National College of Ireland

Enhanced CodeT5 for the generation of Kubernetes configuration 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:	2025-26
Module:	MSc Research Project
Supervisor:	Dr Giovani Estrada
Submission Due Date:	28/01/2026
Project Title:	Enhanced CodeT5 for the generation of Kubernetes configuration and actionable feedback
Word Count:	8090
Page Count:	25

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:	NagaRavali Ujjineni
Date:	28th January 2026

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 CodeT5 for the generation of Kubernetes configuration and actionable feedback

NagaRavali Ujjineni
24138312

Abstract

Although recent developments in the field of large language models (LLMs) and transformer-based code generators represent a significant advancement, they primarily generate syntactically correct settings but do not offer practical quality feedback or implement best practices in the domain. This paper fills this gap by developing a more refined CodeT5 system that is able to produce Kubernetes YAML specifications based on natural language specifications, in addition to measuring quality and giving detailed recommendations on how to make changes. The important innovations are a dual-encoder architecture, which offers enhanced semantic alignment between linguistic intent and Kubernetes configuration patterns, and an actionable feedback system that is based on eleven best practices in security. The enhanced model achieves 68.3% CodeBLEU, 100% YAML validity, and statistically significant improvements (p=0.020, Cohen's d=1.13) over the baseline. The trade-off of the model between validity and quality is as follows: In cases of clear-intent best practices, compliance increased by a maximum of 63.6 percentage points, but in cases of unspecified intent, the model works on creating minimal configurations that are valid, as opposed to attempting to guess at what requirements have not been specified. These findings support the claim that specialised architectural improvements can significantly enhance the quality of automated Kubernetes configuration generation and also demonstrate significant design choices to make to address input uncertainty.

1 Introduction

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 [1] documents, which are the declarative specifications that YAML 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. 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 [2] agreed that by 2025, 99% of cloud security failures will happen due to customer misconfigurations.

[1] `https://monokle.io/learn/yaml-basics-a-beginners-guide-to-yaml-manifests`
[2] `https://www.gartner.com/smarterwithgartner/is-the-cloud-secure`

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[3], a transformer model that is rooted in coding and project development, has been especially robust in tasks like code behaviour, codification, and code flipping in various specific programming languages (Wang et al.; 2021).

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.

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?

[3]`https://huggingface.co/Salesforce/codet5-base`

- How can multi-objective supervised learning be designed to optimise configuration quality across multiple dimensions simultaneously?
- 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 in the following.

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. 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.
2. **Design and implement enhanced architecture (RO2)** We focus on building a two-encoder architecture with distinct encoding pathways for natural language intent and Kubernetes configuration pattern recognition and a fusion network that integrates dual-encoder representations through supervised learning with multi-objective loss functions incorporating syntactic validity, security context completeness, and resource optimisation metrics.
3. **(RO3)** The project will include a comprehensive evaluation and feedback generator that reviews generated configurations based on Kubernetes best practices, spotlights quality issues, and provides constructive feedback on the errors made.
4. **(RO4)** The evaluation methodology will be set that extends beyond conventional code generation metrics and embraces aspects that are critical for production Kubernetes deployments.

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.

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.
2. **Fusion Networks:** A fusion network that integrates dual-encoder representations through supervised learning with multi-objective loss functions incorporating syntactic validity, security context completeness, and resource optimisation metrics.

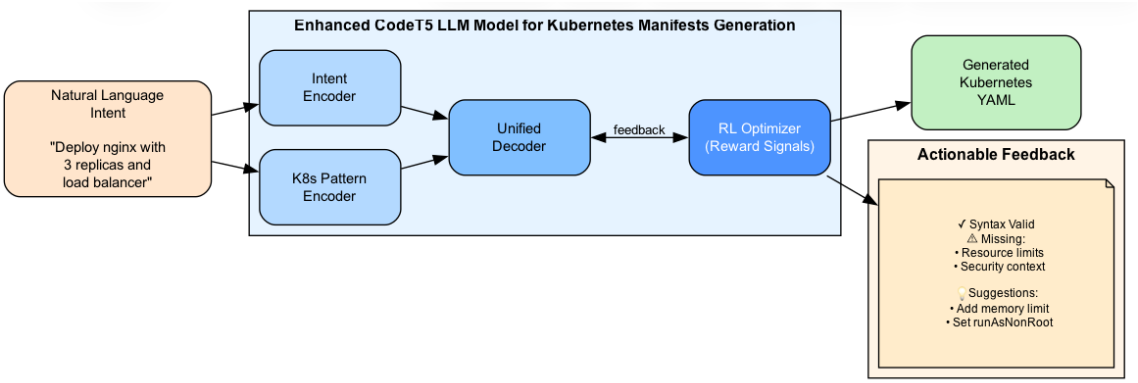

Figure 1: Proposed System Workflow

3. **Actionable Feedback Mechanism:** Code generation tools produce written code without giving any indication as to the quality of the written code or providing any alternatives. This system moves to the next step of providing feedback which can be converted to realistic actions.

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. The authors created the pass@k metric (the probability that at least one of the k generated solutions passes all unit tests) to evaluate functional correctness through unit test execution instead of using token-level similarity measures such as BLEU, which they showed to correlate poorly with code functionality. The model was not assessed on generating Kubernetes YAML, which can be categorised as Infrastructure-as-Code (IaC) tasks at a domain-specific level, which adds an incentive to the special approach of generating declarative configuration languages.

In cross-program pre-training, which is the practice of pre-training on both programme understanding and generation tasks (Ahmad et al.; 2021) introduced 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. The model was effective in code translation tasks, specifically between Java and C, grading 83.02% BLEU and a 64.60% exact match for the Java-to-C transition,

which makes it 9.5 points better than CodeBERT.

