

Politecnico di Torino

Politecnico di Torino

Master's Degree in Computer Engineering

Academic Year 2025/2026

Graduation Session September 2026

Intent-Driven Kubernetes Manifest Generation: An Agentic Approach for Cloud-Native Workloads

Supervisors:

Alessio Sacco

Guido Marchetto

Stefano Braghin

Candidate:

Elisa Bariselli

Abstract

Cloud-native systems have revolutionized the development, deployment, and management of modern applications, enabling organizations to achieve unprecedented scalability and resilience. Despite these advantages, configuring and operating such environments remains complex, particularly for users without extensive technical expertise. A recent industry trend, Agentic AI, leverages large language models (LLMs) capable of orchestrating multiple tools to automate configuration tasks and reduce this operational burden.

This thesis presents an intent-based framework that employs LLMs to automate the generation of configurations in cloud-native ecosystems, using Kubernetes workloads as the primary case study. Although LLMs have achieved remarkable progress in natural language generation, producing correct, reliable, and domain-specific structured configurations remains challenging. To address this, we design a complete pipeline that begins with intent interpretation and prompt formulation, proceeds through iterative manifest generation, and incorporates validation mechanisms and error-aware refinement loops to progressively improve configuration quality across varying levels of complexity.

Our empirical evaluation demonstrates that an agent-based approach enhances reliability compared to single-shot LLM generation.
In our findings, larger models such as `mistral-large-2512` and `gpt-oss-120b` produce higher quality manifests with favorable cost–performance characteristics, while smaller models show greater difficulty adapting to iterative correction and self-refinement. On the other hand, the proposed pipeline substantially improves the robustness of configuration generation by integrating structured validation and feedback loops.

These findings highlight the benefits of our framework to increase accessibility and efficiency of cloud-native technologies, making complex deployment tasks more approachable for a broader range of users. They also surface key open challenges in adapting LLMs to the management of structured, domain-specific, language generation, pointing toward relevant and promising directions for future research.

Table of Contents

List of Tables IV

List of Figures V

1 Introduction 1
- 1.1 Goal . . . 2
- 1.2 Thesis structure . . . 3

2 Background 4
- 2.1 Kubernetes . . . 4
  - 2.1.1 Kubernetes Objects . . . 5
  - 2.1.2 YAML format . . . 7
- 2.2 LLMs and AI Agents . . . 8
  - 2.2.1 Large Language Models . . . 8
  - 2.2.2 AI Agents . . . 10
- 2.3 Frameworks and Tools . . . 11
  - 2.3.1 LangChain . . . 11
  - 2.3.2 LangGraph . . . 12
  - 2.3.3 LiteLLM . . . 13

3 Related Works 14
- 3.1 Architectural Gaps in Existing Approaches . . . 15

4 Method 17
- 4.1 Design Goals . . . 17
  - 4.1.1 System Requirements . . . 18
- 4.2 System Architecture . . . 19
  - 4.2.1 Control Flow . . . 19
  - 4.2.2 Component Details . . . 21
- 4.3 Prompting Strategy . . . 26

5 Empirical Evaluation 28
5.1 Experimental Setup . . . 28
5.1.1 Dataset Construction . . . 28
5.1.2 LLMs . . . 31
5.2 Intent Analysis . . . 32
5.2.1 Intents Similarity Metrics . . . 32
5.2.2 Analysis of Intent Similarity . . . 33
5.3 Manifest Generation Quality . . . 36
5.3.1 Manifests Similarity Metrics . . . 36
5.3.2 Analysis of Intent Structure Effects . . . 40
5.3.3 Analysis of Temperature Effects . . . 47
5.4 Economic Evaluation . . . 51
5.4.1 Execution Time . . . 51
5.4.2 Token Consumption . . . 55
5.5 Cost-Performance Tradeoff . . . 59
5.5.1 Overall Tradeoff Summary . . . 60

6 Comparative Evaluation 61
6.1 Academic Approaches . . . 61
6.1.1 Fine-tuning Techniques . . . 63
6.1.2 Few-shot Learning Technique . . . 64
6.2 Alternative Frameworks . . . 68
6.2.1 Kagent . . . 68
6.2.2 Comparative Analysis . . . 70

7 Conclusions 75
7.1 Future Works . . . 77

A Model-Specific Prompt Set Used in the Architecture 79

B Intents Similarity Heatmaps 81

C Number of Attempts - Simple vs Structured Intents 83

D Pipeline Details for N-Shot Prompting Generation 84

Bibliography 85

List of Tables

5.1 Benchmark Scenario Categories 30
5.2 Comparative analysis of average model performance under Simple and Structured intents (bolded values marking the highest score per metric) 40
5.3 Number of generations successfully completed for each model and temperature setting, out of a maximum of 20 48
5.4 Average min and maxi cost per million tokens across providers (USD) 58

6.1 Overview of Compared Approaches 62
6.2 Comparison of min and max scores obtained between the fine-tuning approach and our vanilla method 64
6.3 Cost vs. Success Rate Comparison across prompting strategies . . . 67
6.4 Comparison of similarity metrics between manifests generated by kagent and our agent 73

List of Figures

2.1 Kubernetes cluster components 5
2.2 Example of tokenization 8
2.3 Illustration of the three core characteristics defining AI Agents . . . 11
2.4 Differences between LangChain and LangGraph pipelines 12
2.5 Representation of LiteLLM's unified interface 13

4.1 Complete agent's flow 20
4.2 Graphical representation of the state's components and their formats 21
4.3 Overview of the three-step validation process 23
4.4 Simplified example of the process used to construct model-specific prompts 27

5.1 Simplified example of the difference between unstructured and structured intents 35
5.2 Comparison between Simple and Structured prompts for the similarity metrics across the six tested models 43
5.2 (continued) 44
5.3 Comparison between Simple and Structured intents for Precision and Recall across the six tested models 46
5.4 Comparison of the six models across temperatures using different similarity metrics 49
5.5 Distribution of generation attempts across models 52
5.6 Distribution of response time for each model 53
5.7 Comparison between the mean total elapsed time and the mean duration of the Kubernetes validator phase 55
5.8 Distribution of total token consumption for each model 57

6.1 Success Rate Comparison Across Prompting Strategies 65
6.2 Manifest Similarity Comparison Across Prompting Strategies 66
6.3 kagent Architecture 70

C.1 Comparison of number of attempts per model with Simple and Structured Intents across models 83

D.1 Representation of the Pipeline Used to Generate Manifests via N-Shot Prompting Strategies 84

Chapter 1

Introduction

The rapid evolution of cloud technologies and the widespread adoption of distributed architectures have profoundly reshaped how modern applications are developed, deployed, and managed. Among these technologies, Kubernetes has emerged as the de facto standard for container orchestration, enabling organizations to operate complex, large-scale systems with high levels of scalability, resilience, and portability. Its diffusion has not been without challenges: designing, configuring, and maintaining Kubernetes environments requires a deep understanding of numerous concepts, abstractions, and networking principles. In enterprise contexts, this complexity increases dramatically as the number of services, dependencies, and security or operational policies grows.

This complexity represents one of the main challenges facing the modern cloud ecosystem today. Configuration files are not merely static artifacts; they are integral components of the application system. A single syntax error, an incorrect resource definition, or a misinterpretation of best practices can lead to downtime, security vulnerabilities, operational inefficiencies, and significant financial impact. In environments characterized by continuous delivery and close collaboration between development and operations teams, the need for tools that reduce the margin for error and enhance productivity becomes crucial.

Within this landscape, the adoption of AI-driven automation and intelligent configuration-generation systems has gained increasing attention in both academia and industry. Automatically generating Kubernetes manifests and YAML configurations offers the potential to accelerate infrastructure creation, reduce reliance on specialized expertise, and support users with varying levels of experience. However, this approach introduces its own set of challenges: the quality of generated configurations depends on the model's ability to correctly interpret requirements, maintain consistency across components, and produce valid, secure configurations that are compatible with real-world runtime environments.

The motivation for this thesis arises precisely from the need to investigate how automated configuration generation can help address these obstacles by introducing an **autonomous agentic workflow** capable of improving task execution through structured planning, iterative reasoning, tool-mediated operations, and feedback-driven refinement. This perspective shifts the focus from simple code generation to the orchestration of a coherent decision-making process that mirrors real-world engineering practices. The objective is therefore not limited to producing correct configuration files; rather, it aims to evaluate whether intelligent systems can reliably support the definition of complex infrastructures, reducing human error and optimizing deployment workflows.
In this sense, the study lies at the intersection of cloud computing, DevOps, artificial intelligence, and automation—domains that are increasingly intertwined in modern software engineering.

1.1 Goal

The primary scope of this thesis is to investigate an alternative approach to the automatic generation of Kubernetes configurations, analyzing its inherent challenges, the limitations of existing methods, and the opportunities introduced by solutions based on large language models. More specifically, the work aims to design a methodology and propose a prototype system capable of supporting the generation of Kubernetes manifests with the dual goal of reducing human error and improving productivity in the development of cloud-native applications through AI agents. The resulting framework should make the entire configuration-generation process more automatic, reliable, and aligned with real-world operational needs.

The main contributions of this work are:

- **Define an agentic architecture** for intent-driven Kubernetes configuration generation, structured around a multi-step validation pipeline integrating both syntactic checks and deployment-level verifications
- **Integrate an iterative self-correction mechanism** for manifest refinement, autonomously managed by the agent and driven by the errors detected throughout the validation pipeline
- **Discuss the effectiveness and robustness** of the proposed approach, evaluating whether it provides measurable practical benefits compared to existing solutions and practices

This work therefore goes beyond the mere production of YAML manifests. Its purpose is to address the entire lifecycle of configuration creation: interpreting user requirements, generating the corresponding Kubernetes resources, validating correctness, and ensuring consistency across components in a runtime environment.

We demonstrate in this thesis that adopting this holistic perspective leveraging the flexibility introduced by AI agents and the advantages of an error-driven refinement loop, constitutes a significant improvement not only for assessing the technical feasibility of automated configuration generation, but also for enhancing the robustness of the entire workflow and the correctness of the produced configurations.

1.2 Thesis structure

For clarity, this section presents the structure of the thesis, illustrating how each chapter contributes to the overall logical progression of the work.

