KubeBench: A Domain-Specific LLM Benchmark for Kubernetes Code Generation

Matt K. Robinson[1], Catherine Weiss[1], Anni Yao[1], Nick Cai[1], Tahlee Stone[1]

[1]University of California, Berkeley

2025-12-10

Abstract

Large Language Models (LLMs) have been rapidly adopted for infrastructure-as-code generation, particularly for Kubernetes configuration. However, traditional evaluation metrics—such as BLEU score and edit distance—are fundamentally unsuitable for assessing LLM-generated Kubernetes YAML configurations. These text-level similarity metrics fail to account for YAML's declarative and non-deterministic nature, penalizing syntactically different but functionally identical outputs. This renders such metrics "dead on arrival" for meaningful evaluation. In this paper, we introduce KubeBench, a novel Kubernetes-specific benchmark that extends prior work by KubeIntellect and CloudEval-YAML.

KubeBench: A Domain-Specific LLM Benchmark for Kubernetes Code Generation

In this paper, we introduce **KubeBench** , a novel Kubernetes-specific benchmark that extends prior work by KubeIntellect and CloudEval-YAML (Ardebili and Bartolini 2025). KubeBench is a three-dimensional evaluation framework that determines the quality of LLM generated YAML across three vectors: (1) runtime validity, (2) operational validity, and (3) evaluation of complete reasoning pathways by an LLM-as-judge . We benchmark more than 16 base, supervised fine tuned, and agentic open-source code generation LLMs on 810 Kubernetes-specific tasks spanning the foundational resources within a Kubernetes cluster. Our results and analysis contribute to a growing field of domain-specific artificial intelligence projects (Ni et al. 2025), and demonstrate the utility and effects of supervised fine tuning (SFT) and retrieval augmented (RAG) systems on domain-specific tasks in Kubernetes infrastructure engineering and operations. Our analysis of the effects of supervised fine-tuning were highly variable: while certain models showed marked improvement on specific task categories, others demonstrated marginal or even negative performance changes. Subsequent analysis underscores the non-uniform nature of domain adaptation across different model architectures and task types (Chen et al. 2025).

---

1. Introduction

The rapid adoption of Large Language Models (LLMs) for code generation has raised critical questions about their practical utility in general software engineering domains (Becker et al. 2025). General-purpose coding benchmarks like HumanEval and MBPP have proliferated, but these are insufficient for evaluating LLM performance on the niche domain of infrastructure-as-code tasks. There are also few benchmarks that can reliably measure the abilities of LLMs in highly specialized domains and thus we observe a preponderance of efforts to create domain-specific benchmarks and guidelines (Anjum et al. 2025). One such highly specialized domain that has thus sofar evaded a comprehensive LLM benchmark is the declarative markup language Yet Another Markup Language, or YAML Ain't Markup Language, or YAML as it is known colloquially. YAML is a human-readable data serialization language commonly used for configuration files. In Kubernetes, YAML

files serve as the primary interface for defining and configuring infrastructure resources—specifying everything from container images and networking rules to storage volumes and security policies in a declarative, text-based format that the Kubernetes API can interpret and execute.

In particular, there is lack of a general evaluation benchmark and framework for Kubernetes YAML generation that aligns with a goal of immediate productivity increase from AI adoption. Recent research has primarily targeted autonomous, self-healing cluster infrastructure (@ Ardebili and Bartolini 2025). While this is a a worthy long-term vision, it nonetheless remains distant from production deployment and fails to address the day-to-day challenges faced by teams managing diverse Kubernetes environments today. The fundamental challenge with developing Kubernetes expert systems and benchmarking their real-world effectiveness lies in how we measure effectiveness, not in how soon we can remove the human domain expert.

The Impracticality of Traditional Benchmarks for YAML We assert that traditional text-level similarity metrics, such as BLEU and Edit Distance, are impractical when evaluating LLMs specializing in generating Kubernetes configuration code. This inadequacy stems directly from the nature of the configuration language (Xu et al. 2023; Ardebili and Bartolini 2025). In Kubernetes systems, YAML is declarative and Non-Deterministic and serves as the de facto standard for cloud-native deployment and system management tooling. Because YAML is a declarative configuration language the object order of key-value pairs is not critical for functional correctness. In other words, two YAML files with identical field values but different orderings are functionally equivalent and are interpreted the same by a Kubernetes system's manager node. This attribute means that traditional evaluation metrics would penalize the differential between two functionality equivalent Kubernetes YAML configurations. Other text-level metrics like BLEU measure lexical similarity between generated YAML and reference YAML. Again, these metrics do not have a way to evaluate without concern for ordering of key-value pairs. This means that BLEU systematically fails to align with YAML's semantic properties where field ordering is not relevant to the manager node responsible for validating and processing Kubernetes YAML.

Traditional Code Generation Benchmarks The field of evaluation of LLM generated code has been dominated by general-purpose benchmarks such as HumanEval, MBPP, and CodeXGLUE. These benchmarks typically employ unit testing for functional correctness but focus on imperative programming languages (Python, Java, JavaScript). Their evaluation paradigms do not translate to declarative infrastructure-as-code domains.