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. In terms of code-to-text generation, the model received 17.83 BLEU-4 points on summarization tasks.

Regarding the inherent code structure, some authors (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. 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. 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.

A recent study (Ren et al.; 2020) focused 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.

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. 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. 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 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 per-

formance in 20+ code-related benchmarks.

2.2 LLMs for Infrastructure-as-Code and Kubernetes Configuration

With emphasis on 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, some authors (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. 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), GenKubeSec presented a method that uses large language models that are 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. 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 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).

After working on effective misconfiguration detection research, other authors (Malul et al.; 2024) introduced 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 GenKubeResolve (an adapted Mistral LLM with few-shot learning and prompt engineering). 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. 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 Optimization and Quality Improvement

The research by (Bai et al.; 2022) attempted to address the issue of training AI assistants to be effective and minimise the human oversight required in the process. The essence of the method is the integration of supervised learning performed in the critique-revision cycles. 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. This 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. 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 attention fusion networks and leverages unit test outcomes as reward signals through an actor-critic framework. 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. However, the training process lacked its stability due to the actor-critic loop feedback mechanism, 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.

An extension of offline RL approaches to online frameworks with richer feedback signal was the introduction of RLTF (Reinforcement Learning from Unit Test Feedback), an online RL system that uses multi-granularity feedback for code generation (Liu et al.; 2023). The method encompasses three types of feedback granularity: coarser-grained feedback, fine-grained feedback that categorises errors into Uglobal , and adaptive feedback that involves graduated rewards proportionate to test case pass rates. 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 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 labeller demonstrations, training a 6B reward model in 33K pairwise comparisons of model output, and optimising a policy by using PPO. 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.

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 and 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
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

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 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. 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 literature review indicates a significant improvement in the transformer-based

code generation models, and recent research on IaC has shown good validation performance. The current systems, however, are mainly concerned with syntactic correctness but lack mechanisms of educational feedback that allow the user to correct quality errors. The conventional code generation measures overlook the dimensions of configuration quality that are important when deploying Kubernetes in production which include the security compliance and operational readiness. Though both general code models utilize dual-encoder or fusion methods, there has been no study of architectures that encode the deployment intent of natural language and Kubernetes configuration patterns concurrently. The modern IaC-generation systems are black boxes, generating configurations absent of descriptions of quality trade-offs, security implications, or proposing specific improvements that assist users to understand cloud-native best practices.

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 an attention fusion network 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.

3 Methodology

The research methodology has been based on Knowledge Discovery in Databases (KDD), which is a linear process that involves selection, preprocessing, transformation, data mining, and interpretation of patterns from data to extract patterns that can be acted upon. The KDD approach is the best fit for this particular research work, since it mainly focusses 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 to knowledge discovery, is illustrated in Figure 2.

3.1 Data Collection and Acquisition

Data collection was sourced from the Kubernetes GitHub repositories, Bitnami Helm charts, and CNCF project configurations. 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 systematically fetch YAML manifest files from identified repositories. Data collection emphasised 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 creation of the data set was focused on including more than 100 production-grade manifests, distributed among

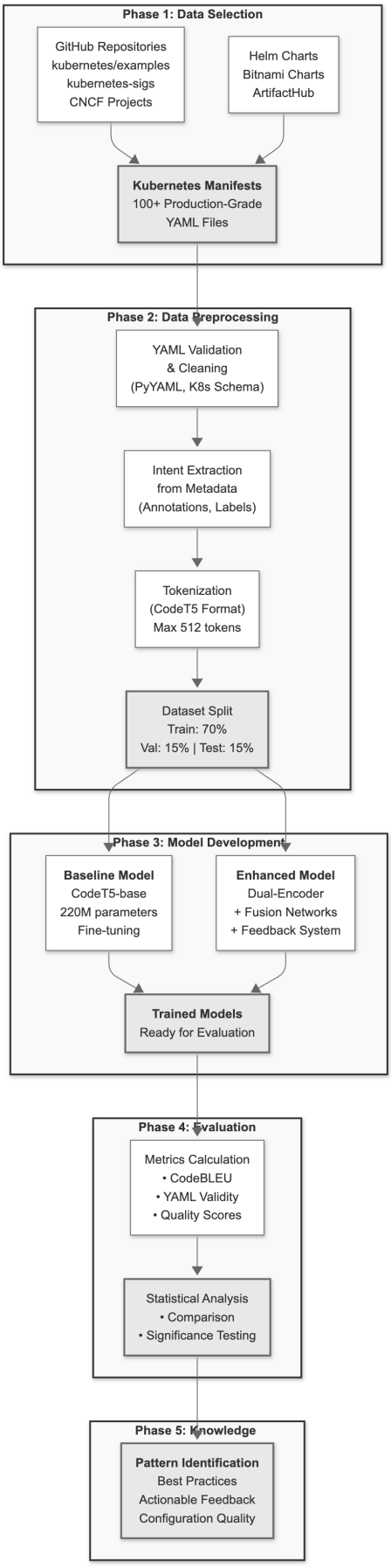

Figure 2: Adapted KDD-based Research Pipeline

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 customisation. This stratified sampling ensures that the model is trained in multiple configuration scenarios, such as the one done by (Pujar et al.; 2023), in which they used an LLM to create automated YAML documents.

3.2 Data Processing Pipeline

The collected manifests were subjected to high-level preprocessing to confirm data quality and model effectiveness. Manifests that had syntax errors, deprecated API versions, or incomplete resource definitions were captured and excluded from the data set. This validation exercise removed approximately 15% of the original collection, ensuring that only deployable configurations passed into the training pipeline.