- **Chapter 2 – Background** introduces the foundational concepts necessary to understand the problem domain. It covers the role of Kubernetes in cloud-native applications, the core principles behind language models and AI agents, and the main tools employed in the proposed solution
- **Chapter 3 – Related Works** surveys existing approaches to automated configuration generation. By reviewing the state of the art, this chapter positions the thesis within the broader context of academic research and industry practices
- **Chapter 4 – Method** presents the core contribution of the thesis. It describes the architecture of the proposed system, the methodological framework, and the integration of language models for Kubernetes manifest generation
- **Chapter 5 – Empirical Evaluation** details the experimental setup, the dataset used for evaluation, and the analysis of the configurations produced by our framework. This chapter compares different models and examines their effectiveness across multiple parameters
- **Chapter 6 – Comparative Evaluation** provides a quantitative comparison between the proposed approach and existing tools or solutions, assessing how the framework performs in a real-world context
- **Chapter 7 – Conclusions** summarizes the key contributions of the thesis, discusses how the objectives have been achieved, and outlines open questions and opportunities for future work in automated configuration generation and cloud-native systems

Chapter 2

Background

2.1 Kubernetes

Kubernetes [1] is an open source container orchestration engine for automating deployment, scaling, and management of containerized applications. The open source project is hosted by the Cloud Native Computing Foundation (CNCF), a non-profit organization that hosts, supports, and oversees open source projects essential for modern cloud computing, such as Helm, Prometheus, Cilium, and many others.

Kubernetes isn't a traditional PaaS platform. It works at the container orchestration layer and offers a set of capabilities that many PaaS systems also provide, but with a fundamentally different philosophy. Instead of delivering a single, monolithic developer experience, Kubernetes supplies modular building blocks that platform teams can assemble into a customized developer platform.

The basic structure of a Kubernetes cluster is composed of two core components:

- ***Master Node***: The control plane of the cluster, responsible for managing its overall state and making scheduling and orchestration decisions. Among its various tasks, it exposes the Kubernetes API, which is accessed through the *kubectl* client used to interact with the cluster.
- ***Worker Node(s)***: The machines that run containerized workloads. They host Pods, provide the Kubernetes runtime environment, and execute the tasks assigned by the control plane.

Every cluster requires at least one control-plane (master) node and one worker node, though production environments typically use multiple control-plane nodes for high availability and many worker nodes for scalability.

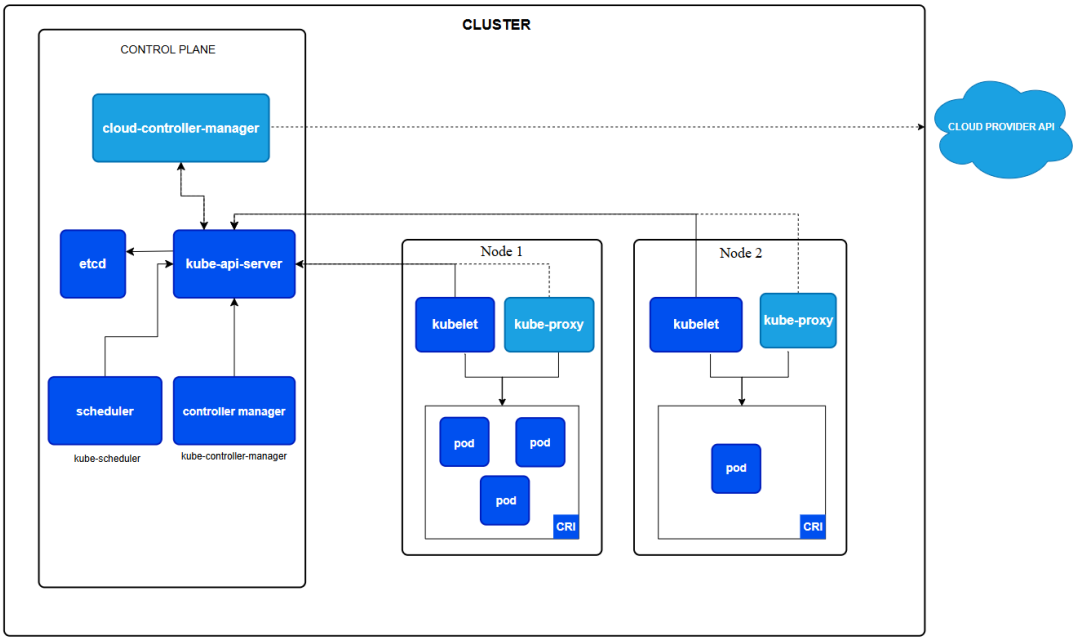


Figure 2.1: Kubernetes cluster components

2.1.1 Kubernetes Objects

Kubernetes objects are persistent resources in the Kubernetes system. Kubernetes uses these entities to represent the desired state of your cluster. Specifically, they can describe what containerized applications should run and on which nodes, the resources available to those applications and the policies that govern their behavior.

Each Kubernetes object is a *"record of intent"*, meaning that once you create or modify an object, the control plane will constantly work to ensure that the actual cluster state corresponds to the desired state you declared. This ongoing reconciliation loop ensures that workloads remain consistent, stable, and aligned with the configuration you specify.

An object contains two nested fields that define its configuration: the *spec* and the *status*. The **status** reflects the object's current state and is continuously populated and updated by the Kubernetes system and its controllers.
When you create an object in Kubernetes, you must provide the object **spec** that describes its desired state.
The way to provide thise information to *kubectl* is through configuration files known as **manifests**, one for each type of object. By convention, manifests are written in YAML format (discussed in Section 2.1.2) or, less commonly, in JSON.

In the manifest file for the Kubernetes object you want to create, you'll need to set values for the following mandatory fields:

- ***apiVersion***: Which version of the Kubernetes API you're using to create this object
- ***kind***: What kind of object you want to create
- ***metadata***: Data that helps uniquely identify the object, including a name string and optional namespace
- ***spec***: What state you desire for the object. The precise format of this field is different for every Kubernetes object, and contains nested fields specific to that object.

Objects are organized into functional categories, each distinguished by a specific *kind* that reflects its role within the cluster. Some examples are:

- ***Workload resources***: Pods, Deployments, StatefulSets, DaemonSets, Jobs, and CronJobs;
- ***Networking resources***: Services, NetworkPolicy and Ingress;
- ***Storage resources***: PersistentVolumes, PersistentVolumeClaims, and StorageClasses;
- ***Configuration resources***: ConfigMaps and Secrets;
- ***Cluster-level resources***: Namespaces, Nodes, and RBAC entities.

This categorization highlights how Kubernetes objects collectively define applications behavior, connectivity, storage and configuration forming the declarative backbone through which the entire system operates. By combining multiple objects, the desired configuration can be assembled to describe workloads of varying complexity, from simple application setups to more articulated deployment patterns.

The smallest deployable unit in Kubernetes is the **Pod**. A Pod represents a logical host for one or more containers that must share resources and an execution context, and its containers are always scheduled on the same node, as shown in the **Figure 2.1**. Although Pods can be created directly, this is not the usual practice. Pods are meant to be ephemeral and disposable, so Kubernetes encourages using higher-level workload resources, such as Deployments, Jobs, or StatefulSets to create and manage Pods automatically. These controllers handle replication, scaling, and self-healing when Pods fail or nodes go down.

To summarize, these files are essential for deploying and managing applications in Kubernetes, but as systems scale, maintaining them becomes increasingly complex. Integrating AI and machine learning, especially LLMs, offers a promising way to improve how manifests are created and managed [2]. By automating repetitive tasks and assisting in configuration optimization, these technologies can simplify operations and enhance the efficiency and reliability of container orchestration.