Attempts at domain-specificity There are numerous attempts at domain-specificity in LLM benchmarking, and they tend to focus on either developing guidelines for benchmarking, or on evaluation and benchmarking in the researchers' particular areas of concern. Two recent examples from 2025 demonstrate that the field of domain-specific benchmarks is young, fragmented, and constantly evolving. (yunhan and gengshen 2025). A recent survey of LLM benchmarks found 283 LLM benchmarks and divided them into general, domain-specific, and target-specific benchmarks (Ni et al. 2025). It is given that there are more LLM benchmarks than those identified in the survey of domain-specific benches, and that the LLM benchmark field is heavily saturated and is particularly confusing for the business community which has to contend with immediate goals - e.g. increasing productivity, obtaining clear ROI - and appropriate evaluation. As Ni et al. (2025) observe, "current benchmarks have problems such as inflated scores caused by data contamination, unfair evaluation due to cultural and linguistic biases, and lack of evaluation on process credibility and dynamic environments" (p. 1).

---

2. Prior Work

KubeIntellect The KubeIntellect framework represented a move shift toward domain-specific evaluation for Kubernetes code generation. KubeIntellect established that BLEU scores and edit distance are insufficient for assessing infrastructure code by emphasizing executable outcomes and semantic similarity metrics over traditional text-based comparisons. The framework validates generated configurations through sandboxed execution environments to measure whether configurations successfully deploy and operate as intended.

CloudEval-YAML CloudEval-YAML advanced this approach by introducing a comprehensive evaluation methodology specifically designed for Kubernetes YAML generation. First, researchers introduced YAML-aware metrics that respect the order-independence of dictionary structures. Second, researchers introduced a "key-value wildcard matching" strategy that allows flexibility on non-critical fields (e.g., image versions) while enforcing strictness on critical configuration elements. Lastly, CloudEval-YAML used unit tests executed in live cluster environments to provide functional validation of generated YAML. Our team concurs with this work's assertion that traditional metrics correlate poorly with functional correctness in Kubernetes configurations (Xu et al. 2023).

---

3. The Kubernetes Benchmark

KubeBench: Extended Evaluation with Reasoning Assessment Building directly on these foundations, KubeBench introduces three innovations. First, we stratified tasks by complexity to better reflect real-world Kubernetes scenarios and prepare models for diverse requirements coming from their human interaction patterns. Tasks in KubeBench are classified as either `easy`, `intermediate`, or `advanced`. We provide examples of same tasks in appendix (see Appendix @ref(appendix-tasks)). Second, our work extends evaluation to post-deployment operational correctness (e.g. verifying that access control modifications actually took place) by using the Kubernetes API to perform `get` requests on the resources created by generated Kubernetes YAML. It is essential to do this because Kubernetes configuration's declarative nature allows for partial validation of configuration. In other words, if three resources are defined within a configuration, one or two of those resources may be created, while the third draws a non-segmenting error. The configuration is thus valid, but not all resources defined are guaranteed to be present. Lastly, we introduce an LLM-as-judge to validate the full reasoning pathway and decision making process demonstrated by models' responses to tasks of variable difficulties and context of variable complexities. This allows our benchmark to go beyond binary success or failure, and employ the LLM evaluator to assess the reasoning pathway and decision-making process in a similar way that an AI agent may be used in a production code generating workflow.

Taken as a whole, these extensions to evaluate runtime validity, operational validity, and reasoning pathway address the full lifecycle of Kubernetes configuration tasks from code generation through deployment, operations, and reasoning verification. We thus attempt to resolve issues that exist among current benchmarks and offer an iterative Kubernetes benchmarking framework grounded in the principle of comprehensiveness and compactness (@ Ni et al. 2025) (p.1).

3.1 Task Specification

This current iteration of KubeBench comprises 810 Kubernetes-specific `CREATE` tasks spanning eight foundational Kubernetes resource categories: `clusterrole`, `clusterrolebinding`, `configmap`, `namespace`, `role`, `rolebinding`, `secret`, and `serviceaccount`. These resources represent the bedrock of any production-level Kubernetes cluster. A Kubernetes cluster cannot realistically function in a production environment without them properly defined and configured.

These foundational resources serve critical infrastructure functions that every cluster requires: `namespace` provides logical isolation between different applications or teams; `configmap` and `secret` store application configuration data and sensitive credentials respectively; while `role`, `rolebinding`, `clusterrole`, `clusterrolebinding`, and `serviceaccount` collectively implement Role-Based Access Control (RBAC). RBAC determines which users, applications, and processes have permission to perform which operations within the cluster. A cluster is either completely insecure or completely non-functional without RBAC correctly and securely configured.

What makes these resources particularly challenging for LLM code generation is their extreme variability and customization potential. Unlike deploying a simple web application that follows relatively standard patterns, these foundational resources must be tailored to each organization's specific security policies, compliance requirements, multi-tenancy needs, and operational workflows. For example, a financial services company's RBAC configuration will differ fundamentally from a research institution's namespace strategy or a SaaS platform's secrets management approach. The combinatorial space of valid, secure, and operationally sound

configurations for these resources is effectively infinite. This attribute makes them making them ideal starting test cases for evaluating whether LLMs can generate contextually appropriate infrastructure code rather than merely reproducing common templates.