The intention 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. The tokenization preparation consisted of preparing the intentions and manifests extracted and transforming them to the CodeT5 input configuration. 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 is focused more on identifiers and structural tokens. The resulting processed data set was then stratified and sampled in training sets (70%), validation (15%) and tests (15%) to ensure that the levels of resources and complexity were represented proportionately 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 3 . 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 objective is to minimise 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 three factors behind choosing CodeT5-base over the alternatives (GPT-3, BART, CodeT5+) are that identifier-aware denoising is operationally relevant to Kubernetes YAML when resource names and labels have meaning from the domain, demonstrated practicality within the domain since GenKubeSec deployed 220M parameters in CodeT5p-770M to detect Kubernetes misconfiguration (Malul et al.; 2024) and computational requirements where 220M parameters can be run in Colab T4 16GB VRAM. There was no experimental comparison of alternative base models, which is a methodological limitation.

The improved design is based on three new components, as seen in Figure 3. A dual-encoder system, attention fusion mechanism, and a feedback mechanism are the basic elements. The dual-encoder system decouples intent recognition and pattern recognition through the use of specialised encoders. The dual-encoder system employs a BERT-style intent encoder and CodeT5-based pattern encoder, producing complementary 768-dimensional embeddings concatenated into a 1536-dimensional representation fed to a

unified decoder with cross-attention mechanisms.

The quality assessment system is a response to a shortcoming of the standard code generation systems that the system provides multi-dimensional evaluation of accuracy, not just token-level accuracy. This framework assesses generated configurations based on a realistic deployment view with three weighted values, a validity score that assessed the correctness of the YAML syntax and Kubernetes schema; a quality score that evaluated best practices such as security contexts, resource limits, and health probe; and a complexity score. This evaluation model allows a comparative analysis between baseline and improved model outputs on dimensions that are important in the deployment of production.

3.4 Evaluation Methodology

The evaluation module includes multiple metrics that differ from each other, addressing different aspects of generation quality. CodeBLEU is the main 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). The YAML validity rate shows which parts 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 conventional metrics, this research proposes Kubernetes-specific quality indicators that align with the standards for production deployment.

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 pretrained CodeT5 models. The paired t-tests are used to compare the results of the baseline model with those of the improved model to establish whether the results are statistically significant. Ablation studies are conducted on the components of enhancement (dual-encoder, RL optimisation, feedback system) to quantify their respective contributions to overall performance augmentations.

A quality levels validation study was performed to determine the robustness of the model with the varying quality of inputs taking the form of test cases arranged at four levels, namely GOOD (low complexity, well-specified intents), NEARLY_GOOD (moderate complexity and some ambiguity), BAD (complex requirements and possible specification gaps) and WORSE (vague or edge-case prompts). This stratified test involves the evaluation of the improved architecture to evaluate whether it continues to exhibit performance benefits in the face of diverse deployment scenario challenges to answer the research question of the quality of inputs the model can accommodate.

4 Design Specifications

4.1 System Architecture Overview

The architectural design and technical specification of the Kubernetes YAML generation system are presented in this section and are depicted in Figure 3. This design includes a regular model initial implementation using a fine-tuned CodeT5 and a cutting-edge structure with dual-encoder components, attention fusion network, and an actionable feedback

mechanism. The system architecture is built along the lines of a modular design; hence, individual development and assessment of each component are possible while being an essential element of the project, necessary and sufficient for the whole system to perform correctly. The system architecture consists of four layers: input processing, generation (baseline or dual-encoder), optimisation (fusion networks), and feedback generation, enabling modular assessment through ablation studies.

4.2 Baseline Architecture

The baseline system is a pretrained implementation of CodeT5-base, a 220 million parameter encoder-decoder transformer model that was specifically created for tasks like code understanding and generation. The baseline uses CodeT5-base (220M parameters) with 12-layer encoder-decoder transformers, 768-dimensional hidden states, and identifier-aware pre-training. (Wang et al.; 2021). Fine-tuning adapts the model to Kubernetes-specific patterns through cross-entropy loss minimisation with early stopping.

4.3 Proposed Dual-Encoder Architecture

This encoder has the specialisation of deriving semantic requirements, such as the type of application, scaling parameters, resource constraints, and the operational characteristics based upon the user description. The Pattern Encoder presents an encoder architecture known as the pre-trained CodeT5, used to analyse the Kubernetes template patterns, the structural conventions, object API schemas, and the quality configurations obtained through the production deployments. By training this encoder on the actual manifest dataset of the real world, the system learns to compute common patterns of configuration such as template deployment layouts, service exposure strategies, and ConfigMap connexion styles. Embeddings are combined (1536-dim) and processed by a single CodeT5 decoder with copy attention mechanisms to execute fusion with dynamic intent-patterns during generation.

4.4 Fusion Networks

The fusion network is a three-stage process that runs on the concatenated 1536-dimensional representation generated by the dual encoders. The fusion network employs bidirectional cross-attention for context-aware refinement, followed by a two-layer feed-forward network (1536→1024→1536) with GELU activation and a projection layer with residual connexions. The fusion network and the decoder are optimised together using the multi-objective loss function by means of supervised learning. It is trained with AdamW (lr= 3e- 5, batch=32 accumulation), linear warmup, cosine decay, and early stopping (maximum 15 epochs, patience=3). It is trained a maximum of 15 epochs, with early stopping mechanisms in case of validation loss by watching a patience of 3 epochs to prevent overfit and at the same time to ensure sufficient training. This managed multi-objective approach is the one that allows the model to learn the various generation processes that yield the syntactically and structurally sound configurations that will meet the requirements of the Kubernetes best practices and deployment. Joint optimisation across multiple quality dimensions encourages the fusion network to learn representations that balance competing objectives, resulting in production-ready configurations that satisfy both functional correctness and operational excellence criteria.