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: nginx
5    namespace: my-app-ns
6  spec:
7    containers:
8    - name: nginx
9      image: nginx:1.14.2
10     ports:
11     - containerPort: 80
```

Listing 2.1: Example of a simple Kubernetes Pod manifest

2.1.2 YAML format

YAML is a human-friendly data serialization language introduced in 2001 and designed specifically for configuration files and data-exchange scenarios that require frequent manual editing.

Compared to other serialization formats such as JSON and XML, YAML offers several advantages. Its use of indentation to structure data, rather than brackets or tags, significantly enhances readability. YAML also natively supports comments, which are essential in configuration files, and provides anchors and references that allow repeated sections to be defined once and reused.
A single YAML file can also contain multiple documents separated by delimiters ("`---`"). The language supports rich data types such as dates, timestamps, and booleans, and offers flexible mechanisms for multiline strings, which are particularly useful in complex configurations.

As shown in Listing 2.1, the simplest YAML structure is a ***key-value*** pair, defined using a colon followed by a space. This minimal syntax contributes to the overall simplicity and readability of the language, which, as we said, is sensitive to indentation.

In Kubernetes, YAML is the standard format for writing manifests because it allows complex configurations to be represented through a simple and intuitive hierarchical structure. Unlike other serialization formats, it does not require fields to appear in a strict order; what matters is the hierarchy defined by indentation. As long as the nesting of keys is correct, the configuration remains valid, giving users greater flexibility and reducing the likelihood of syntactic errors. This property is particularly important in Kubernetes configuration files, where clarity and structural consistency are essential.

To ensure predictable parsing, Kubernetes adopts a restricted subset of YAML. Features such as anchors, aliases, and advanced scalar typing are intentionally unsupported, and every YAML document is internally converted to JSON before being processed by the API server. This design choice eliminates ambiguities and enforces deterministic interpretation of configuration fields, combining YAML's readability with the strictness required by Kubernetes.

2.2 LLMs and AI Agents

2.2.1 Large Language Models

A Large Language Model [3] (LLM) is a neural network trained on massive amounts of data, enabling it to understand and generate natural language and other content across many tasks. Modern LLMs are built on the transformer architecture, which is designed to process sequences of text and capture complex patterns within them.

LLMs do not interpret text as continuous characters or full words; they operate on **tokens**. A tokenizer transforms the input into these tokens, which are sub-word units drawn from a fixed vocabulary. Depending on the tokenizer, a token may correspond to an entire word, a single character, a fragment of a word, or even punctuation, as illustrated in **Figure 2.2**.

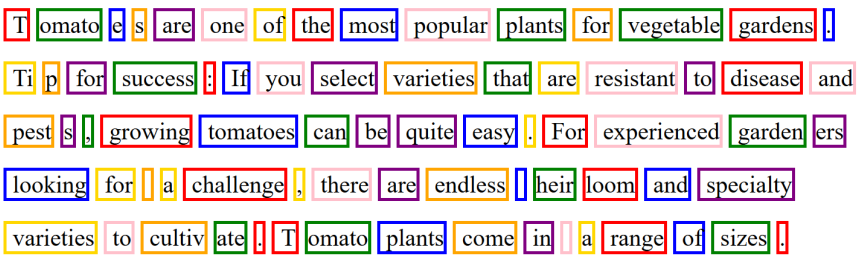


Figure 2.2: Example of tokenization

Talking about tokens matter because they are the fundamental unit of computation for LLMs: pricing is per-token, context windows are measured in tokens, and every prompt is converted into tokens before the model processes it. Longer prompts consume more tokens, cost more, and leave less room for the model's output in the context windows.

To do its job, LLMs function as large statistical prediction machines that repeatedly predict the next word in a sequence (or more precisely the next token). When you send a prompt, the model analyzes all tokens and produces a probability distribution over its vocabulary, estimating which token is most likely to come next. One token is selected, appended to the sequence, and the updated sequence is fed back into the model. This autoregressive loop continues until a stop token appears or a length limit is reached.
The shape of the probability distribution is influenced by a parameter called **temperature**, which controls randomness in generation. A low temperature makes the model more deterministic, favoring the highest-probability tokens. A high temperature increases variability, making the model more willing to choose less probable tokens. Adjusting temperature allows users to balance coherence and creativity when generating output depending on the specific use case.

Traditional models map inputs to outputs directly, relying on fast pattern recognition to generate responses. In contrast, other models try to mimic human logic by actively "thinking" before answering. This process is called **reasoning**, which means decomposing complex problems, exploring alternative solution paths, and internally correcting errors through a chain-of-thought process. This capability emerges from the same autoregressive prediction process described above, where the model integrates contextual information and evaluates many possible continuations before selecting the next token. Despite their many strengths and benefits, reasoning LLMs also introduce notable drawbacks. They are prone to overthinking, often generating far more tokens than necessary, on average 1,953% more than standard models [4] to reach the same answer. This internal reasoning substantially increases cost and latency, and in many practical scenarios the overhead introduced by this "thinking" phase is far from negligible.

In recent years, LLMs have shown strong capabilities in natural language understanding and in generating structured or complex outputs such as code and configuration files. However, direct generation is often insufficient, as outputs must satisfy domain-specific rules and constraints that the model does not inherently enforce. LLMs also rely on patterns learned from training data and remain limited by knowledge gaps and reasoning constraints, making it challenging to produce valid, context-aware artifacts in specialized domains. These limitations have motivated the development of agentic AI approaches.

2.2.2 AI Agents

An **AI Agent** is a system that autonomously performs tasks by designing workflows with available tools [5]. Unlike deterministic automation scripts that follow fixed procedures, AI Agents demonstrate reactive intelligence and some level of adaptability, allowing them to interpret dynamic inputs and reconfigure outputs accordingly. At the core of AI agents there are still LLMs, but these systems extend their capabilities by using tools in the background to access up-to-date information, optimize workflows, and break tasks into smaller steps to achieve complex goals.

While LLMs primarily function as reactive prompt followers, AI Agents operate automatically within explicitly defined scopes, engaging dynamically with inputs and producing actionable outputs in real-time environments.
In this process, the agent learns to adapt to the user's need over the time by storing past interactions and planning future actions, creating a more personalized and coherent experience. Another advantage of this approach is that tool calling can be achieved without human intervention and can expand the range of real-world applications these AI systems can support.

The three foundational characteristics of AI Agents [6] are :

- ***Autonomy***: agents' ability to operate with minimal or no human intervention after deployment. Once initialized, agents can perceive environmental inputs, reason over contextual data, and execute predefined or adaptive actions in real time. In practical terms, *autonomy* does not imply unrestricted or fully independent decision-making. Instead, it refers to an agent's ability to operate within a predefined action space and execute decisions without requiring continuous human supervision. An agent can interpret environmental inputs, select appropriate tools, and adjust its workflow based on intermediate results, all operating within explicit constraints that ensure its behavior remains predictable and controllable.

- ***Task-Specificity***: agents are optimized to perform repeatable operations within a fixed well-defined domain. This specialization allows for efficiency, interpretability and high precision in automating tasks where general-purpose reasoning would be less effective.

- ***Reactivity and Adaptation***: agents use feedback mechanisms, including interactions with other agents or human-in-the-loop (HITL) corrections, to refine their reasoning and improve output quality. These mechanisms, which is commonly referred to as *iterative refinement* [7], improve the AI agent's reasoning and accuracy. Through this continuous refinement, the agent can maintain the level of autonomy discussed previously.

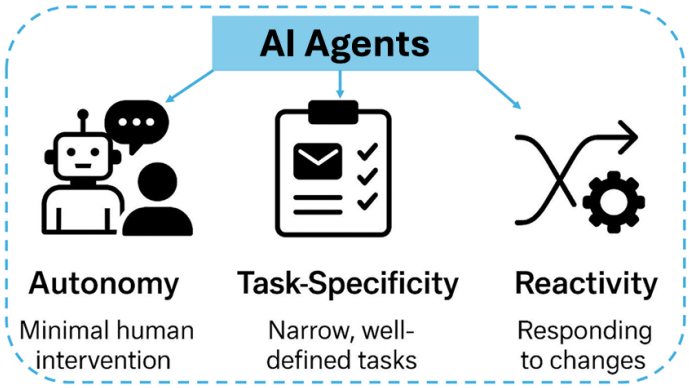


Figure 2.3: Illustration of the three core characteristics defining AI Agents

These core characteristics collectively enable AI Agents to serve as modular, lightweight interfaces between pretrained AI models and domain-specific utility pipelines. Their architectural simplicity and operational efficiency position them as key enablers of scalable automation across enterprise and industrial settings. Their high usability and performance within constrained task boundaries have made them foundational components in contemporary intelligent system design.

2.3 Frameworks and Tools

The development of AI Agents and LLM systems relies not only on model capabilities but also on the surrounding infrastructure. Specialized frameworks provide abstractions for workflows, tool orchestration, state management, and safe interaction with external environments. These systems serve as the operational backbone of agent systems, allowing to integrate pretrained models into reliable pipelines, enforce constraints, and scale autonomous behaviors across real-world applications. The following section outlines the primary frameworks and tools used in this work.

2.3.1 LangChain

LangChain [8] is an open-source orchestration framework designed to streamline the development of applications built on LLMs. Available in both Python and JavaScript, it offers a rich ecosystem of modular abstractions—such as chains, prompts, models, memory, and tool wrappers—that encapsulate common patterns in LLM-driven workflows. These components act as building blocks that developers can combine to construct complex natural language processing pipelines.

At its core, LangChain enables **multi-step reasoning** by allowing developers to *"chain"* together multiple operations. This design lowers the barrier to entry for rapid prototyping: newcomers can assemble functional applications without deep expertise in model internals, while experienced developers benefit from a structured framework that accelerates experimentation. A particularly powerful feature is LangChain's support for autonomous agents, which integrate conversational LLMs with external tools, such as search engines, APIs, and databases, to perform iterative decision-making.

2.3.2 LangGraph

LangGraph is a framework built on top of LangChain by introducing a **graph-structured** paradigm for designing and controlling LLM-based workflows. Whereas LangChain emphasizes modular components and sequential chains, LangGraph models an application as a directed graph whose nodes represent computational steps—LLM calls, tool invocations, memory updates—and whose edges define transitions between states. This structure supports cyclic, branching, and multi-agent workflows that go beyond the linear pipelines typical of LangChain.

A defining feature of LangGraph is its *state* management system, which maintains persistent internal variables across steps. This enables advanced behaviors such as iterative planning, tool-use loops, and collaborative multi-agent interactions. The graph-based execution model also provides fine-grained control: developers can pause, resume, replay or inspect intermediate states, making debugging and monitoring more transparent. LangGraph integrates seamlessly with LangChain's existing components and together form a complementary ecosystem: LangChain supplies flexible abstractions for building LLM workflows, while LangGraph provides the structural rigor needed for stateful, reproducible, and controllable execution.

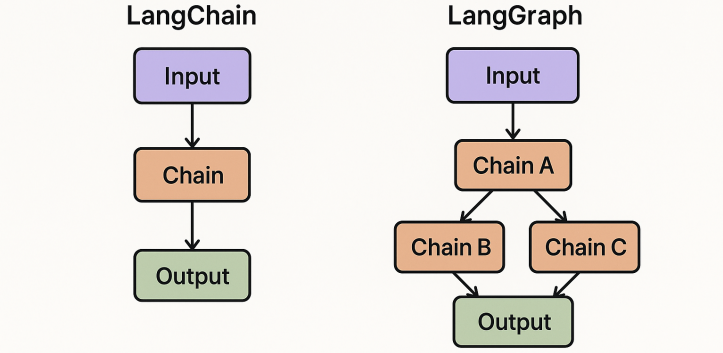


