patterns and the corresponding conventions. The improved design is based on three new components, as seen in Figure 4. A dual-encoder system, a reinforcement learning (RL) optimizer, and a feedback mechanism are the basic elements. The dual-encoder system decouples intent recognition and pattern recognition through the use of specialized encoders. The intent encoder has a BERT-style transformer-based architecture that takes in natural language specifications and produces intent embeddings that are 768-dimensional and represent the user requirements. The pattern encoder, in its turn, will use the pre-trained CodeT5 encoder to decode Kubernetes template patterns, which yields 768-dimensional pattern encodings, which are interpreted as structural and semantic knowledge about configuration. These complementary encodings get concaten-

ated into a 1536-dimensional representation, which in turn feeds into the unified decoder, a CodeT5 decoder that has been enhanced with cross-attention mechanisms to merge intent and pattern information during the generation process.

The RL optimizer deals with a limitation in standard supervised learning pipelines, which is the disconnect between training objectives and deployment requirements. Instead of just optimizing for token-level accuracy, the RL component introduces a multi-faceted reward function that evaluates the configurations produced from a practical deployment perspective. This reward function is made up of three weighted components: a validity score evaluating the correctness of YAML syntax as well as Kubernetes schema compliance; a quality score accounting for best practices, including security contexts, resource limits, health probes and complexity penalty; and lastly, further complex configurations when simplicity is appropriate. This optimization through rewards has its basis in DRL, which has been applied with execution feedback to CodeRL (Le et al., 2022) with a substantial impact through improved code generation. The policy gradient optimization process updates model parameters in a way that maximizes expected rewards, which leads to the behavior of the generation process getting in line with production deployment requirements, rather than just mimicking the training examples.

3.4 Evaluation Methodology

The evaluation module includes multiple metrics that differ from one another, addressing different aspects of generation quality. CodeBLEU is the primary code generation metric, which combines n-gram matching, weighted n-gram matching, abstract syntax tree matching, and data flow matching to present an inclusive code similarity assessment (Ren et al.; 2020). By considering both the code structure and semantics, this metric has established itself as the standard for code generation system evaluation, which is more nuanced than traditional BLEU scores. The YAML validity rate shows what part of the configurations are syntactically correct and schema-compliant, which is a reflection of basic functional correctness that is a must for deployment.

In addition to the conventional metrics, this research proposes Kubernetes-specific quality indicators that are in alignment with the standards for production deployment. Configuration Complexity Score (CCS) is the metric that measures the complexity of configurations through the use of a weighted sum on the number of resource types, nested structures, and inter-resource dependencies; this adds to the transparency concerning maintainability and operational overhead. The metrics of best practices adherence contemplate whether security contexts (that block privileged access), resource limits (that reserve resources), health probes (which are for self-healing), and labelling (which helps manage and monitor) are present or not. Security compliance assessment is done by comparing configurations with CNCF best practices and relevant CIS Kubernetes Benchmark recommendations.

The experiment has been carried out in Google Colab with GPU acceleration (Tesla T4 with 16 GB memory) that ensures reproducible execution environments that can be accessed by the research community.The training and inference are done with PyTorch 1.13 and the Hugging Face Transformers library version 4.26 which is compatible with pre-trained CodeT5 models. The paired t-tests are used to compare the results of the baseline model with those of the improved model in order to establish whether the results are statistically significant. Ablation studies are conducted on the components of enhancement (dual-encoder, RL optimization, feedback system) to quantify their respect-

ive contributions to overall performance augmentations.

3.5 Validation and Quality Assurance

The validation process has a number of aspects that ensure that the research is credible and that it can be applied into practice. Unit tests are used to produce tests on the functionality of specific units such as data loaders, tokenizers, reward calculators, and feedback generators according to various inputs. Integration testing is the process validation that works end-to-end from intent input through YAML generation to feedback provision. The configurations generated are being tested for deployment in isolated Kubernetes clusters (using Minikube or Kind) to assure their real-world, beyond syntactical correctness, deployability. These human assessment procedures are used to determine the quality of feedback by having the Kubernetes practitioners, who are experienced in evaluating the quality of feedback provided by the system (like the Kubernetes practitioners themselves), sample the quality of the generated improvement suggestions based on their actionability, accuracy, and relevance. This multi-dimensional validation methodology deals with the technical accuracy of the implementation as well as the usefulness of the final product.