3.2 YAML template generation (Create operations) We focused this initial iteration of KubeBench exclusively on CREATE operations and YAML template generation for several strategic reasons. First, scaffolding new infrastructure configurations represents the highest-frequency task in typical Kubernetes workflows—engineers constantly define new namespaces for projects, create RBAC policies for new services, generate ConfigMaps for application deployments, and establish ServiceAccounts for automation tooling. Second, CREATE operations expose the full complexity of each resource type without the additional context dependencies required for UPDATE or DELETE tasks, making them ideal baseline evaluation targets. Third, template generation from natural language requirements represents the most immediate productivity opportunity for LLM assistance . Rather than consulting documentation and manually writing boilerplate YAML, productivity gains are be demonstrated when human engineers can describe their intent and iterate on generated configurations, with a strong preference for fewer conversational iterations (Hyun et al. 2025). We create a foundation for expanding to READ, UPDATE, and DELETE operations that require additional cluster state awareness and change impact reasoning by establishing rigorous evaluation standards for CREATE operations where models must first demonstrate understanding of resource schemas, security implications, and operational context.

3.3 Evaluation Framework **Runtime Validity**: Does the generated YAML validate successfully against the Kubernetes cluster specifications without error?

Operational Validity: If the configuration deployed successfully to a Kubernetes cluster, do the declared resources exist?

Operational Accuracy: Does the modification achieve the intended effect? For example:

- If creating or modifying a `role` or `clusterrole`, does the resulting resource grant exactly the specified permissions to the intended API resources and verbs?
- If establishing a `rolebinding` or `clusterrolebinding`, does it appear that the the bound `serviceaccount` or user could now successfully perform the permitted operations and can we see that they would be denied unauthorized operations?

LLM-as-Judge Reasoning Assessment: An additional LLM agent has two evaluation pathways to analyze the models' reasoning pathways and then it comments on whether they appear to have been appropriate for the task. The LLM judge evaluates generated YAMLs for all cases, even in cases of failed execution. We then sometimes use this error analysis in subsequent discussions in this paper, and in retraining and evaluation of the benchmarking methodology.

These evaluation dimensions form a multi-stage pipeline that assesses each generated manifest through validation gates. Each stage filters for aspects of correctness: syntactic validity precedes deployment success, which precedes operational correctness, which precedes reasoning quality assessment. This staged approach provides granular diagnostic information about failure modes. For example, a model that passes runtime validity but fails operational accuracy exhibits different deficiencies than one that cannot generate syntactically valid YAML. The evaluation pipeline as it is configured here feeds a closed-loop refinement process where failure patterns can inform synthetic test case generation. In turn the synthetic test cases produce targeted training examples for subsequent fine-tuning. Our evaluation framework thus creates a continuous improvement cycle where the benchmark both measures model performance and actively drives model enhancement through error analysis and data augmentation.

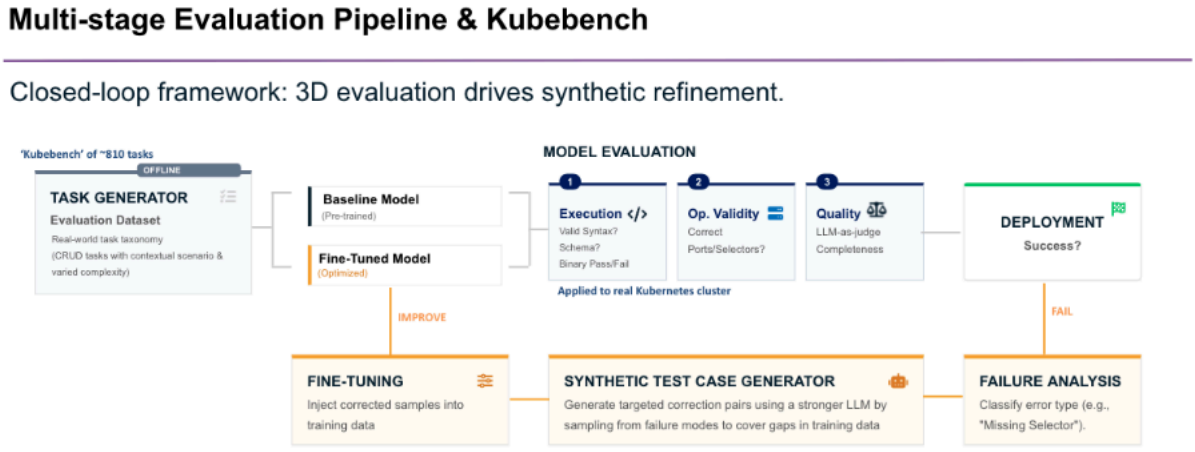

Figure 1: Multi-stage Evaluation Pipeline

4. Experimental Design

Our Kubernetes benchmarking philosophy stems from the below formal hypothesis that a Supervised Fine Tuning (SFT) intervention will affect performance of a base model on the set of domain-specific Kubernetes tasks. The null hypothesis that the our SFT intervention will have no effect on models' performance on the benchmark tasks and that in this case there will be no differential between models' performance. The alternative hypothesis states that the SFT intervention will affect the models' performance and we will observe a differential between SFT models and base models' performance on the Kubernetes domain-specific tasks. We then extend the hypothesis to include a quantitative methodology to test the null hypothesis and determine whether SFT had any impact. With the methodology run, we then move to either reject the null hypothesis or fail to reject the null for the models' in question.