Figure 2.4: Differences between LangChain and LangGraph pipelines

2.3.3 LiteLLM

LiteLLM [9] is a lightweight, open-source library that provides a unified interface for interacting with large language models across diverse providers. Its core purpose is to standardize the heterogeneous APIs of commercial and open-source LLMs under a single function signature modeled after the OpenAI API. This abstraction allows developers to switch between models or providers with minimal code changes, greatly simplifying experimentation, integration, and deployment.

A major advantage of LiteLLM is its ability to support both cloud-based and locally hosted models. Beyond commercial APIs, it can connect to local runtimes such as Ollama, enabling models to run entirely on a developer's machine or on self-managed servers. By exposing local and remote models through the same interface, LiteLLM ensures seamless transitions across development, testing, and production environments.

LiteLLM can be used directly within Python applications or deployed as a proxy server, allowing organizations to centralize model access, enforce authentication and usage policies, and maintain consistent interfaces across teams. This unified model-invocation layer integrates naturally with higher-level abstractions such as LangChain and LangGraph, which can rely on LiteLLM to handle model calls in an interchangeable and provider-agnostic way. By separating model management from workflow logic, developers can build complex reasoning pipelines or agent-based systems while retaining full flexibility in choosing, combining, or switching the underlying LLM providers.

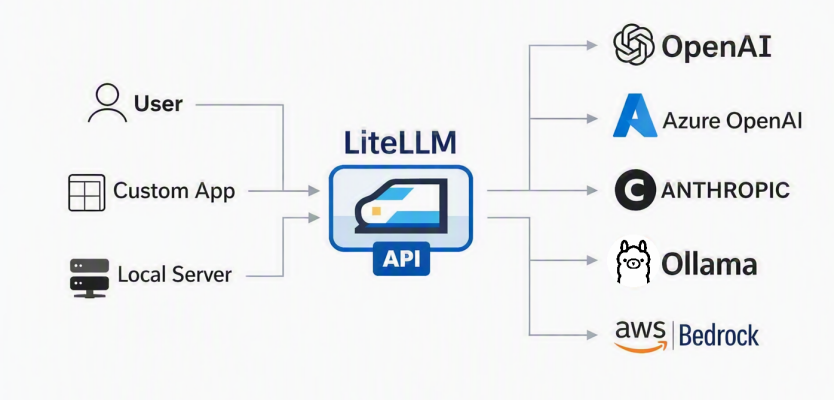


Figure 2.5: Representation of LiteLLM's unified interface

Chapter 3

Related Works

The growing field of autonomous AI agents powered by large language models has stimulated extensive research across multiple domains. In this section, we review the most relevant studies to clarify the foundations from which the need for our proposed approach emerges.

Many studies have explored the use of LLMs for code generation in popular programming languages. However, their performance in Domain-Specific Languages (DSLs)—specialized languages designed to express operations, constraints, or configurations within a narrow technical domain—remains a critical challenge. This is mainly due to severe data scarcity and the highly specialized syntax and semantics that are poorly represented in general-purpose datasets [10] [11] [12].

The rise of Generative AI is progressively transforming the automation of complex tasks across multiple domains, including network management, with a particularly strong potential impact on cloud computing operations [13]. Within this context, several works investigate Intent-Based Networking (IBN) [14], a paradigm that enables seamless intent-to-program translation and addresses the limitations of traditional network management, which relies heavily on manual coding and specialized expertise. By abstracting low-level configurations, IBN supports both non-specialists and experienced operators in managing the growing complexity of modern networks.

Other research efforts focus on the automatic generation of structured cloud configurations used to orchestrate and automate service deployment across distributed infrastructures. These configuration models have shown seamless integration within containerized environments, microservices architectures, and Kubernetes-based clusters. A particularly relevant line of work examines YAML files, the de facto standard adopted by numerous cloud-native tools such as Ansible [15], GitHub Actions [16] and microservices in general [17]. Existing approaches typically rely on one-shot or few-shot prompting, often combined with handcrafted datasets and

prompt rephrasing [18] to mitigate the scarcity of domain-specific examples.

In the specific case of Kubernetes configuration, KGen [19] represents an intent-based system that trains an LLM using few-shot examples to generate Kubernetes manifests from natural-language intents.

Other experiments on open-source LLMs demonstrate that intent-based K8s manifest generation can be achieved through model fine-tuning, while also highlighting the open challenges that must be addressed before achieving autonomous and self-healing Kubernetes infrastructures managed through Agentic AI. In particular, research shows that even powerful LLMs struggle with data manipulation and tend to overthink, especially when processes involve long chains of reasoning with self-reflection. These findings suggest that fully autonomous LLM-driven cloud control remains premature at this stage [20].

More recently, Beuter et al. incorporated Kubernetes error reports into an iterative refinement workflow, demonstrating that feedback-aware generation can substantially improve deployment success rates [2]. These findings motivate the use of agentic architectures capable of autonomously incorporating operational feedback.

Our work extends this direction by framing manifest generation as an agentic task that integrates planning, validation, reasoning, and correction within a unified execution workflow.

3.1 Architectural Gaps in Existing Approaches

Current research on the automatic generation of Kubernetes configuration manifests reveals several recurring limitations that must be addressed to achieve the broader objectives of this thesis:

- **Lack of an End-to-End Framework**
  Although existing studies demonstrate that LLMs can produce Kubernetes manifests from natural-language intent, most proposals remain proof-of-concept prototypes. They lack a unified, plug-and-play architecture capable of handling the full pipeline—from intent interpretation to manifest generation, correction, and consistency enforcement. As a result, these systems cannot be directly adopted in real-world workflows.
- **Validation as an External Evaluation Step**
  Validation is typically performed only after generation and used exclusively to assess model performance. Current architectures rarely integrate schema-aware, in-loop validation mechanisms that iteratively refine the output, correct structural or semantic errors, and ensure compliance with Kubernetes specifications and best practices.

- **Absence of Security Considerations**
  Security is almost entirely absent from existing proposals. They do not incorporate checks for common misconfigurations, unsafe defaults, or violations of Kubernetes security guidelines. This omission significantly limits their applicability in environments where secure configuration is non-negotiable.
- **Limited Support for Complex Scenarios**
  Most works focus on simple, single-resource manifests, avoiding the complexity of multi-object configurations, inter-resource dependencies, or full application stacks. This narrow scope prevents current approaches from addressing realistic Kubernetes use cases.
- **Attention to Methodological Robustness**
  A further limitation concerns the emphasis placed almost exclusively on output quality rather than on the robustness of the generation process itself. While high-quality manifests are essential, the operational context—cloud-native environments that demand reliability, resilience, and predictable behavior—requires frameworks that remain stable across diverse inputs, edge cases, and failure modes.

Overall, the literature confirms the feasibility of using LLMs to generate Kubernetes manifests but stops short of delivering a complete, validated, secure, robust, and extensible framework. These gaps point toward a promising architectural direction: a cohesive system that integrates intent understanding, iterative refinement, schema-aware validation, security checks, and multi-object generation into a unified workflow.

Chapter 4

Method

Having outlined the current state of the literature and identified its main limitations, we now shift to the proposed solution and the architectural principles that guide this work. This chapter presents how the system addresses the gaps previously discussed and provides a detailed description of the designed framework.

The following sections introduce the design goals that guide the solution, describe the core architectural components that compose the framework, and provide additional implementation details, including the prompting strategy adopted within the system and the principles behind its construction.

4.1 Design Goals

The proposed system is conceived as a controlled generation-and-validation framework for Kubernetes manifests. Rather than relying on a single, unconstrained LLM invocation, the architecture decomposes the task into a sequence of specialized stages, each responsible for validating a specific aspect of the output.

At the core of the approach is the idea of treating YAML generation as an iterative, stateful process. The agent receives a natural-language task, produces an initial manifest, evaluates it through multiple independent checks, and uses the resulting feedback to refine the output when necessary. Correctness is therefore not assumed at the first attempt; instead, it is progressively enforced through explicit validation steps.

A final objective of the implementation is to provide a ready-to-use framework accessible through an API, enabling seamless integration into more complex systems that require the generation of Kubernetes manifests of varying complexity. This ensures that the solution is not only methodologically robust but also practical and easily adoptable in real-world workflows.

4.1.1 System Requirements

To guide the detailed architectural description that follows, we summarize the system's purpose and requirements, organizing them into three complementary pillars to clarify the upcoming design choices.

Capability Requirements

- **Natural-language Task interpretation**
  The agent must translate a natural-language request into a Kubernetes manifest while ensuring the prompt is within the supported scope.
- **Multi-stage validation**
  Every generated manifest must undergo syntactic validation (YAML correctness), Kubernetes validation (cluster applicability), and semantic validation (alignment with the original task).
- **Iterative refinement**
  When a validation step fails, the agent must incorporate the corresponding feedback and produce an improved version of the manifest without regenerating it from scratch.

Quality and Reliability Requirements

- **Robustness**
  The agent must remain stable across different models and failure modes, handling invalid outputs gracefully triggering controlled responses.
- **Correctness-by-construction**
  The architecture must minimize the risk of accepting incorrect manifests by enforcing validation at multiple stages instead of relying on a single check.

Architectural Requirements

- **Modularity**
  Each stage of the pipeline must be implemented as an independent component to simplify maintenance and evolution.
- **Extensibility**
  The architecture must support different LLM backends and prompting configurations through externalized settings.
- **Traceability**
  The system must record validation outcomes and feedback to make the decision process reconstructible.

4.2 System Architecture

The proposed architecture is implemented as a LangGraph-based state machine (**Section 2.3.2**), responsible for coordinating the interactions among the language model, the validation tools, and the feedback loop. This design enables a high degree of flexibility and fine-grained customization, ensuring that the agent's behavior can be adapted to diverse operational requirements.
We now examine in detail the internal structure of the agent and the role of each architectural component.

4.2.1 Control Flow

To provide a clearer understanding of the agent's behavior, we begin with an overview of its execution flow. The complete control flow is shown in **Figure 4.1**, which illustrates the linear progression through the pipeline of nodes, each responsible for a specific processing stage. Dashed edges represent conditional transitions triggered by errors: when a validation step fails, the agent either terminates the execution or re-enters the generation loop to refine the output.

The process starts with the user request, expressed in natural language. This initial input constitutes the intent, which captures the user's needs. The first node performs a **scope consistency check**, ensuring that the intent indeed concerns the generation of Kubernetes manifest configurations. If the request falls outside this scope, the pipeline terminates immediately. Otherwise, the validated intent is forwarded to the generator node, the core component of the agent, where the initial manifest is produced.