4 Design Specification (overview of contribution or new algorithm)

Automated the deployment of code pipeline using Git operations. Included a better Blue metric to get better sense of code quality. Included feedback to users and included support for Python.

Second half of your manuscript! Here, include a concrete description of your developed artefact. This is a relatively short section. Be precise in your words.

- If a new algorithm or model is proposed, a description or pseudocode of the algorithm/model functionality should be included here. Show a drawing of the new: neural network[4], classification strategy, data flow, cloud pattern, etc.
- If you only changed a bit of an existing work, a description of the change should appear here. What is the main difference from the underlying baseline (architecture and/or framework)?
- When you outline the results and scope of your new algorithm/model, mention the associated constraints (tested under X conditions).

No implementation or actual code is required here. Screenshots could be included in the Configuration Manual. The high-level implementation details will appear below in Section 5.

5 Implementation

5.1 High-level implementation details

You will now discuss details of what was described in Section 4. Keep screenshots and other implementation details to a minimum (better place them in the Configuration

[4] `https://keras.io/api/utils/model_plotting_utils/`

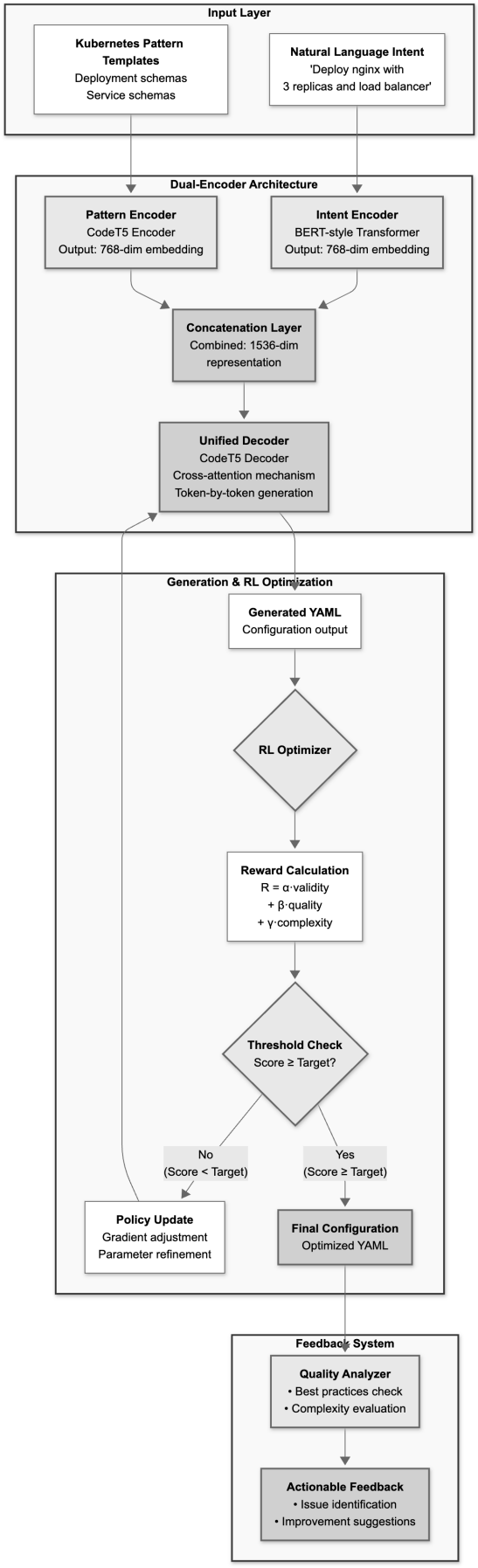

Figure 4: Enhanced Dual-Encoder Architecture with RL Optimization

Manual).

Hints

- ✘ A single long piece of messy code will only make your life harder.
- ✓ Create a README file to show what the code does (or how to compile the project). If you create many files, add a numerical prefix (step01, step02, etc) to the filename to understand the sequence in which you have to run the project.
- ✓ Include a footnote in your GitHub repository.