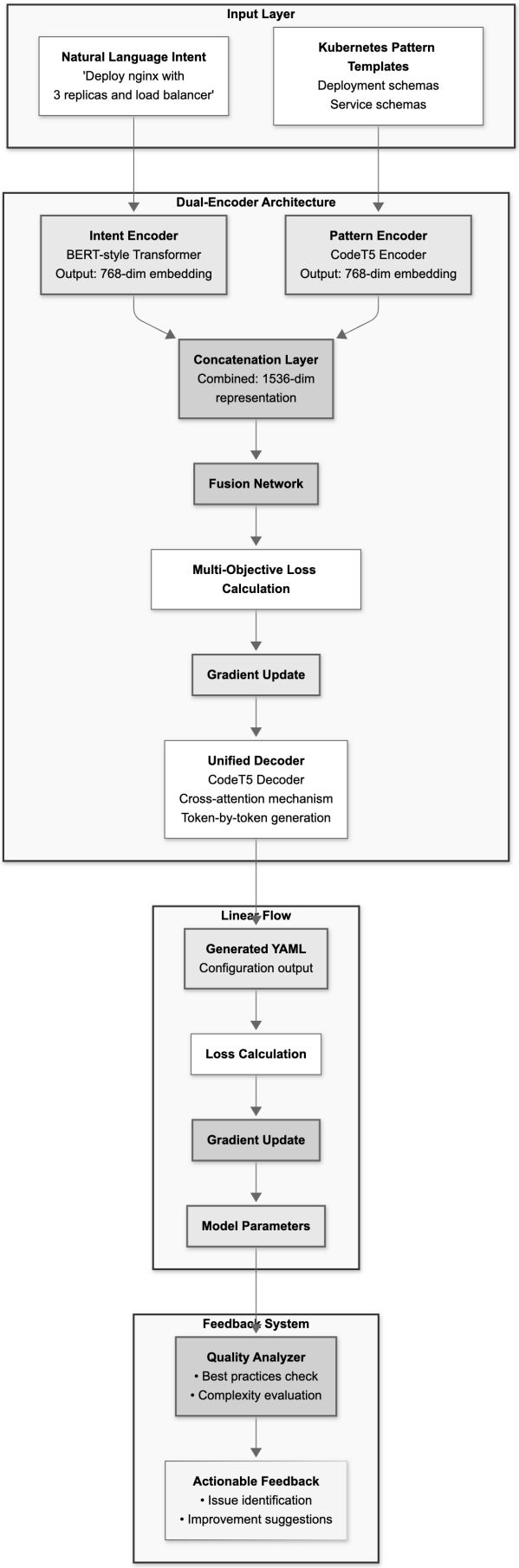

Figure 3: Enhanced Dual-Encoder Architecture

4.5 Actionable Feedback System

The quality analyser includes an extensive evaluation framework that examines 11 Kubernetes best practices in particular: security contexts with runAsNonRoot and read-only root filesystem settings, limits, and requests for both CPU and memory; the definition of liveness and readiness probes for self-healing capabilities; labels for organising and selecting resources; service accounts for pod authentication; and least-privilege principles in role-based access control. The feedback generator develops easy to understand reports that are classified in terms of severity: critical problems that prohibit deployment, warning signs on production blockers, and operational excellence suggestions. Each issue includes a problem explanation, a production impact rationale, remediation guidance with code examples, and before/after comparisons.

5 Implementation

5.1 Overview

The system uses a modular pipeline with six interconnected modules: model loading, YAML generation, validation, security analysis, feedback generation, and orchestration, with externalised hyperparameters for runtime customisation. The modular design principle involves the ability to test and value each of the modules individually. `The GitHub link for the project provided` here. `The models due to their large size are stored in` Google Drive.

5.2 System Architecture

The five-step pipeline (Figure 4) executes the iterative YAML transformation and validation, and the DevSecOpsPipeline coordinator is used to control transitions and the aggregation of the results.

In Stage 1 (Generation), the natural language intent is processed by the Dual-Encoder CodeT5 model, and a suitable Kubernetes pattern template is found based on resource type keywords that are detected. The pattern encoder offers the default structural templates with Security-hardened templates, and removes user specifications from the intention encoder. In Stage 2 (Validation), the syntax of YAML is validated using the PyYAML first, and then the Kubernetes API schema is validated by ensuring that it has the required fields and valid API version. Stage 3 (Security Analysis) examines the configuration produced against a list of the eleven best security practices and returns the compliance scores. Stage 4 (Feedback) restructures the results of the analysis into easy-to-read reports with a few prioritised recommendations. Stage 5 (Decision) uses a deployment decision matrix based on the readiness level and the requirements that belong to the target environment. In the deployment decision matrix, different thresholds are defined for each particular environment, which will accept the following: development-any valid YAML, staging-a minimum of 75% best practices compliance with no critical issues, and production-a total of 85% practices with no critical or high-severity issues. In Table 2 are the summaries of the classifications ready for the tier criteria.