The generated manifest then undergoes a **syntax validation**, which verifies that the output is a well-formed YAML file and does not contain other formats. If a syntax error is detected, the pipeline triggers a backward transition to the generator node, prompting a corrected regeneration. Once the syntax is validated, the manifest proceeds to the **operational validation**, which checks whether the YAML is structurally valid and operationally compatible with a Kubernetes cluster. Errors at this stage follow the same refinement mechanism.

If all previous validations succeed, the manifest reaches the **semantic consistency check**, which evaluates whether the generated configuration satisfies the requirements expressed in the intent. Any detected inconsistency triggers another refinement cycle.

The pipeline terminates successfully only when all validation nodes are passed without errors. Conversely, if the refinement loop fails to converge to a correct solution, specifically after seven unsuccessful regeneration attempts, the process is halted and the agent returns an error.

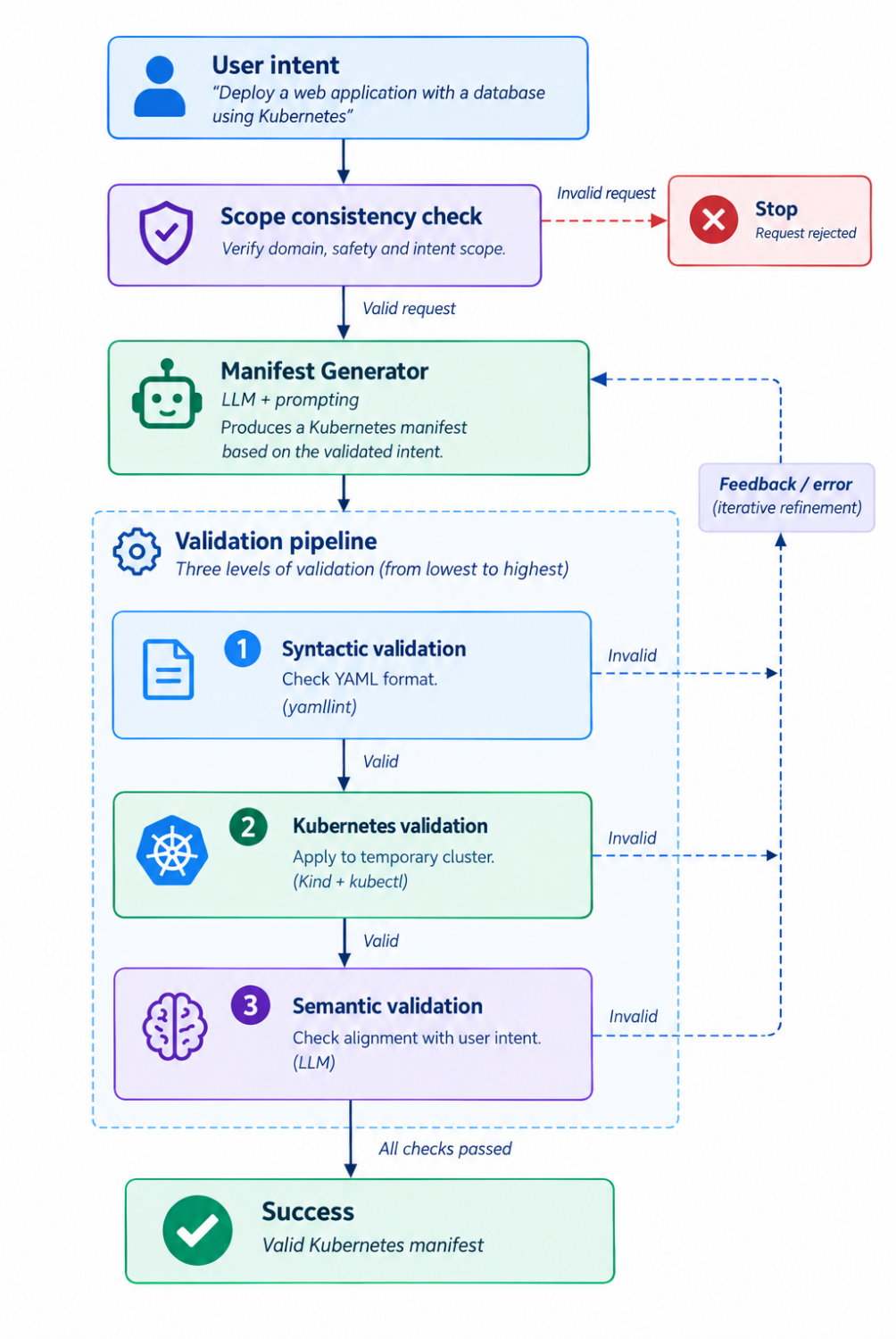


Figure 4.1: Complete agent's flow

4.2.2 Component Details

We now examine the architectural components one by one, in detail, to clarify how the functionalities described above are implemented.

Agent State As previously discussed, the agent operates as a LangGraph-based state machine, which requires a structured mechanism to store and share information across all nodes involved in the generation and validation pipeline. This structure is called `AgentState`, a structured dictionary that represents the shared execution context of the entire generation process and maintains all information needed to coordinate the iterative refinement loop.

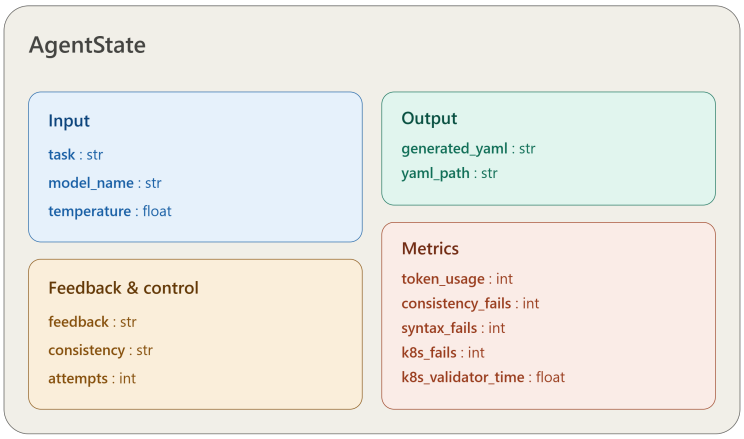


Figure 4.2: Graphical representation of the state's components and their formats

The `AgentState` (**Figure 4.2**) organizes all the relevant information into four main conceptual categories :

- **Input information** The original user task, the selected model, and its temperature. These parameters define the initial conditions of the generation process and the information to asset the generation.
- **Generation outputs** The latest YAML manifest produced by the agent and, when applicable, the path of the file written to disk.
- **Control variables for the iterative loop** The most recent validation feedback, a flag associated with prompt-consistency failures, and the number of attempts performed. These elements coordinate the refinement cycle and determine whether further iterations are required.

- **Execution-tracking metadata** Error counters for each validation stage (syntactic, Kubernetes, semantic), timing information, and token-usage statistics. These attributes support evaluation, debugging, and reproducibility.

Together, these components provide a complete view of the agent's internal current execution flow, enabling controlled iteration, structured validation, and transparent traceability across the entire pipeline.

Generation Node The `generator_node` is the core component of the system. Its role is to produce new Kubernetes manifests, or repair existing ones, based on the user's request.
At the beginning of each run, the node initializes the LLM instance using the model name and temperature stored in the shared state. This initialization is performed through *LiteLLM*(details in **Section 2.3.3**).

Once the model is instantiated, the node constructs the prompt for the current execution, which consists of:

- **System Message** Contains the specification for the generation task associated with the selected model. It defines the model's responsibilities and constraints (see **Section 4.3** for details about prompts construction)
- **Human Message** Contains the *Task*, i.e., the original intent expressed by the user, passed verbatim

This combined prompt is sent to the LLM to generate the manifest.

If the system is in a repair phase—meaning a previous iteration produced an invalid manifest and the *feedback* field in the state is populated—the prompt is extended with additional elements:

- **YAML to correct** The manifest generated in the previous iteration
- **Error to fix** The validation error detected in that YAML file
- **Repair instructions** Explicit guidance instructing the model to preserve valid sections and modify only the parts that violate constraints. This addition proved essential in our experiments, enabling the LLM to perform precise, minimal corrections rather than regenerating the entire configuration

This design encourages targeted, incremental fixes instead of full regeneration, improving stability and reducing unnecessary changes.

Once the LLM returns its output, the node extracts the generated YAML and stores it in the shared state together with updated execution metrics: the *yaml_path*, the incremented *attempts* counter, and the updated *token_usage*. The workflow then proceeds to the next node for validation.

Safety pre-filter The first stage of the pipeline, as shown in the flow description, is the `scope_consistency_node`, a preliminary safety gate. This node introduces an first layer of protection within the framework: it ensures that the user's request is genuinely related to the generation of Kubernetes manifests and not to unrelated tasks such as generic programming, arbitrary code generation, or requests that fall outside the intended scope of the system. It also verifies that the input does not contain malicious intent, such as requests to generate malware, ransomware, offensive payloads, or any harmful artifact.

The node performs this check through an LLM call configured with a dedicated prompt tailored to this purpose. The prompt includes a detailed description of what constitutes a safe, valid, and rejectable request. The message passed to the LLM follows the same structure used in the generator node.

If the LLM determines that the request is invalid, the node acts as a first-line filter: it stops the pipeline immediately, preventing the request from entering the generation and validation stages. If the request passes this gate, the workflow continues normally toward the generation and validation phases.

As with the other nodes, the system updates the token usage metrics and records the outcome of this stage in the *consistency* field of the shared state.

Validation Nodes The validation layer is composed of three specialized nodes, each responsible for checking a different level of correctness of the generated manifest. This modular design is essential because a Kubernetes YAML file may be syntactically valid yet still fail at deployment time, while a deployment may succeed without fully satisfying the original task. For this reason, the agent does not rely on a single acceptance criterion, but on a staged validation pipeline that progressively narrows the space of possible errors.

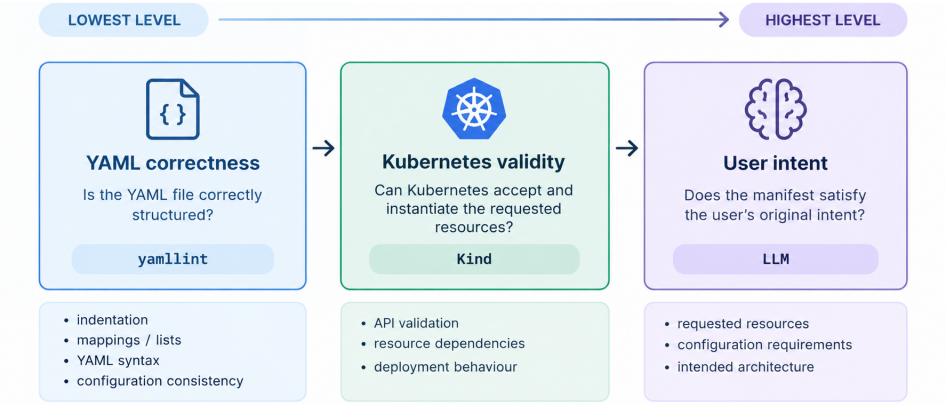


Figure 4.3: Overview of the three-step validation process

The three validation steps are conceptually executed sequentially, progressing from the lowest level of correctness to the highest, as illustrated in **Figure 4.3**. The following sections describe each component in detail.

The `syntax_validator_node` is the first validation step. Its purpose is to verify that the generated manifest is a well-formed YAML before it is passed to the Kubernetes runtime.
In our implemented architecture, this step is performed through `yamllint`, a tool that parses the file and reports **structural issues** such as indentation errors, malformed mappings, invalid list syntax, and other configuration inconsistencies. We adapted the default yamllint configuration to better suit our use case—for example, disabling certain bracket-related checks and adjusting indentation rules to correctly handle Kubernetes-style manifests.

If yamllint returns a successful exit code, the manifest is considered syntactically valid, and the workflow proceeds to the next validation stage. If instead the command reports errors, the lint output is extracted and stored as *feedback* for the next generation attempt.

This node acts as a fast, deterministic filter: it validates the output at the lowest level of correctness and provides actionable error messages that can be directly fed back to the generator. By catching malformed files early, the system avoids the computational cost of performing deployment-level validation on manifests that are not even syntactically valid.

The `kubernetes_validator_node` evaluates whether the generated manifest can be successfully applied to a running Kubernetes cluster. In our implementation, this validation is performed on a *Kind* cluster provisioned through the project's utilities. The system applies the generated YAML directly to this environment, ensuring that the manifest is accepted by the Kubernetes API and that the described resources can be instantiated under realistic runtime conditions.

To guarantee isolation and reproducibility, the test environment is implemented as a temporary, self-contained Kind cluster created specifically for each generation attempt. Instead of relying on a shared or persistent deployment environment, the system instantiates a fresh local cluster composed of a lightweight control plane and a single worker node—providing a minimal yet fully functional Kubernetes runtime. The cluster is managed within a controlled lifecycle: it is created at the beginning of each validation, used exclusively for that attempt, and destroyed immediately after the result is recorded. This approach ensures a clean, deterministic environment for every test and prevents interference between iterations.

We deliberately chose to perform a full application using `kubectl apply` instead of relying on `kubectl apply --dry-run`. Although dry-run mode is faster and often sufficient for single-file manifests, it exhibits inconsistencies when validating multi-file configurations. In particular, dry-run cannot always detect whether cross-resource references are valid—for example, whether selectors match existing labels or whether dependent objects are correctly defined. For our use case, this limitation makes dry-run inadequate as a standalone validation mechanism.

If the apply operation succeeds, the manifest is considered valid from a **deployment perspective**, and the workflow advances to the semantic consistency stage. If the operation fails, the validator captures the command output and stores it as *feedback*, enabling the generator to refine the resource specification in the next iteration.

The `semantic_consistency_node` is the final node in the validation phase and the last node of the entire pipeline. It evaluates whether the generated YAML actually satisfies the intent expressed by the user. Unlike syntax and deployment validation, this stage is not purely mechanical: it verifies whether the manifest is aligned with the task requirements and whether the generated resources correctly represent the requested scenario. In the current implementation, this evaluation is performed by invoking the LLM with both the original task and the generated manifest. A specialized prompt guides the model through the evaluation by specifying the aspects that must be checked and the elements of the user's intent that should be taken into account. The LLM is then asked to determine whether the generated manifest is semantically valid to terminate the workflow. Otherwise, the generated feedback is returned to the generator node, which uses it to refine the manifest in the next generation attempt.

This node addresses a key limitation of purely syntactic or runtime validation: a manifest may be valid YAML and deployable, yet still not represent the intended architecture. By explicitly verifying semantic alignment, the system reduces the risk of false positives where the generated configuration satisfies technical constraints but fails to respect the user's specification.