4.1 Formal Hypothesis

Let

$$P_{default}$$

represent the performance of default coding LLMs and

$$P_{finetuned}$$

represent the performance of fine-tuned coding LLMs on language-specific tasks, where performance is measured as a score of execution validity, operational validity, and troubleshooting utility.

Null Hypothesis (H):

$$H_0 : P_{default} = P_{finetuned}$$

Alternative Hypothesis (H):

$$H_a : P_{default} \neq P_{finetuned}$$

Performance Evaluation Framework

Performance can be operationalized as a weighted score:

$$P = w_1 \cdot E_v + w_2 \cdot O_v + w_3 \cdot T_u$$

Where: -

$$E_v$$

= Execution validity (binary: 0 or 1) -

$$O_v$$

= Operational validity (binary: 0 or 1) -

$$T_u$$

= Troubleshooting utility (binary: 0 or 1) -

$$w_1, w_2, w_3$$

= weights such that

$$w_1 + w_2 + w_3 = 1$$

4.2 Model Selection We selected four models to make available on our website. The four models were selected for fine-tuning based on demonstrated code generation capabilities, parameter efficiency, and diverse architectural approaches. The selection spans 0.5B to 8B parameters to evaluate how domain-specific fine-tuning scales across model capacity and whether smaller models can achieve production-grade performance on constrained infrastructure tasks.

Qwen K8s Assistant – 0.5B (qwen_0.5b_k8s-qlora_v4): Ultra-lightweight model optimized for manifest generation and quick scaffolding in resource-constrained environments. Ideal for CI/CD pipelines, edge deployments, and rapid iteration on small clusters.

Gemma 2 – K8s Deep Reasoner 2B (gemma_2b_k8s-qlora_v4): Specialized for complex reasoning tasks including migration planning, debugging sessions, and architecture reviews. Excels at providing detailed explanations and "what & why" narrative alongside configurations.

Qwen K8s Specialist – 1.5B (qwen_1.5b_k8s-qlora_v3): Balanced model with enhanced multi-step reasoning for production workflows and troubleshooting. Strong performance on rollout planning, CI/CD configurations, and operational tasks.

Llama 3.1 – K8s Expert 8B (llama_v4): Production-grade model deeply fine-tuned for RBAC, controllers, operators, and enterprise deployment patterns. Provides opinionated best-practice defaults with both code generation and architectural guidance.

This selection allows us to evaluate whether fine-tuning benefits generalize across architectural families and parameter scales, and whether smaller models can match larger baselines through domain-specific training.

4.3 Evaluation Implementation The evaluation methodology implements a three-stage validation pipeline that progressively assesses model-generated Kubernetes configurations. Each stage applies distinct validation criteria, with subsequent stages operating only on tasks that successfully passed preceding validation.

Runtime Evaluation Stage The runtime evaluation stage tests whether model-generated YAML can be successfully applied to a Kubernetes cluster. Each evaluation runs within a controlled security context using a simulated RBAC role (`system:serviceaccount:default:evaluation-api-sa`) to ensure consistent permission boundaries. For each task, the generated YAML is written to a temporary file and applied to a dedicated evaluation namespace using `kubectl apply`. The operation's return code determines runtime validity: zero indicates successful application (`runtime_valid=True`), while any non-zero return code results in failure (`runtime_valid=False`). This stage filters out syntactically invalid or inapplicable configurations before proceeding to operational verification.

```
# Runtime evaluation process
apply_result = kubectl.apply(
    yaml_template,
    namespace='evaluation-namespace',
    service_account='evaluation-api-sa'
)
runtime_valid = (apply_result.return_code == 0)
```

Operational Evaluation Stage The operational evaluation stage verifies that resources created during runtime evaluation persist and remain accessible within the cluster. This stage operates exclusively on tasks that achieved `runtime_valid=True` status. The evaluation process parses standard output from the runtime stage to extract the resource type and name of each created Kubernetes resource. For each identified resource, the system executes `kubectl get` to retrieve the resource's current state from the cluster. Successful retrieval (return code zero) results in `operational_valid=True`, while failure to retrieve results in `operational_valid=False`. This stage ensures that resources not only apply successfully but also exist as queryable entities within the cluster.

```
# Operational evaluation process
created_resources = parse_resources_from_stdout(runtime_stdout)
for resource in created_resources:
    get_result = kubectl.get(
        resource_type=resource.type,
        resource_name=resource.name
    )
    operational_valid = (get_result.return_code == 0)
```

LLM Judge Evaluation Stage The LLM judge evaluation stage provides semantic assessment of resource quality and alignment with task requirements. This stage operates on all tasks regardless of whether they achieved both `runtime_valid=True` and `operational_valid=True` status. The evaluation employs either KimiK2Instruct or ChatGPT-5.1 as the judge model and is configured with structured output constraints to assess three dimensions: task alignment, resource verbosity, and scope adherence. The judge receives the actual resource configuration (retrieved via `kubectl get`), the original task description, and contextual scenario information. The evaluation produces three outputs: a boolean `flags` value indicating whether the resource exhibits misalignment or unnecessary complexity, a `flag_detail` string providing explanatory reasoning, and a `quality_score` float ranging from 0.0 to 1.0 that quantifies alignment with task requirements. This stage filters resources that, while technically valid, fail to meet semantic or qualitative requirements specified in the task description.