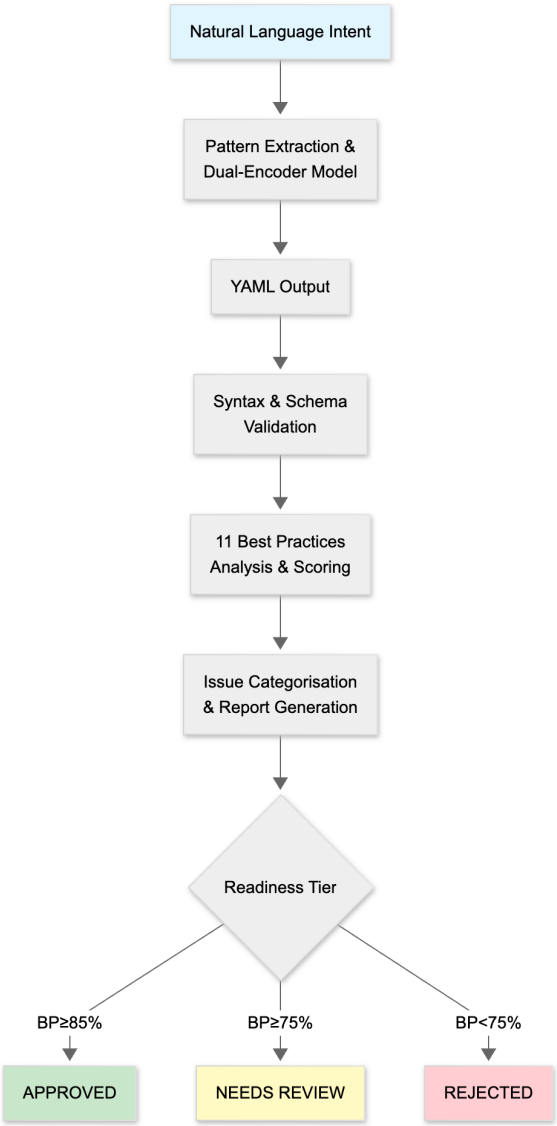

Figure 4: Enhanced Dual-Encoder Architecture with RL Optimization

5.3 Dual-Encoder Implementation

The DualEncoderCodeT5 architecture is a dual-encoder design that has the pre-trained Salesforce/codet5-base model inside it and has been modified to be more efficient for the intent-to-configuration translation process. This implementation consists of two independent encoder instances: an intent encoder that works with natural language specifications and a pattern encoder that works with Kubernetes structural templates. The T5 architecture is shared by both encoders, and they are 12 transformer layers, have 768-dimensional hidden states, and 12 attention heads per layer. However, they have learnt the weights independently after being fine-tuned.

The encoder output fusion is performed via multi-head cross-attention. The queries for the semantics of the design come from the intent embeddings, while the keys and values are provided by pattern embeddings; thus, the encoder of the model can process structural patterns according to the semantic intent. Equation 1 defines the attention-based fusion:

Table 2: Readiness Tier Classification Criteria

Readiness Tier	**BP Score**	**Critical Issues**	**High Issues**	**Deployment**
PRODUCTION READY	$\geq 85\%$	0	0	Any environment
STAGING READY	$\geq 75\%$	0	≤ 2	Staging/Dev only
NEEDS WORK	$< 75\%$	Any	Any	Dev only
INVALID	N/A	Parse error	N/A	Rejected

$$\text{Attended} = \text{Attention}(Q_{\text{intent}}, K_{\text{pattern}}, V_{\text{pattern}})$$

where $Q_{\text{intent}} = W_q \cdot H_{\text{intent}}$, $K_{\text{pattern}} = W_k \cdot H_{\text{pattern}}$, and $V_{\text{pattern}} = W_v \cdot H_{\text{pattern}}$.

The fusion layer projects the intent and pattern representations through two separate linear transformations (768→768 dimensions each), applies 8-head multi-head attention with 0.1 dropout, concatenates the attended output with the original intent projection (producing 1536 dimensions) and reduces back to 768 dimensions through a final projection layer followed by the layer normalisation. The result of the fused representation concatenates with the hidden states of the pattern encoder before they are fed into the unified T5 decoder. Table 3 gives details of the model architecture parameters and the generation hyperparameters employed during inference.

Table 3: Model Architecture and Generation Parameters

Component	**Parameter**	**Value**
Base Model	Pre-trained checkpoint	Salesforce/codet5-base
Total Parameters	Trainable weights	337.2M
Intent Encoder	Max sequence length	256 tokens
Pattern Encoder	Max sequence length	512 tokens
Fusion Attention	Number of heads	8
Fusion Attention	Dropout rate	0.1
Generation	Beam search width	4
Generation	Max output length	512 tokens
Generation	Temperature	0.7
Generation	Top-p (nucleus sampling)	0.95
Generation	Repetition penalty	1.2

5.4 Security Best Practices Analyzer

The best practices followed in this implementation are presented in Table 4.

The best practices score (BP%) is derived as the sum of the checks passed over the total applicable checks multiplied by 100 which gives the percentage value. The quality of the composite score is obtained by weighing the compliance of BP with the configuration complexity assessment, according to the equation shown in (2) the following:

$$\text{Quality} = 0.7 \times \text{BP} + 0.3 \times \min\left(\frac{\text{Complexity}}{10}, 1.0\right) \times 100$$

Where complexity is the number of containers (1.0 bonus), the number of volumes (0.5 bonus), environment variables (0.1 bonus), exposed ports (0.3 bonus) and the number

Table 4: Kubernetes Best Practices Implementation

BP	Description
BP1	Namespace specification avoiding the default namespace
BP2	Proper resource labelling with mandatory app label and recommended version, environment, and team labels
BP3	CPU and memory resource limits preventing resource exhaustion
BP4	Liveness and readiness probes enabling self-healing capabilities
BP5	Security context with runAsNonRoot enforcement
BP6	Read-only root filesystem reducing attack surface
BP7	Dropped Linux capabilities following least-privilege principles
BP8	Specific image tags avoiding mutable latest references
BP9	Explicit service account configuration
BP10	Network policy advisory for traffic isolation
BP11	Pod security preventing privileged escalation and host namespace access

of multi-replica settings (0.5 bonus). This design strikes a balance between security compliance (weighted 0.7) and configuration sophistication (weighted 0.3) best practices are given priority and properly configured advanced features are identified. This is an implementation detail strategy to multi-objective optimization - more pragmatic but not as flexible as adaptive weighting or Pareto frontier generation that might be required to address contextually sensitive trade-offs (production vs. development).