External Access Layer Beyond the internal agent architecture, the system includes an external access layer implemented through a **FastAPI** service. This component exposes the generation-and-validation pipeline as a callable API, enabling seamless integration into external tools and applications. The API acts as a thin interface over the LangGraph agent: it receives a natural-language task, initializes the agent state, triggers the full generation workflow, and returns the final validated manifest together with execution metadata.

This layer transforms the agent from a standalone research prototype into a reusable component that can be embedded in CI/CD pipelines, developer tools, or higher-level orchestration systems.

4.3 Prompting Strategy

The design of the prompting strategy is a central element of the proposed architecture. Prompts determine how the LLM interprets the user request, how it generates Kubernetes manifests, and how it evaluates their correctness throughout the validation pipeline. For this reason, the framework adopts a structured and modular prompting strategy that constrains the model's behavior and aligns it with the objectives of each node in the workflow.

Modular Prompting Rather than relying on a single monolithic prompt, the system decomposes the workflow into specialized nodes, each with its own clearly defined objective. Every stage that involves an LLM call uses a dedicated prompt that constrains the model's behavior and aligns it with the node's objective:

- **Scope consistency Prompt**: Defines the boundaries of the agent's domain. It instructs the model to determine whether the user request falls within the scope of Kubernetes manifest generation, providing examples of acceptable and unacceptable tasks (including malicious and unrelated requests).
- **Generator Prompt**: Guides the model toward producing a manifest that satisfies the user's intent. It explicitly instructs the LLM to output only the YAML—without explanations, comments, or formatting fences—and to adhere strictly to Kubernetes conventions.
- **Semantic Consistency Prompt**: Specifies how the model must assess the alignment between the generated manifest and the original user request. It focuses on relevance, correctness, and adherence to the intended architecture.

This modular design ensures that each node operates with a prompt tailored to its role, improving reliability and reducing ambiguity.

Model-Specific Prompting A key design choice is to adapt prompts to the specific LLM used, rather than assuming that all models respond equally well to the same instructions. Instead of relying on a single universal prompt, for all LLMs, we first defines a canonical version of each prompt type, essentially a blueprint describing the basic constraints and behaviors required. Then from this baseline, we generate using an LLM a customized version of the prompts that rewrites the canonical prompt into a version aligned with that model's preferred phrasing, structural patterns, and interpretive strengths.

This reconstruction ensures that every model receives instructions in a form it can process most effectively. As a result, prompts become tailored to the model's linguistic tendencies, enabling more reliable behavior at each stage of the workflow and improving the model's ability to understand and execute its assigned task.

These customized prompts are stored externally, ensuring clean separation between agent logic and prompt configuration.

This approach offers several advantages:

- **Portability**: Prompts can be adapted to new models without modifying the core logic
- **Reproducibility**: Prompt configurations are externalized, versionable and not hard-coded into the pipeline
- **Interoperability**: The same architecture can operate consistently across different LLMs only swapping the prompt
- **Performance**: Each model receives instructions in the style it responds to best.

The prompting strategy is therefore a foundational component of the system. It enforces generation constraints, ensures consistent behavior across different models, and enables the feedback-driven repair loop that characterizes the architecture.

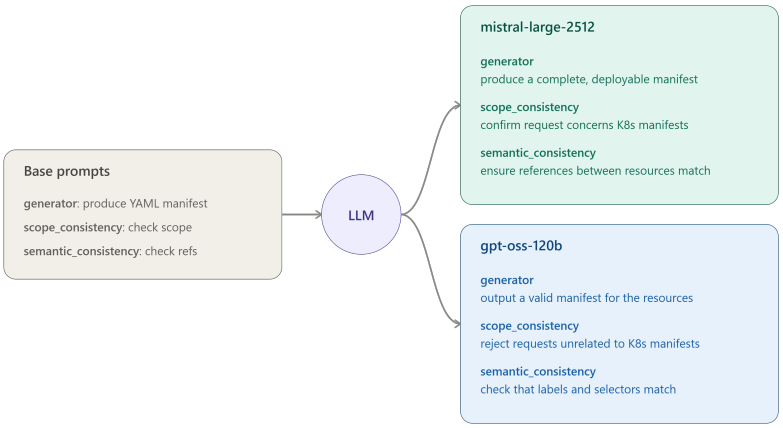


Figure 4.4: Simplified example of the process used to construct model-specific prompts

A complete example of the prompts used in our architecture is provided in **Appendix A**, offering a clearer view of the constraints we adopted.

Chapter 5

Empirical Evaluation

In this chapter, we present the results obtained from testing our agent and use them to assess its effectiveness and practical applicability.

We begin by describing the setup of our evaluation, followed by an analysis of the intents used for testing, with particular attention to their structure and mutual similarity. We then examine the similarity scores produced during direct manifest comparison, discussing how these scores relate to the intents used in the evaluation. Finally, we evaluate the economic efficiency of the different models employed, comparing their performance with their overall cost sustainability.

5.1 Experimental Setup

All our subsequent tests are designed to stress the agent's capabilities and to determine whether its performance meets our expectations, with the ultimate goal of assessing both the syntactic quality, operational validity of the generated YAML configurations and robustness of the agent. In this section, we describe how the dataset used in the evaluation was constructed and specify the models employed in the experiments.

5.1.1 Dataset Construction

The evaluation of the proposed agent is based on a benchmark dataset specifically constructed to represent a diverse set of cloud-native deployment scenarios. The main objective of the dataset is to evaluate the ability of the agent to translate natural-language requirements into complete Kubernetes configurations, covering different levels of architectural complexity. The scenarios range from simple single-service deployments to more articulated applications involving multiple Kubernetes resources.

Each scenario is defined by two complementary elements: a set of **Kubernetes manifests** describing the target configuration and a **natural-language intent** describing the requirements that the configuration is expected to satisfy. This structure allows the generation task to be evaluated independently from the original manifest: during the experiments, the agent is provided only with the natural-language intent and is required to generate the corresponding Kubernetes configuration without access to the reference manifests. The latter are instead used as part of the evaluation process to assess whether the generated configuration correctly addresses the intended scenario.

The construction of the dataset presented a non-trivial challenge due to the nature of Kubernetes manifests themselves. Unlike more general forms of source code or configuration files, Kubernetes manifests are usually created for specific deployment environments and are primarily used for internal cluster orchestration. As a result, complete and well-structured examples covering realistic application scenarios are relatively uncommon in publicly available repositories.

To address these limitations, the dataset was manually constructed, scenario by scenario, using publicly available Kubernetes configurations as sources of inspiration. The collected examples were analyzed and adapted to define self-contained scenarios that could be described through clear and reproducible natural-language intents. Since the reference configurations originate from heterogeneous sources and may therefore reflect different implementation choices or conventions, these variations are not considered critical for the purpose of the benchmark. The evaluation focuses instead on whether the generated configuration provides the Kubernetes resources and relationships required by the corresponding intent.

The resulting dataset consists of **ten scenarios**, each organized around a distinct deployment theme and represented by more Kubernetes manifest files. The scenarios were selected to provide a progressive increase in complexity, both in terms of number of resources involved and in terms of the relationships between them. Simple scenarios primarily focus on individual Kubernetes capabilities, while more complex ones require the agent to coordinate multiple resources and satisfy several requirements simultaneously. This progression makes it possible to evaluate not only whether the agent can generate valid Kubernetes configurations, but also how its performance changes as the complexity of the requested deployment increases.

Table 5.1 summarizes the composition of the benchmark according to the main functionality represented by each scenario and its associated complexity level.

Category	# Intents	Complexity
Simple Deployments	1	Low
Persistent Storage	2	Low
Autoscaling Scenarios	1	Low
Environment Isolation	1	Medium
Ingress Scenarios	1	Medium
Scheduled Database Operations	1	Medium
Multi-Tier Apps	2	High
Production-Like Workloads	1	High
Total	**10**	

Table 5.1: Benchmark Scenario Categories

The *Low* complexity category contains scenarios focused on a limited number of Kubernetes features, such as basic deployments, persistent storage, and autoscaling. The *Medium* category introduces additional interactions between resources, including environment isolation, ingress configuration, and scheduled database operations. Finally, the *High* complexity category contains multi-tier and production-like workloads, where several resources and configuration requirements must be combined to obtain the desired architecture.

The complete dataset is publicly available in a dedicated *Hugging Face repository*[1]. The repository contains the Kubernetes manifests associated with each scenario together with their corresponding natural-language intents, making the benchmark reproducible and allowing the dataset to be used for further evaluation or comparison.

[1]Elisa Bariselli, *kubenetes-configurations* (2026) Available at: `https://huggingface.co/datasets/elisa0205/kubernetes-configurations`. DOI: 10.57967/hf/10308

5.1.2 LLMs

For our experiments, we selected a heterogeneous set of general-purpose language models differing in size, architecture, and reasoning capabilities. This diversity allows us to observe how the agent behaves under varying computational conditions and to investigate whether performance trends correlate with model scale, instruction-following ability, or domain-specific robustness.

The models included in our evaluation are:

- **`granite-4-h-small`** reasoning, 32B
- **`llama-3-3-70b-instruct`** reasoning, 70B
- **`llama-4-maverick-17b-128e-instruct-fp8`** reasoning, 400B (17B active)
- **`gpt-oss-120b`** reasoning not enabled, 117B (5.1B active)
- **`mistral-small-3-1-24b-instruct-2503`** non-reasoning, 24B
- **`mistral-large-2512`** non-reasoning, 675B (41B active)

Beyond architectural differences, an important aspect of this selection is the presence of both reasoning-enabled and non-reasoning models. This distinction is relevant because reasoning-augmented models typically apply additional internal steps—such as structured planning or chain-of-thought compression—that may influence the accuracy and consistency of generated Kubernetes manifests. Including both categories allows us to assess whether such mechanisms provide measurable benefits in tasks requiring structured declarative output.
In next tests, all models were evaluated using the same prompt design and the same validation protocol to ensure comparability across the entire test suite.

To facilitate full reproducibility of our results, the complete experimental pipeline—including the agent implementation, prompt set and evaluation scripts—is publicly available in a dedicated **GitHub repository**[2]. The repository contains all resources required to replicate the experiments presented in this chapter, including instructions for environment setup and execution.

[2]Elisa Bariselli, Github Repository *Agent4net-intent* (2026) Available at: `https://github.com/elisa0205/agent4net-intent`

5.2 Intent Analysis

This section provides a detailed description of the set of intents used to evaluate our agent and the analyses performed on them. Our dataset consists of ten reference Kubernetes manifests, each paired with its corresponding intent. Since this number is limited, we expanded the intents dataset to increase variability in the generations. To do so, we employed an inference-based augmentation process: two different LLMs were used to generate multiple paraphrased versions of each original intent. Each model produced 30 rephrasings per intent, resulting in 60 intents per scenario and a total of 600 distinct intents.