The three-stage pipeline implements progressive filtering where each stage applies increasingly sophisticated validation criteria to an increasingly refined subset of tasks, to produce a comprehensive assessment of models' capabilities across technical validity, operational persistence, and semantic quality dimensions of their generated YAML templates.

5.0 Task Specification

We have selected an array of 810 kubernetes tests, each with one unique requirement and specific context. The code generation tasks of this current KubeBench iteration can be classed into the sub-classes: YAML template generation (C).

For each of the task sets and evaluation criterion, there is a preferred strict binary response, and the only flexibility allowed is that there is an additional marker if the command was successful, but there was a warning from a local or cloud system. This binary response evaluation variable applies for all sets of tasks between YAML templates and CLI commands. To validate the operational validity and accuracy of CLI commands, an additional LLM agent was used which to query whether the information returns from any successful CLI command either met expectations or was unable to obtain the correct information under valid application (200 code, no error code, but wrong information). The further validate the operational accuracy of issued commands, a set of API requests targeting specific IPs or URLs are sent after successful application of generated commands to further measure whether the successfully applied commands had either the desired operational effect (such as adding or removing IPs, changing a network configuration, or modification resource allocation) or was erroneous but did not draw any hard runtime errors.

6.0 Project Intent Kubernetes has decisively won the cloud orchestration wars, with enterprises across industries investing billions of dollars in infrastructure and employing hundreds of thousands of human en-

gineers responsible for either working with or directly managing extraordinarily diverse deployments. These deployments span financial services, healthcare, e-commerce, machine learning platforms, IoT systems, and countless other domains with unique compliance requirements, performance constraints, security postures, and operational patterns. For example, a banking institution's Kubernetes deployment bears little resemblance to a genomics research cluster or a real-time gaming platform's infrastructure.

This diversity presents a fundamental challenge to the autonomous agent approach: **it is highly unlikely that any single LLM agent, regardless of how well-trained, can achieve expert-level proficiency across all possible Kubernetes deployment patterns, industry-specific requirements, and organizational constraints.** The knowledge required to navigate HIPAA-compliant healthcare clusters differs fundamentally from that needed to optimize high-frequency trading infrastructure or manage edge computing deployments across thousands of retail locations.

While we build upon the evaluation frameworks established by KubeIntellect and CloudEval-YAML (Xu et al. 2023; Ardebili and Bartolini 2025), our research diverges significantly in its ultimate objective. KubeIntellect's vision of multi-agent LLM systems capable of autonomously managing entire Kubernetes clusters represents an ambitious and technically sophisticated endeavor. However, we contend that this autonomous agent paradigm, while valuable for research, may be impractical for the current state of the Kubernetes ecosystem.

Rather than pursuing full automation through autonomous agents, our focus centers on **augmenting human infrastructure engineers** who possess irreplaceable domain expertise, organizational context, and decision-making authority. These engineers already understand their specific deployment environments, regulatory requirements, and business constraints. What they need is not replacement, but rather a more efficient, contextually aware, and factually grounded tool for interacting with Kubernetes' rapidly evolving ecosystem.

The infrastructure engineering domain faces unique challenges that distinguish it from general software development:

1. **Documentation Velocity**: Kubernetes releases occur quarterly, with each release introducing new features, deprecations, and API changes. Cloud providers (GCP, AWS, Azure) continuously extend Kubernetes with proprietary services and custom resource definitions. Keeping current with best practices across this moving target is a substantial cognitive burden for human engineers.
2. **Context Switching Costs**: Infrastructure engineers frequently move between different cluster environments and each cluster may have its distinct configurations, networking policies, storage backends, and service mesh implementations. Recalling the specific syntax, available resources, and operational patterns for each context imposes significant overhead.
3. **High Stakes of Errors**: Unlike application code that can be quickly rolled back, infrastructure misconfigurations can cause cascading failures, security vulnerabilities, or compliance violations with severe business consequences. Engineers need high-confidence, verifiable answers; and probabilistic approximations are markedly less useful and errors are hard to identify in YAML template configurations alone.
4. **Complexity of Troubleshooting**: Diagnosing failures in distributed systems requires correlating evidence across logs, metrics, network policies, resource quotas, RBAC configurations, and application state. An LLM tool that can understand Kubernetes contextss and synthesize relevant documentation and suggest diagnostic approaches based on current Kubernetes best practices could dramatically reduce mean time-to-resolution.