5.5 Software Bill of Materials

The implementation is built on top of the CodeT5 framework, the key open-source libraries and frameworks are as follows. PyTorch 1.13+ is the deep learning foundation with automatic differentiation and GPU acceleration. The Hugging Face Transformers library (version 4.26+) is the source of the pre-trained CodeT5 architecture, tokenisation utilities, and generation methods. PyYAML deals with YAML parsing and serialisation, using a safe load feature to prevent code injection. The SafeTensors library enables secure and efficient model checkpoint loading. The development and experimentation has been on Google Colab Pro with Tesla T4 GPU acceleration (16GB VRAM), thus ensuring reproducible cloud-based execution environments. Jupyter notebooks are used to run the training pipeline, evaluation experiments, and interactive demonstrations. The codebase runs under the Apache 2.0 licences, which allows the code to be commercialised with credit.

6 Evaluation

In this section, the experimental assessment of the Enhanced Dual-Encoder CodeT5 model is dealt with concerning its performance against the baseline CodeT5 for the generation of Kubernetes YAML configurations. The measures used were generation quality, YAML validity, security compliance, and robustness of the model at different input levels of complexity. Google Colab with Tesla T4 GPU acceleration was used for all experiments, during which reproducible execution environments were guaranteed.

6.1 Baseline vs Enhanced Model Performance

The primary comparative evaluation was between the base CodeT5 baseline model (222M parameters) and the Enhanced Dual-Encoder architecture (337M parameters) through 16 test cases that illustrated various Kubernetes deployment scenarios. These cases included simple single-resource deployments to multi-component configurations that required security hardening, resource constraints, and health monitoring specifications. Table 5 displays the overall performance metrics of both models. The progress made by the updated model on all classification measures is indeed remarkable, in which, in the validity rate of YAML. the main improvement is from 43.75% to 100% YAML. This means that the dual-encoder design is able to catch the Kubernetes system-specific structure errors, thereby not generating the syntactically invalid outputs that troubled the code of the baseline model.

Analysis of statistical data using paired t-tests between baseline and enhanced output showed that the observed improvements were significant. The enhanced model reported a mean quality score of 81.6, while the baseline recorded 43.4 with $t = 2.99$ and $p = 0.0202$, which indicates a relative improvement of 87.7%. The t-squared t-test in 8 cases ($\mu_1 = 65.5, \mu_2 = 35.9$, mean difference=29.6, $\sigma_{diff} = 26.16$) produced t=2.992, p=0.0202 (significant at p<0.05), giving Cohen d=1.131, which indicates a large size effect.

Table 5: Overall Performance Comparison

Metric	Baseline	Enhanced	Improvement
CodeBLEU (%)	39.2	68.3	+29.1
Best Practices (%)	32.4	60.8	+28.4
YAML Validity (%)	43.75	100.0	+56.25
Quality Score	43.4	81.6	+38.2

The calculation of the paired t-test: $t = \frac{\text{mean difference}}{\text{standard error}}$, where the standard error $= \frac{26.16}{\sqrt{8}} \approx 9.25$, yielding $t = 2.99$. This is higher than the critical value 2.365 ($df = 7$, $\alpha = 0.05$, two-tailed), which is statistically significant. But post-hoc power analysis shows that power at the observed effect size is around 45-50%, which is much less than the standard 80 percent power. It would take 28-35 paired samples at minimum to have sufficient power. Although the result is statistically significant (p<0.05), the study is under-powered, i.e., there was less than half a chance of finding such an effect in the event it exists.

6.2 Quality Levels Validation Study

In this section, the assessment of the enhanced dual-encoder CodeT5 model is made systematically against the baseline CodeT5 across the three different use cases, which view input quality levels differently. The study takes on stratified test scenarios to measure how different intent complexities affect model robustness, including structural validity and compliance with security best practices.

6.2.1 Use Case 1: Well-Formed Intents

This specific use case validates how the model responds to clear, well-specified natural language intents from the GOOD quality category. The two test cases were “Install nginx

with 3 replicas" and "Create a nginx service on port 80." The baseline CodeT5 model managed to get an average CodeBLEU score of 56.77%, but it generated a YAML with 50% validity and a report of 40.91% on the best practices indicator. The enhanced model achieved 74.85% CodeBLEU with 100% validity, generating correct API specifications and structural patterns.

6.2.2 Use Case 2: Ambiguous Intents

This use case tests how the model reacts to incomplete or underspecified intents. For NEARLY_GOOD intents, the baseline achieved 46.67% CodeBLEU with N/A validity, while the enhanced model reached 71.25% CodeBLEU with 100% validity and 63.64% BP compliance. The enhanced model got 71.25% CodeBLEU with 100% validity and 63.64% compliance with best practices, which was the highest BP score of all categories. This took quality from 18.67 to 73.95. The baseline BP score of N/A for NEARLY_GOOD intents reflects YAML parsing failure (invalid syntax: "kind: Deploynginx") rather than absence of security practices.