This enlarged dataset allows us to study how the agent behaves under a wide range of linguistic formulations of the same underlying task.

5.2.1 Intents Similarity Metrics

To assess the quality of the generated intents and quantify how much they differ from one another, we rely on two complementary similarity metrics: one token-based and one embedding-based.
These metrics allow us to measure how strongly each rephrasing diverges from the original intent, capturing both **lexical** and **semantic** variation. Using both approaches is important because lexical similarity alone may not accurately reflect whether two intents express the same requirements. The combination of the two metrics therefore provides a more complete characterization of the variations introduced in the generated intents.

Jaccard Similarity Jaccard similarity is a statistical measure used to quantify the overlap between two sets. Its value ranges from 0 to 1: 0 indicates that the sets share no elements, 1 indicates that they are identical.

For two sets A and B, it is defined as:

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} \tag{5.1}$$

This metric is also known as Intersection over Union (IoU), since the numerator represents the intersection of the sets, while the denominator represents their union.

Jaccard similarity is an effective way to calculate document similarity, where documents are represented as sets of words. As a token-based method, it measures the degree of lexical overlap between two documents, capturing how many words they share and how many differ. This makes it particularly suitable for evaluating structural and syntactic similarity, since it reflects how closely two intents align in terms of wording and token composition.

Sentence Transformer Similarity If token-based methods compare words directly, embedding-based similarity methods evaluate how close two texts are in terms of semantic meaning rather than surface-level lexical overlap. To do that, each text is converted into a dense numerical vector, called *embedding*, that captures its contextual and semantic information.

Sentence Transformers provide a widely used approach for generating such embeddings. It produce fixed-size sentence-level representations that preserve semantic relationships: texts with similar meaning are mapped to vectors that lie close to each other in the embedding space.

Once the embeddings are obtained, similarity is computed using cosine similarity, defined as:

$$\cos_\text{sim}(u, v) = \frac{u \cdot v}{\|u\| \, \|v\|} \tag{5.2}$$

Cosine similarity measures the angle between two vectors, producing values between –1 and 1. Higher values indicate that the vectors point in similar directions, meaning the corresponding texts express semantically related content.

This embedding-based approach provides a deeper understanding of how closely two texts align in meaning, even when their lexical forms differ substantially.

5.2.2 Analysis of Intent Similarity

We compute both similarity metrics on the entire expanded set of intents to quantify how much the rephrasings differ from one another in terms of lexical variation, syntactic structure, and semantic meaning. Each comparison is performed independently for every reference scenario, ensuring that similarity is evaluated only within the group of intents derived from the same original task.

To facilitate interpretation, we also generate heatmaps for each similarity matrix. These visualizations highlight dense similarity regions, structural patterns, and potential cluster boundaries. A pair of representative heatmaps—one for each similarity metric—is included in the **Appendix B**.

Clustering Our first goal is to determine whether the rephrasings produced for each scenario give rise to distinct intents and to quantify their differences using the two similarity metrics. To this end, we begin by identifying clusters within the generated intents. Clustering provides a high-level view of the semantic landscape of the paraphrases and allows us to group highly similar formulations together, reducing redundancy and revealing meaningful variations.

To construct the clusters, we first load the similarity matrix and convert it into a distance matrix by computing $1 - S$, where S denotes the pairwise similarity score between items. This transformation is necessary because hierarchical clustering algorithms operate on distances rather than similarities. We then apply agglomerative hierarchical clustering with the average-linkage criterion, which iteratively merges the closest groups based on the average pairwise distance between their members.

This choice is well suited to our setting for two reasons.

- **Hierarchical clustering** operates directly on distance matrices and does not rely on assumptions such as spherical cluster shapes or predefined centroids. Compared with methods like k-means or DBSCAN, it is therefore better suited for similarity-based semantic data, where relationships are expressed through pairwise similarity scores rather than geometric structure
- **Average-linkage** provides a balanced compromise between single-linkage (which is sensitive to noise and chaining effects) and complete-linkage (which tends to produce overly compact clusters). In our case, average-linkage yields clusters that are both stable and representative of the overall semantic relationships among intents

The final cluster assignment is obtained by cutting the hierarchy at a chosen number of clusters k. Since the optimal value of k is not known a priori, we evaluate values from $k = 2$ to $k = 5$ using the silhouette coefficient, a standard metric that measures how well each item fits within its assigned cluster relative to others. The value of k that maximizes the silhouette score is selected automatically.

Finally, for each cluster we compute internal cohesion statistics—such as mean, minimum, and maximum internal similarity—and identify the *medoid*, defined as the most representative item of the group. This procedure allows us to group semantically related intents into coherent clusters while retaining an interpretable and data-driven criterion for determining the number of clusters.

Results At first glance, the clustering results obtained using the **Jaccard similarity** metric appear unusual. Across all reference scenarios, the optimal clustering consistently splits the intents into two groups: one cluster containing intents with very high mutual similarity, and another composed of more heterogeneous and structurally diverse formulations.

This behavior was initially unexpected. However, a qualitative inspection of the generated intents revealed that this metric tends to amplify differences arising from token frequency and distribution, making even minor structural variations appear more significant than they are.

As a result, the paraphrases naturally fall into two distinct structural categories:

- **Unstructured intents**, expressed as free-form sentences resembling dialogue or natural language instructions.
- **Structured intents**, characterised by explicit formatting elements such as symbols, lists, or semi-formal templates that impose a more rigid and recognisable structure.

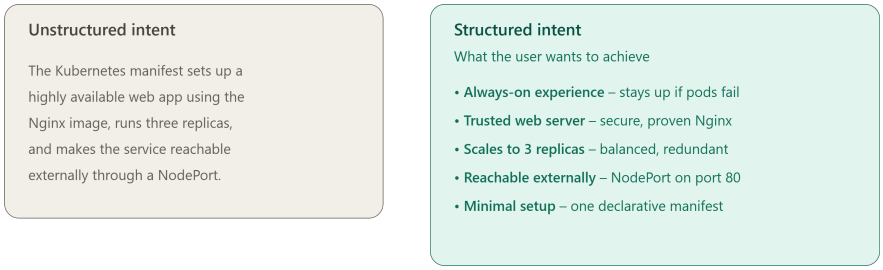


Figure 5.1: Simplified example of the difference between unstructured and structured intents

A closer look at the clusters confirms that this structural distinction is precisely what the Jaccard-based clustering captures: intents with similar token composition tend to group together, while structurally different intents form separate clusters.

Turning to the results obtained with the **Sentence Transformer** similarity metric, the behavior is partially different. Given the semantic nature of this metric, we expected high similarity scores across all intents derived from the same reference example. This expectation is largely confirmed: cosine similarity values rarely fall below 0.75 on average, indicating that all paraphrases preserve the core semantic meaning of the original instruction.

Nevertheless, even in this case the clustering reveals a subtle separation into two groups. These clusters almost perfectly overlap with those obtained using Jaccard similarity, although the distinction is less pronounced. This highlights an important property of embedding-based methods: while they do not rely on shared vocabulary, the degree of structural organisation of a text still influence the resulting embeddings and therefore the similarity scores.

This observation motivated a further line of investigation: whether the behavior of our agent changes depending on the structural form of the input intent. This hypothesis is explored in detail in **Section 5.3.2**.

5.3 Manifest Generation Quality

Once we have described the analyses performed on the intents used by the agent, we move on to the evaluation of manifest generation, with the goal of determining whether the agent produces reliable and correct outputs. We begin by presenting the metrics adopted for assessing similarity between manifests, followed by a discussion of the results obtained across the different test scenarios.

5.3.1 Manifests Similarity Metrics

As discussed in the background section, manifests are written in YAML format, and no dedicated metric exists for comparing YAML files directly. For this reason, we adopted some evaluation metrics already known in the state of the art for tokens similarity, and we additionally introduce a set of custom metrics specifically designed to capture structural and semantic similarity in these configuration files. These metrics allow us to evaluate how closely the generated manifests align with the expected outputs and to identify differences that may affect correctness or reliability.

We compute two types of comprehensive scores using a total of six metrics, each capturing different aspects of the generated YAML files [18].
The first method produces **text-level scores**, using metrics such as BLEU and Edit Distance to evaluate similarity at the textual level.
The second method produces a **YAML-aware score**, using CodeBLEU, Key Match, Key-Value Exact Match, and Key-Value Wildcard Match to assess structural and a sort of semantic correctness within the YAML format.
All the metrics are calculated and then normalized in a range of 0 - 1 to be comparable between them, with the higher scores being more desirable.

Here is how each metric is calculated:

BLEU (Bilingual Evaluation Understudy) BLEU [21] is a widely used metric for evaluating the quality of machine-generated text by measuring how similar it is to one or more reference sentences.
It measures the similarity by comparing tokens overlapping n-grams, representing contiguous sequences of words, between the generated text and the reference. The metric computes precision scores for 1-gram through 4-gram matches, capturing both local word accuracy and short-range fluency, and then combine these score using a weighted geometric mean. This ensures that the final score reflects consistent matching across different n-grams lengths rather than being dominated by only short or long matches.
In our project, we compute the BLEU score using the standard NLTK implementation, applying uniform weights across all n-gram levels.

Edit Distance Edit Distance is a metric that measures how different two pieces of text are by counting the minimum number of operations needed to transform one string into the other. The standard version, known as *Levenshtein distance*, count the smallest number of insertion, deletion and substitution of a character required to make the two strings identical. A lower value indicates that the generated text is more similar to the reference, while a higher value reflects greater divergence.
To compute this value, algorithms rely on dynamic programming: a matrix is built where each cell represents the cost of converting a prefix of one string into a prefix of the other. By filling this matrix step by step and considering the cheapest operation at each position, the algorithm finds the minimal transformation cost.

Because it evaluates similarity at the character level, Edit Distance is sensitive to small changes such as typos, formatting differences, or minor structural variations, making it useful for assessing fine-grained textual accuracy.

Also in this case we use the NLTK implementation, normalizing the result and transforming the distance into a similarity score.