Be clear on what you implemented, e.g. feature engineering, data augmentation, code and models developed, questionnaires administered, etc. Installation instructions, a screenshot of the code developed, and step-by-step code output will go to the Configuration Manual.

Hints

- ✘ Do *not* copy/paste code or screenshots from your program output here. Gather relevant results on tables (screenshots look ugly!).
- ✓ Include a diagram with the logical flow of your implementation (zoom-in on the overall diagram you created in Section 3).

5.2 Software Bill of Materials

Do not spend too much time talking about scikit-learn, PyTorch, Keras, AWS, etc. It is enough to mention the key libraries, platforms and (when relevant) licencing[5] required. It is important to mention the provenance of any piece of software you used. The baseline code (and associated key paper) should be acknowledged here. Modifications done to the code should also detailed here.

- ✘ Do *not* copy / paste code blindly.
- ✓ Get familiar with the licencing options, including permissive ones, such as Apache 2.0

6 Evaluation (use cases & results)

Here is where you show what you *did* and what you *got*, and then what all these *mean*. More details on Moodle[6]. The first section is about the experiments; then comes the comprehensive analysis of the results and main findings, as well as the implications of these findings.

- In subsections **Experiment / Case Study** you basically write a justification of the experiment, what do you plan to evaluate or look at, and then write what you got.

[5] `https://snyk.io/articles/apache-license/`

[6] `https://moodle2023.ncirl.ie/mod/resource/view.php?id=22557`

- while in **Discussion** you take each **Experiment / Case Study** and interpret the results, one by one.

Hints

✘ Too small font sizes make the plots difficult to read and interpret.

✓ Only include material aligned with the answer to your research question.

6.1 Use Case 1: valid code

did X_1, got Y_1 ...

6.2 Use Case 2: code with syntax errors

did X_2, got Y_2 ...

6.3 Use Case 3: random code

did X_3, got Y_3 ...

6.4 Discussion (interpretation or what those results *mean*)

We finally arrive at the **core** part of your results. You should essentially provide an interpretation and explain why you got those results. A figure or table, such as the one shown in Table 3, summarising the experiments / case studies would definitively help you.

Table 3: Summary of findings (with focus on what you want to contrast)

Experiment	**Main findings and discussions**
X_1	got Y_1; Why did you get that? Is it expected? What does it mean?
X_2	got Y_2; Why did you get that? Is it expected? What does it mean?
X_3	got Y_3; Why did you get that? Is it expected? What does it mean?
X_n	got Y_n; Why did you get that? Is it expected? What does it mean?

Include a detailed discussion of the findings from each **Experiment / Case study**, one-by-one. At least one paragraph for each **Experiment / Case study**. Hints: What does the custom loss function mean? Does it converge at all? What does that histogram mean? What does the confusion matrix tell us? Do not forget to explain *each* figure and table. Think in the "so-what" question and put yourself in the boots of a hypothetical reader, someone outside the College in the years to come.

Hints

✘ do *not* cherry-pick results that confirms your existing beliefs or ideas (avoid confirmation bias)

✓ be honest, it is fine to say something *did not work*; negative results are valuable *results* too.

✓ take the summary from Table 3 and provide a rigorous, in-depth analysis of the results (one by one).

Stay focused and use the results to create a "story"; only the most relevant results related to your research question and the objectives are presented. Statistical tools should be used to critically evaluate and assess the experimental research outputs and levels of significance (mean, median, confidence intervals, p-values, etc). Use visual aids such as graphs, charts, graphs, etc. to make a convincing case for your results.

Hints

✘ Do not be shy. Write! Here, you demonstrate your understanding of the problem at hand.

✓ Make it *abundantly clear* that you know what the results mean **and** provide answers to "so-what?" questions[7].

6.5 Limitations (self-reflection on reproducibility)

As you mentioned in your review of the literature, no work is perfect. Put your work on the spot. Discuss the **limitations** of your own work. You should *criticise* the experiments and be honest about whether your design was fit for purpose. You should always place your findings in the context of previous research that you found during your literature review. The readers want to know the reproducibility of your results. Typically, you can talk about data limitations (sample size, statistical power), limitations of techniques (linear vs. nonlinear), class imbalance bias, expected bias in the train/test sets, shortcuts taken, results generated from only a few runs or epochs, limited feature engineering, missing hyper-parameter optimisation, and so on.