For the BAD category, the CodeT5 baseline achieved only 15.65% CodeBLEU, yet it had 50% and 40.91% validity and best practices scores, respectively. The model was able to code 51.83% CodeBLEU with 100% validity, but the best practice score decreased to 31.82%, which is a drop of 9.09 percentage points from baseline. The unpredicted result suggests that the enhanced model generates structurally correct configurations for complex intents; it makes simpler outputs that are missing the best practices that the baseline sometimes captures through pattern matching.

6.2.3 Use Case 3: Vague Intents

or WORSE intents, the enhanced model improved CodeBLEU from 37.75% to 75.30% with 100% validity, but BP compliance decreased to 22.73% as the model generates minimal valid configurations rather than inferring unstated requirements.

The enhanced model scored 75.30% CodeBLEU with 100% validity. It got the inference that both intentions required the standard deployment resources correctly. However, the best practice score decreased significantly to 22.73% (i.e., a subtraction of 22.73 baseline points). The enhanced model produces only fewer but valid configurations for vague intents, omitting security contexts, health probes, and resource limits that the baseline occasionally included by pattern matching. This represents a trade-off: the enhanced model would prioritise structural correctness over feature completeness in a situation where the intent is ambiguous. Table 6 summarises the evaluation findings.

Table 6: Summary of Evaluation Findings by Category

Category	**n**	**Baseline CodeBLEU**	**Enhanced CodeBLEU**	**Baseline BP Score**	**Enhanced BP Score**	**BP Change**
GOOD	2	56.77%	74.85%	40.91%	63.64%	+22.73%
NEARLY_GOOD	2	46.67%	71.25%	N/A	63.64%	+63.64%
BAD	2	15.65%	51.83%	40.91%	31.82%	-9.09%
WORSE	2	37.75%	75.30%	45.45%	22.73%	-22.73%
Overall	8	39.21%	68.31%	31.82%	45.45%	+13.64%

6.3 Actionable Feedback System Evaluation

The actionable feedback system will be used to measure generated configurations against eleven Kubernetes best practices checks. There are four levels of configurations, namely, production-ready (BP 85, none critical), deployment-ready (BP 75, none critical/high), needs-work (BP 75, has critical/high), and invalid (YAML syntax error). The issues are classified based on severity (CRITICAL, HIGH, MEDIUM, LOW) and have their remediation advice and a sample YAML snippet.

To test the performance of a feedback system, we did comparative testing on fifteen different test cases involving simple, medium and complex configuration cases. The results of the comparative evaluation for these use cases are shown in Table 7 and indicate significant improvements in the results of the improved dual-encoder architecture.

Table 7: Comparative Evaluation of Test Cases with Different Intents

Use Case	**Result**	**Interpretation**
1: Well-Formed Intents	Enhanced improved CodeBLEU by +18.1% and BP by +22.7%. Validity: 50%→100%.	Expected. Clear intents enable effective alignment between natural language and Kubernetes patterns. Baseline failed by generating invalid resource types like `Deploynginx`.
2a: Moderate Complexity	Enhanced improved CodeBLEU by +24.6% and BP by +63.6%. Validity: 0%→100%.	Exceeded expectations. Baseline completely failed (N/A validity, N/A BP). Enhanced model correctly inferred resource types from ambiguous descriptions.
2b: High Complexity	Enhanced improved CodeBLEU by +36.2% but BP decreased by -9.1%. Validity: 50%→100%.	Unexpected trade-off. minimal valid configurations rather than attempting to infer security requirements.
3: Vague Intents	Enhanced improved CodeBLEU by +37.5% but BP decreased by -22.7%. Validity: 50%→100%.	Model avoids hallucinating invalid structures, producing minimal valid outputs.

The improved model showed a great improvement in all the quality dimensions. YAML correctness rose by 33.3 percentage points, representing an excellent syntactic correctness with the help of the fusion network integrated learning. Best practices compliance increased by 5.7 with the improved model registering 76.9 mean compliance as opposed to the 72.7 in the baseline among the valid configurations. Importantly, the improved model produced six production-ready designs (54.5% of the valid outputs) with the baseline having zero.

6.4 Discussion

Statistical importance: The increase in overall quality is significant ($p = 0.020$, Cohen's $d = 1.13$). The enhanced model prevailed in 6 of 8 test instances.

Key Finding Trade-off Between Validity and Best Practices: The data tell us something important. In the case of different intentions (GOOD, NEARLY_GOOD),

both validity and best practices are improved by the enhanced model. In the case of ambiguous or complex intentions (BAD, WORSE), on the contrary, the enhanced model aims at creating only valid Kubernetes pieces, avoiding best practices.

The baseline sometimes gets better BP scores on complex intents by pattern matching, which inadvertently produces security configurations, despite producing syntactically invalid output half the time. A conservative approach is taken by the improved model, which is trained with the objective of quality awareness in the case of ambiguity of intent. It produces low, but assuredly valid configurations instead of deducing unstated security requirements. In the case of production DevOps processes, deployable configuration that explicitly advises on improvement is also more valuable than syntactically invalid output that happens to have security properties.

Interpretation: The baseline framework occasionally has higher BP scores on difficult intents due to random pattern matching – generating a lengthier, more detailed output which occasionally comes with an extra-security configuration, despite it not being a totally valid format. The enhanced model, which has learnt through quality-aware objectives, is capable of producing conservative minimum configurations when the intent is unclear which is preferrable in a production environment.