Our objective, therefore, is to create a **domain-adapted LLM assistant** that serves as an expert consultant for human engineers. One that has been fine-tuned on the most recent Kubernetes documentation, understands the semantic structure of YAML configurations, and can provide factually grounded recommendations that engineers can validate and deploy with confidence. This approach respects the reality that human judgment remains essential for infrastructure decisions while acknowledging that LLMs can substantially reduce the friction of working with complex, rapidly evolving tools.

By focusing on human-in-the-loop augmentation rather than full autonomy, we also sidestep the trust and

safety challenges inherent in fully-autonomous infrastructure agents. Engineers can leverage the LLM's broad knowledge of Kubernetes patterns while retaining final authority over what gets deployed to production systems. This paradigm is far more compatible with existing organizational workflows, change management processes, and regulatory compliance requirements than fully-autonomous agent systems.

The KubeBench benchmark is designed specifically to measure progress toward this human-augmentation goal. Our evaluation criteria directly correspond to what a human engineer needs from an LLM assistant: syntactically correct configurations, functionally valid deployments, and transparent reasoning that can be audited and understood. The inclusion of complexity tiers and diverse task types aims to reflect the diversity of real-world scenarios that production engineers encounter and our fine-tuning experiments test whether domain-specific knowledge adaptation can produce the factual accuracy and contextual awareness that infrastructure engineering work demands.

7.0 Finetuning and Training Data Strategy Dataset Sources

The Stack

Our fine-tuning dataset is derived from The Stack, a comprehensive corpus containing over 6TB of permissively-licensed source code files covering 358 programming languages (Kocetkov et al. 2022). Created as part of the BigCode Project—an open scientific collaboration focused on the responsible development of Large Language Models for Code—The Stack serves as a foundational pre-training dataset for Code LLMs. These code-generating AI systems enable program synthesis from natural language descriptions as well as code completion and transformation from existing snippets.

From The Stack's extensive collection, we extracted and focused specifically on YAML configurations from Kubernetes-related repositories. We implemented rigorous data quality controls including filtering for correctness (removing buggy or outdated examples), validating against current Kubernetes API schemas, and ensuring structural consistency. The resulting dataset is cleaned, structured, and formatted specifically for fine-tuning workflows—prioritizing examples that demonstrate best practices for the eight foundational Kubernetes resources evaluated in KubeBench.

Kubernetes Official Documentation

We curated an additional dataset from Kubernetes open source documentation from official Kubernetes websites covered by the Creative Commons CC-4.0 and Apache 2.0 licenses. Documentation sources included:

- Kubernetes official documentation (kubernetes.io)
- Kubernetes Developer Documentation
- kubectl official documentation
- kubectl Developer Documentation

The combined datasets contained hundreds of thousands of valid Kubernetes samples for implementing fine-tuning workflows across our selected models.

Fine-Tuning Process

We applied QLora fine-tuning to all selected models using the curated Kubernetes documentation corpus. The entire documentation set was scraped from public URLs and structured into instruction-tuning formats compatible with each model architecture.

Instruction Fine-Tuning Data Structure

Fine-tuning data followed established instruction-tuning patterns adapted for Kubernetes documentation. We adopted a structure similar to documentation-focused fine-tuning approaches used for technical reference materials:


```
{
  "text": "Class: ClusterRole\nUsage: Defined in apiVersion: rbac.authorization.k8s.io/v1\nkind: Cluste
}
```

8.0 Project Accomplishments & Roadmap

Overview

KubeBench was designed to address a fundamental gap in infrastructure engineering: general-purpose LLMs often generate Kubernetes code that appears correct but fails in production. By fine-tuning domain-specific models and evaluating them through actual cluster deployment, we've built a platform that delivers measurable improvements in code quality and completed this work with the goal of supporting greater developer productivity and safe deployment of AI agents.

Model Development & Fine-Tuning Results

We applied QLoRA to train 16+ Kubernetes-specialized models across different size classes. We trained models of varying sizes from ultra-lightweight 0.5B parameter models suitable for CI/CD pipelines to more capable 8B parameter models for complex reasoning tasks; and have made four of the best models available on KubeBench.dev. The results demonstrate clear performance improvements over baseline models:

- **+18–25% execution validity**: Generated YAML successfully validates and applies to Kubernetes clusters
- **+21–27% YAML quality**: Configurations align with best practices and operational requirements
- **+18–23% overall benchmark score**: Combined improvement across runtime validity, operational correctness and reasoning assessment

Notably, smaller models (0.5B–2B parameters) benefited most from domain-specific fine-tuning. This finding has significant implications for edge deployments and resource-constrained environments where developers still need fast and accurate assistance but are without internet or large compute.