7 Conclusion and Future Work

Finally, a quick summary / recap:

- Restate your research question(s) and objective(s) – from Section 1.2 and Section 1.3.
- In summary, mention your key findings and state how successful you have been in answering the research question(s) quantitatively.
- See the wider picture. Things you can now do, but could not do before.
- The research gap has now been hopefully narrowed; does it open any opportunities elsewhere?
- Write a couple of paragraphs on any recommendations for future work or potential for commercialisation.

[7] Google it or read the following paper `https://doi.org/10.1080/17439884.2013.848454`

Hints

- ✘ Do not introduce new results here. Conclusions should mainly be drawn from Section 6.4.
- ✘ Do not include figures or tables here.
- ✓ What can you do now with your results? What is the main learning?
- ✓ Write about something you wish you had done differently (one or two items from Section 6.5)

Write *meaningful* future work. Having more time or having more CPUs in your work is probably not meaningful. Discuss interesting follow-up research projects. What would you recommend someone to improve, try differently, or expand on?

Note: Final step

It is time to write your **Abstract**. Write it at the end of your work[8]. Since most readers will only go through your **Abstract** and perhaps your **Conclusions**, write these two sections in a self-contained way! In other words, the **Abstract** should invite someone to read your **Conclusions** ... and ... perhaps ... full manuscript! So, think like your readers. *Why* should they read your work?

- ✓ Make abundantly clear *why* your work is important? *Why* does it matter? *Why* should we care about it? *What* is the relevance of this work? *Who* could potentially use your findings?
- ✓ mention the best known result for this problem, and what your best result is
- ✓ include key sentences: We found / discovered / analysed / confirmed / proposed a novel X / etc.
- ✘ Avoid acronyms and things that obscure your work (just push readers away!)

References

Ahmad, W. U., Chakraborty, S., Ray, B. and Chang, K.-W. (2021). Unified pre-training for program understanding and generation, *arXiv preprint arXiv:2103.06333* .
URL: *https://doi.org/10.48550/arXiv.2103.06333*

Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A., Chen, A., Goldie, A., Mirhoseini, A., McKinnon, C. et al. (2022). Constitutional ai: Harmlessness from ai feedback, *arXiv preprint arXiv:2212.08073* .
URL: *https://doi.org/10.48550/arXiv.2212.08073*

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G. et al. (2021). Evaluating large language models trained on code, *arXiv preprint arXiv:2107.03374* .
URL: *https://doi.org/10.48550/arXiv.2107.03374*

[8] `https://www.nature.com/documents/nature-summary-paragraph.pdf`

Dubey, A., Singh, C. P. and Nadig, D. (2024). Leveraging large language models for intent-based generation of cloud-native configurations, *2024 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*, IEEE, pp. 1–6.
URL: *10.1109/ANTS63515.2024.10898244*

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D. et al. (2020). Codebert: A pre-trained model for programming and natural languages, *arXiv preprint arXiv:2002.08155* .
URL: *https://doi.org/10.48550/arXiv.2002.08155*

George, R., Thuseethan, S., Ragel, R. G., Mahendrakumaran, K., Nimishan, S., Wimalasooriya, C. and Alazab, M. (2025). Past, present and future of deep plant leaf disease recognition: A survey, *Computers and Electronics in Agriculture* **234**: 110128.
URL: *https://doi.org/10.1016/j.compag.2025.110128*

Ghorab, M. A. and Saied, M. A. (2025). Towards secure cloud-native computing: Unveiling kubernetes misconfigurations with large language models, *2025 IEEE 18th International Conference on Cloud Computing (CLOUD)*, IEEE, pp. 86–96.
URL: *10.1109/CLOUD67622.2025.00019*

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S. et al. (2020). Graphcodebert: Pre-training code representations with data flow, *arXiv preprint arXiv:2009.08366* .
URL: *https://doi.org/10.48550/arXiv.2009.08366*

Kumar, S., Pv, G. and Jackson, B. (2023). Machine learning-based timeseries analysis for cryptocurrency price prediction: A systematic review and research, *2023 International Conference on Networking and Communications (ICNWC)*, pp. 1–5.
URL: *https://doi.org/10.1109/ICNWC57852.2023.10127439*