Validity Improvement: Specifically, the largest improvement comes from YAML validity which was initially 37.5% for the baseline and moved to 100% for the enhanced model, attesting the dual-encoder's prowess.

Operational Impact: The YAML validity enhancement (43.75% to 100%) eradicates a severe friction point that was introduced by the baseline model that needed to be corrected manually before testing. The 100% validity of the improved model makes it possible to deploy the improved model to the development environment instantly and to use it with validation tools (kubeval, kubscore) that expect input to be syntactically correct. Moreover, 54.5% of improved model results are production-ready compared to no production-ready baseline results, i.e. more than half of generated configurations do not require any steps during the manual review.

6.5 Limitations

While the results are encouraging, there is also a set of limitations which should be taken into account:

Sample Size: The assessment was based on only 8 test cases randomly selected from four quality categories, each composed of 2 cases. It is an under-powered analysis. Although it gained statistical significance ($p = 0.020$), the small sample size limits the generalisation of the results. A more extensive test set grouping 50+ cases per category is what one would need to further increase the statistical power and be able to see performance patterns that are absent in this work.

Dataset Constraints: About 130 production-grade Kubernetes manifests were included in the training dataset, which is relatively small compared to previous research. The insufficient training data could probably explain why the enhanced model is somewhat more conservative – sometimes it goes back to the defaulting mode when it is not familiar with new patterns and thus uses only a minimal configuration that is valid. The annotation overhead at which the matching of each manifest with natural language intent descriptions and generating quality-stratified variants (GOOD, NEARLY_GOOD, BAD, WORSE) affect the constrained dataset growth. In comparison to the automated collection of 276K configurations provided by GenKubeSec, extraction of intents and val-

idation of their quality required domain knowledge, and therefore, this work was human-annotation limited, as opposed to source-availability limited.

Focus on single resource type: Research has mainly focused so far on Deployments and Services. Multi-resource configurations (e.g., Deployment + Service + ConfigMap + Secret together) were never practiced. A coordinated multi-resource manifest is a requirement for typically running production Kubernetes applications, therefore, this is a pronounced gap in the evaluation format.

Hyperparameter Optimization: Hyperparameter tuning was conducted on a limited scale. The setting of the weights of the reward function (0.3 validity, 0.5 quality, -0.2 complexity) was based on intuition rather than systematic optimisation. Grid search or Bayesian optimisation over these weights could lead to even better results.

Ground Truth Subjectivity: The configurations that serve as ground truth used for CodeBLEU calculations are merely one out of many possible solutions of correct configurations that one can suggest. Kubernetes does permit the same purpose to be accomplished through various routes.

7 Conclusion and Future Work

The inquiry was focused on the research objective of how to implement the transformer-based language models so that it can not only create the Kubernetes YAML configurations but also provide users with quality feedback that is actionable with examples of configuration errors and show them how to reduce those configuration errors. Four main research objectives were pursued: first, establish baseline performance (RO1), second, design a more efficient dual-encoder architecture with the help of RL optimisation (RO2), third, develop a mechanism for actionable feedback (RO3) and fourth, create a comprehensive evaluation framework (RO4).

The main result is that the enhanced dual-encoder CodeT5 outperforms the baseline statistically significantly ($p = 0.020$, Cohen's $d = 1.13$). The validity of YAML increased from 37.5% to 100% after eliminating the baseline which tended to fabricate Kubernetes resource types. CodeBLEU's performance also improved from 39.2% to 68.3%, and, in the end, compliance with the best practices went from 31.8% to 45.5%. The evaluation result was ejected of a major trade-off: the enhanced model in the case of clear intents improves both validity and best practices, while for ambiguous intents, it prioritises generating valid minimal configurations over attempting to infer unstated requirements. This conservative style combined with the system of actionable feedback that points out missing best practices gives a more reliable pipeline than the baseline's unpredictable outputs.

The research gap identified in Section 1.1, in which systems are said to be concerned with syntactic correctness, without providing actionable feedback, was partly fulfilled. The newly developed system has made it possible for the DevOps practitioners to not just write valid environmental Kubernetes configurations from natural language descriptions, but also to get specific advice on how to better them which did not exist in fact at the baseline system that most of the time would have invalid configurations and without guidance on remediation.

From this, several future work directions emerge. First, scaling the training dataset to 1000+ variances will very likely enhance the model's capacity to deal with complex intents without compromising the best practices compliance. Second, the multi-resource

generation that creates ordered Deployment, Service, ConfigMap, and Secret configurations from a single idea would be a better depiction of production necessities. Third, the enhancement of the feedback loop that integrates the feedback mechanism with an iterative refinement in which the model automatically applies its recommendations could result in a higher quality output. The distance from the active feedback mode is helpful for educational purposes that distinguish it from pure generator tools rather than being used for companies to train junior DevOps engineers on Kubernetes best practices.

References

Ahmad, W. U., Chakraborty, S., Ray, B. and Chang, K.-W. (2021). Unified pre-training for program understanding and generation, *arXiv preprint* .
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* .
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* .
URL: *https://doi.org/10.48550/arXiv.2107.03374*

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: *https://doi.org/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* .
URL: *https://doi.org/10.48550/arXiv.2002.08155*

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: *https://doi.org/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* .
URL: *https://doi.org/10.48550/arXiv.2009.08366*

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* .
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* .
URL: *https://doi.org/10.48550/arXiv.2405.19954*

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: *https://doi.org/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: *https://doi.org/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* .
URL: *https://doi.org/10.48550/arXiv.2009.10297*

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* .
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* .
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* .
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* .
URL: *https://doi.org/10.48550/arXiv.2304.10778*