Results from finetuned_Llama3.1B (8B parameters)

Table 1: Table 1: Runtime Evaluation Summary - Fine-tuned Llama 3.1 8B

Task Type	Complexity	Total	Success	Success Rate
Clusterrole	basic	30	29	96.7%
Clusterrole	intermediate	30	22	73.3%
Clusterrole	advanced	30	11	36.7%
Clusterrolebinding	basic	30	20	66.7%
Clusterrolebinding	intermediate	30	15	50.0%
Clusterrolebinding	advanced	30	11	36.7%
Configmap	basic	30	28	93.3%
Configmap	intermediate	30	24	80.0%
Configmap	advanced	30	30	100.0%
Namespace	basic	30	27	90.0%
Namespace	intermediate	30	14	46.7%
Namespace	advanced	30	24	80.0%
Role	basic	30	29	96.7%
Role	intermediate	30	19	63.3%
Role	advanced	30	17	56.7%
Rolebinding	basic	30	30	100.0%
Rolebinding	intermediate	30	24	80.0%
Rolebinding	advanced	30	27	90.0%
Secret	basic	30	10	33.3%
Secret	intermediate	30	7	23.3%
Secret	advanced	30	13	43.3%
Serviceaccount	basic	30	27	90.0%

Task Type	Complexity	Total	Success	Success Rate
Serviceaccount	intermediate	30	24	80.0%
Serviceaccount	advanced	30	28	93.3%
OVERALL	ALL	720	510	70.8%

Table 1 presents runtime evaluation results for the fine-tuned Llama 3.1 8B model, achieving 70.8% overall success rate across 720 tasks. This model represents our best-performing configuration and demonstrates the potential of QLoRA fine-tuning for domain-specific code generation. Comprehensive evaluation results for all model variants, including detailed performance comparisons against baseline models and statistical analysis of improvement margins, will be reported in future iterations of this work.

Benchmark Pipeline Development

We built a repeatable and scalable benchmark generator that currently covers the eight core Kubernetes resource types with a focus on CREATE operations. The evaluation framework automates scoring for any model against baseline performance through three-stage validation: schema compliance, successful cluster deployment and operational correctness assessment via LLM-as-judge. This pipeline enables rapid iteration on model development and provides researchers with a standardized methodology for comparing Kubernetes code generation capabilities.

Production Deployment

The platform is fully deployed on cloud infrastructure with FastAPI inference endpoints powering an interactive web interface. Users can engage with working chat functionality, explore model capabilities and download any of the four fine-tuned model files hosted for community use and research.

Key Learnings from Initial Deployment

Early user feedback and evaluation results revealed important insights about fine-tuning for infrastructure code generation. Fine-tuned models demonstrate substantially deeper Kubernetes knowledge and greater specificity than their base model counterparts, particularly in generating correct RBAC configurations and understanding resource relationships. However, fine-tuning alone may not produce the desired effect on already instruction-tuned models at larger sizes; achieving strong performance required substantial additional work in prompt engineering, dataset curation and evaluation methodology refinement.

Importantly, user testimonials confirm that perfection is neither a desired nor expected outcome. Infrastructure engineers value models that reduce cognitive overhead and accelerate initial scaffolding, even if manual review and adjustment remains required. This aligns with our design philosophy: augmenting human expertise rather than replacing it.

Future Directions

Benchmark Expansion: Our immediate priority is extending coverage beyond CREATE operations to include READ, UPDATE and DELETE tasks, as well as CLI generation scenarios. We aim to expand from the current 810 tasks to a 10,000-task research-grade benchmark covering the full spectrum of Kubernetes resource types and operational workflows.

Fine-Tuning Improvements: We will increase dataset curation through enhanced filtering, deduplication and validation against current Kubernetes API schemas. Experimentation with hybrid training approaches combining QLoRA with targeted instruction tuning may improve performance on complex multi-resource tasks. Adding automated loss-curve analysis and error-pattern detection will accelerate training iteration cycles.

Front-End & Agent Capabilities: Planned upgrades include context window management for session memory, unique chat experiences for each fine-tuned model, file and document upload features for context-

aware generation, multi-file editing with live schema validation and an "explain mode" that shows why the model generated each field.

Architectural Evolution: Long-term development includes RAG integration with official Kubernetes documentation for real-time reference, domain-specific context templates tailored to common infrastructure workflows and agent orchestration capabilities for handling larger multi-step infrastructure tasks that span multiple resource types and deployment stages.

KubeBench demonstrates that domain-specific fine-tuning of coding LLMs can address the inaccuracy and inefficiency of general-purpose models when generating infrastructure code. The result is more reliable YAML output, faster developer workflows and measurable productivity gains for engineers managing production Kubernetes environments.

Appendix: Sample Benchmark Tasks

ClusterRole Tasks

Example 1: Basic CREATE Task

Task ID: ClusterRole_C_c2u7

Operation Type: CREATE

Complexity Level: Basic

Description: Create a ClusterRole allowing access to list and get services across all namespaces for a monitoring tool.

Context Scenario: The corporate DevOps team must deploy a monitoring microservice to generate topology maps of all services in the cluster, but the tool requires read-only access to services in all namespaces.

Example 2: Advanced CREATE Task

Task ID: ClusterRole_C_z3l2

Operation Type: CREATE

Complexity Level: Advanced

Description: Create a ClusterRole to permit a team to manage all custom resources in the 'logging.openshift.io' and 'networking.k8s.io' API groups across the cluster while maintaining read access to all other resources in these API groups for non-privileged users.

Context Scenario: An enterprise is deploying a multi-tenant Kubernetes cluster where one team requires exclusive control over logging and networking CRDs but must preserve baseline read permissions for non-privileged users to avoid audit gaps in these critical systems.