Le, H., Wang, Y., Gotmare, A. D., Savarese, S. and Hoi, S. C. H. (2022). Coderl: Mastering code generation through pretrained models and deep reinforcement learning, *Advances in Neural Information Processing Systems* **35**: 21314–21328.
URL: *https://doi.org/10.48550/arXiv.2207.01780*

Liu, J., Zhu, Y., Xiao, K., Fu, Q., Han, X., Yang, W. and Ye, D. (2023). Rltf: Reinforcement learning from unit test feedback, *arXiv preprint arXiv:2307.04349* .
URL: *https://doi.org/10.48550/arXiv.2307.04349*

Malul, E., Meidan, Y., Mimran, D., Elovici, Y. and Shabtai, A. (2024). Genkubesec: Llm-based kubernetes misconfiguration detection, localization, reasoning, and remediation, *arXiv preprint arXiv:2405.19954* .
URL: *https://doi.org/10.48550/arXiv.2405.19954*

Nguyen, T. T., Nguyen, N. D. and Nahavandi, S. (2020). Deep reinforcement learning for multiagent systems: A review of challenges, solutions, and applications, *IEEE Transactions on Cybernetics* **50**(9): 3826–3839.
URL: *https://doi.org/10.1109/TCYB.2020.2977374*

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A. et al. (2022). Training language models to follow instructions with human feedback, *Advances in neural information processing systems* **35**: 27730–27744.
URL: *https://doi.org/10.48550/arXiv.2203.02155*

Pujar, S., Buratti, L., Guo, X., Dupuis, N., Lewis, B., Suneja, S., Sood, A., Nalawade, G., Jones, M., Morari, A. et al. (2023). Automated code generation for information technology tasks in yaml through large language models, *2023 60th ACM/IEEE Design Automation Conference (DAC)*, IEEE, pp. 1–4.
URL: *10.1109/DAC56929.2023.10247987*

Raiaan, M. A. K., Mukta, M. S. H., Fatema, K., Fahad, N. M., Sakib, S., Mim, M. M. J., Ahmad, J., Ali, M. E. and Azam, S. (2024). A review on large language models: Architectures, applications, taxonomies, open issues and challenges, *IEEE access* **12**: 26839–26874.
URL: *10.1109/ACCESS.2024.3365742*

Ren, S., Guo, D., Lu, S., Zhou, L., Liu, S., Tang, D., Sundaresan, N., Zhou, M., Blanco, A. and Ma, S. (2020). Codebleu: a method for automatic evaluation of code synthesis, *arXiv preprint arXiv:2009.10297* .
URL: *https://doi.org/10.48550/arXiv.2009.10297*

Repina, O., Carvalho, R. C., Coco, G., Antolínez, J. A., de Santiago, I., Harley, M. D., Jaramillo, C., Splinter, K. D., Vitousek, S. and Woodroffe, C. D. (2025). Evaluating five shoreline change models against 40 years of field survey data at an embayed sandy beach, *Coastal Engineering* **199**: 104738.
URL: *https://doi.org/10.1016/j.coastaleng.2025.104738*

Srivatsa, K. G., Mukhopadhyay, S., Katrapati, G. and Shrivastava, M. (2024). A survey of using large language models for generating infrastructure as code, *arXiv preprint arXiv:2404.00227* .
URL: *https://doi.org/10.48550/arXiv.2404.00227*

Wang, Y., Le, H., Gotmare, A. D., Bui, N. D., Li, J. and Hoi, S. C. (2023). Codet5+: Open code large language models for code understanding and generation, *arXiv preprint arXiv:2305.07922* .
URL: *https://doi.org/10.48550/arXiv.2305.07922*

Wang, Y., Wang, W., Joty, S. and Hoi, S. C. (2021). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, *arXiv preprint arXiv:2109.00859* .
URL: *https://doi.org/10.48550/arXiv.2109.00859*

Yetiştiren, B., Özsoy, I., Ayerdem, M. and Tüzün, E. (2023). Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt, *arXiv preprint arXiv:2304.10778* .
URL: *https://doi.org/10.48550/arXiv.2304.10778*