Role Tasks

Example 1: Intermediate CREATE Task

Task ID: Role_C_sz2r

Operation Type: CREATE

Complexity Level: Intermediate

Description: Create a Role that grants get and describe access on persistent volumes (pv) in a specific namespace, but prohibits deletion or modification of any volume claims in 'shared-storage-ns'. The role should require proper subject binding constraints.

Context Scenario: A namespace requires storage access for analytics engines, but must prevent accidental volume deletion. Access should be scoped to allow PV observations while maintaining separate rules for actual storage operations.

ConfigMap Tasks

Example 1: Intermediate CREATE Task

Task ID: ConfigMap_C_qatc

Operation Type: CREATE

Complexity Level: Intermediate

Description: Create a ConfigMap that stores connection details for RabbitMQ queues used by 12 different services. Each service should have a unique key format (e.g., svc-.queue-dsn), and include environment variables compatibility notes.

Context Scenario: Microservices orchestration system migration from Docker Swarm required reformatting messaging queue connection details for Kubernetes, while maintaining backward compatibility with the application layer.

Cluster Specifications

Evaluation tasks were executed on two distinct Kubernetes environments to ensure benchmark validity across both local development and production cloud infrastructure.

Local Development Environment: Minikube

Cluster Configuration:

- **Kubernetes Version:** v1.32.1 (server), v1.32.2 (client)
- **Minikube Version:** v1.35.0
- **Driver:** Docker
- **Container Runtime:** Docker
- **Nodes:** 1

Host Machine Specifications:

- **Model:** MacBook Pro (2020, Model ID: MacBookPro17,1)
- **Chip:** Apple M1
- **CPU Cores:** 8 (4 performance, 4 efficiency)
- **Memory:** 16 GB
- **OS:** macOS 14.4 (Sonoma)

Production Environment: Google Kubernetes Engine (GKE) Autopilot

Cluster Configuration:

- **Name:** yoelo-autopilot-cluster
- **Type:** GKE Autopilot (fully managed)

- **Region:** us-central1 (multi-zone: a, b, c, f)
- **Kubernetes Version:** v1.33.5-gke
- **Current Nodes:** 4 (autoscaling)
- **Container Runtime:** COS_CONTAINERD
- **Datapath:** Advanced (GKE Datapath v2)

Available Machine Types (Autoscaling Node Pools):

- **Compute-optimized:** ek-standard-8, ek-standard-16, ek-standard-32
- **General-purpose:** e2-medium, e2-standard-2/4/8/16/32
- **GPU-enabled:** n1-standard-32 with NVIDIA Tesla T4 (1 GPU)
- **Max nodes per pool:** 1000

Key Features:

- Autopilot mode with automatic node provisioning and management
- Vertical and horizontal pod autoscaling
- Managed Prometheus monitoring
- Shielded GKE nodes (Secure Boot + Integrity Monitoring)

References

Anjum, Khizar, Muhammad Arbab Arshad, Kadhim Hayawi, Efstathios Polyzos, Asadullah Tariq, Mohamed Adel Serhani, Laiba Batool, et al. 2025. "Domain Specific Benchmarks for Evaluating Multimodal Large Language Models." https://arxiv.org/abs/2506.12958.

Ardebili, Mohsen Seyedkazemi, and Andrea Bartolini. 2025. "KubeIntellect: A Modular LLM-Orchestrated Agent Framework for End-to-End Kubernetes Management." https://arxiv.org/abs/2509.02449.

Becker, Joel, Nate Rush, Elizabeth Barnes, and David Rein. 2025. "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity." https://arxiv.org/abs/2507.09089.

Chen, Rubing, Jiaxin Wu, Jian Wang, Xulu Zhang, Wenqi Fan, Chenghua Lin, Xiao-Yong Wei, and Qing Li. 2025. "Benchmarking for Domain-Specific LLMs: A Case Study on Academia and Beyond." https://arxiv.org/abs/2508.07353.

Hyun, Sangwon, Hyunjun Kim, Jinhyuk Jang, Hyojin Choi, and M. Ali Babar. 2025. "Experimental Analysis of Productive Interaction Strategy with ChatGPT: User Study on Function and Project-Level Code Generation Tasks." https://arxiv.org/abs/2508.04125.

Kocetkov, Denis, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, et al. 2022. "The Stack: 3 TB of Permissively Licensed Source Code." *Preprint.*

Ni, Shiwen, Guhong Chen, Shuaimin Li, Xuanang Chen, Siyi Li, Bingli Wang, Qiyao Wang, et al. 2025. "A Survey on Large Language Model Benchmarks." https://arxiv.org/abs/2508.15361.

Xu, Yifei, Yuning Chen, Xumiao Zhang, Xianshang Lin, Pan Hu, Yunfei Ma, Songwu Lu, et al. 2023. "CloudEval-YAML: A Practical Benchmark for Cloud Configuration Generation." https://arxiv.org/abs/2401.06786.

yunhan, Li, and Wu gengshen. 2025. "LegalEval-q: A New Benchmark for the Quality Evaluation of LLM-Generated Legal Text." https://arxiv.org/abs/2505.24826.