codeBLEU codeBLEU [22] is an evaluation metric designed to adapt the standard BLEU score to programming languages. Traditional BLEU is tailored to natural language, but code differs substantially: it has a limited and highly meaningful set of keywords, a rigid syntactic structure and semantics that depend on compilation and data flow rather than sequential interpretation. Traditional BLEU's token-matching approach fails to capture all these key concepts.
CodeBLEU addresses these limits by combining four complementary components:

$$CodeBLEU = \alpha \cdot BLEU + \beta \cdot BLEU_{weight} + \gamma \cdot Match_{AST} + \delta \cdot Match_{df}$$

Where:

- $BLEU$ is the standard BLEU metrics.
- $BLEU_{weight}$ is the weighted n-gram match that assigns higher weights to keywords and other semantically important tokens, reflecting their greater significance in code.
- $BLEU_{AST}$ is the syntactic score computed using the Abstract Syntax Tree (AST), a tree representation of the program's structure. The algorithm extracts all subtrees from both the candidate and reference code and counts how many appear in both, ignoring order. This captures structural similarity beyond surface tokens.
- $BLEU_{df}$ is the semantic score based on data-flow analysis. Code is represented as a graph where nodes correspond to variables and edges represent how values propagate between them. Matching these graphs helps determine whether the candidate preserves the same variable flow and behavior as the reference.

The official implementation of codeBLEU, however, is not designed for imperative code formats such as YAML manifests. For this reason, inspired by the original formulation, we developed a variant better suited to configuration files of this type.

First, we adapted the weighting scheme used in the computation of $BLEU_{weight}$ by defining three levels of keywords: **High-weight tokens** include elements such as kind, metadata, and spec; **medium-weight tokens** include name, namespace, and image; and **low-weight tokens** cover boolean values and other symbols.

Second, for $BLEU_{AST}$, we employed a YAML-specific tree parser to construct a structural representation of the document that faithfully reflects YAML's hierarchical organization.

Finally the $BLEU_{df}$ component was disabled. YAML is not an executable language: it has no variables, assignments, or value-propagation flows. So, there is no semantic data flow to compare, unlike in imperative languages such as Python, Java, or C++, where variable dependencies and execution semantics are essential.

Key-Value Exact Match This metric evaluates how closely two YAML documents match by comparing their key–value structure rather than their raw text.

Different from the exact match that ignores the fact that order does not matter in YAML, the Key-Value Exact score is computed by loading both the generated and reference documents into dictionaries and measuring the structural and semantic differences between them.

The comparison is calculated at leaf items level, meaning every terminal value in the YAML tree is examined. By comparing all leaves independently of their position, the metric ensures that each key–value pair is evaluated for correctness, focusing solely on whether the content is, while fully ignoring ordering.

Key-Value Wildcard Match This metric is a variant of the Key–Value Exact Match designed to handle a specific characteristic of Kubernetes configuration files.

In these manifests, several fields (key) are required to define the resource object, but their specific values are arbitrary. Typical examples include resource names, namespace names, labels, and annotations. Similarly, it is often acceptable for a cluster to use either `ubuntu:22.04` or `ubuntu:20.04` as an image; such variations naturally occur across generations and environments and should not influence the similarity score.

To address this, the algorithm extends the Key–Value Exact Match by introducing a **wildcard list**: a set of leaf nodes whose values must be ignored during comparison. Any mismatch on these fields is intentionally excluded from the difference count. By combining structural comparison with selective value tolerance, the metric preserves the hierarchical organization of the YAML document, ignoring ordering, and accepts minor variations in specific fields. This results in a more semantically meaningful and context-appropriate similarity measure.

Key Match Defining a wildcard list that works well across all Kubernetes resources without oversimplifying the comparison or ignoring values that are actually important is not straightforward. For example, the port specified in a Kubernetes Service is often arbitrary unless the user explicitly requests a particular value. This makes it difficult to design a universal wildcard list that is both flexible and precise. In addition, we observed that a complementary metric was needed to provide a more general view of the structure of the automatically generated manifest, independently of the specific values chosen by each model.

To address these challenges, we introduce this additional metric, called *Key Match*, whose objective is precisely to ignore all values and compare only the keys of the leaf nodes in the YAML structure, thereby capturing the high-level organisation of the file. This provides a structural measure of similarity between the generated and reference manifests, independent of the specific values produced by each model or the configuration choices made in different environments. By focusing solely on the presence and arrangement of keys, this metric assesses whether the generated manifest adheres to the expected schema and structural layout, offering a complementary perspective to value-based similarity measures.

When computing all these metrics on our manifests, we expect text-level scores to be significantly lower than YAML-aware scores. Text-based metrics are well suited for comparing natural language, but in structured formats such as YAML or code they are heavily affected by superficial differences, including ordering and layout. Despite this limitation, text-level metrics still provide insight into how closely the generated manifest matches the reference in terms of sequence and exact construction, helping us assess whether certain models are more consistent in reproducing the reference manifest from an intent.

In contrast, the YAML-specific metrics are designed explicitly for structured files and capture the functional characteristics of each object. For this reason, we expect them to yield more significant and higher scores. These metrics allow us to evaluate how the generated manifest differs from the reference in terms of actual functionality and object-specific properties, without being influenced by ordering, formatting, or other structural variations.

In all subsequent evaluations, each manifest generated by our agent is compared against the corresponding base manifest used as the reference for that specific scenario and included in the complete dataset introduced earlier. Each metric is computed separately and independently for every resource kind (e.g., Deployment objects are evaluated only against other Deployment objects) to ensure greater accuracy. The aggregate value reported in the following analyses and tables represent the mean computed across all resource kinds present in the manifest.

5.3.2 Analysis of Intent Structure Effects

To evaluate how the structure and format of the intent influence the quality of the generated outputs, we tested each model using two versions of the intent for every scenario in our dataset: a **non-structured**, human-readable intent and a **structured** intent, expressed through a table, list, or other formal schema.

We selected these two intents by choosing the most structurally distant pair identified through the *Jaccard-based clustering* discussed earlier. This ensures that the evaluation captures how models behave when presented with intents that differ maximally in format.

The **Table 5.2** reports the results of these generations, using the scoring criteria introduced in the previous section.

Metric	Intent	Granite-4	L3.3-70B	L4-Mav	GPT-120B	Mistral-S	Mistral-L
BLEU	*Simple*	0.53	0.49	0.55	0.53	0.51	**0.56**
	Structured	0.52	0.48	0.50	0.52	0.53	**0.56**
Edit Distance	*Simple*	0.69	0.68	**0.71**	0.68	0.67	**0.71**
	Structured	0.67	0.66	0.66	0.67	**0.70**	0.69
codeBLEU	*Simple*	0.60	0.56	**0.63**	0.60	0.59	**0.63**
	Structured	0.59	0.55	0.58	0.61	0.61	**0.63**
KV Match	*Simple*	0.64	0.62	**0.66**	0.61	0.63	0.64
	Structured	0.61	0.61	**0.65**	0.63	0.63	0.64
KVW Match	*Simple*	**0.89**	0.86	0.86	0.84	0.88	0.88
	Structured	0.86	0.86	0.83	0.85	0.87	**0.89**
Key Match	*Simple*	**0.96**	0.90	0.90	0.91	0.92	0.95
	Structured	0.93	0.91	0.86	0.87	0.93	**0.96**

Table 5.2: Comparative analysis of average model performance under Simple and Structured intents (bolded values marking the highest score per metric)

Legend:
Granite-4 = granite 4-h-small;
L3.3-70B = llama 3.3-70b-instruct;
L4-Mav = llama 4-maverick-17b-128e-instruct-fp8;
GPT-120B = gpt-oss-120b;
Mistral-S = mistral small 3.1-24b-instruct-2503;
Mistral-L = mistral large-2512.

Analyzing these results, we observe that the six models achieve broadly similar average performance within each individual metric, with no substantial differences at the aggregate level. However, some trends do emerge when examining the results more closely.

First of all, as expected, the average **text-level scores** are significantly lower than the YAML-aware scores for every model. Although the manifests are generated from intents derived from the reference, each model is free to structure and format the YAML output differently. Variations in ordering, indentation, field arrangement, and even minor differences in values strongly affect text-level metrics. In particular, BLEU performs worse than Edit Distance because BLEU is highly sensitive to exact token sequences and n-gram overlap, making it especially vulnerable to structural and ordering differences. Edit Distance, instead, tolerates reordering better and focuses on the minimal number of edits required to transform one sequence into another, which makes it slightly more robust for structured formats.

YAML-aware metrics, on the other hand, achieve significantly higher scores. Among them, codeBLEU is the lowest, which aligns with how the metric is constructed: it extends traditional BLEU with adjustments tailored for code and structured formats, but it still inherits BLEU's sensitivity to token ordering and surface-level structure. As a result, its performance remains strongly influenced by the limitations of BLEU, even when applied to YAML.

As expected, Key-Value Wildcard Match and Key Match achieve the highest scores across all models. This follows directly from the nature of these metrics: both are designed to focus on the structural and semantic characteristics of YAML manifests rather than their textual surface form.

Key-Value Wildcard Match evaluates only the most relevant key–value pairs, ignoring fields that do not affect the functional behavior of the Kubernetes object. By filtering out non-essential differences, such as metadata fields or optional configuration details, it captures the core semantics of the manifest and is therefore less sensitive to formatting or ordering variations.

Key Match, instead, analyzes the "skeleton" of the manifest by comparing only the set of keys present in the generated and reference files. Since it does not consider values or ordering, it is even more robust to structural differences and tends to produce consistently high scores whenever the model generates the correct object structure. Together, these metrics reward semantic correctness and structural fidelity, which explains why they outperform text-level metrics that are more sensitive to formatting or token-level differences.

Talking now about how the **structure of the intent affects model performance,** we need to examine the results more closely.

When looking at the best score for each metric across all models, we see that Simple and Structured intents produce almost identical top values. This suggests that, at a global level, the structure of the intent does not lead to substantial performance differences. However, when we analyze the results within each model, a more nuanced picture emerges.

`Granite-4-h-small` and `llama-4-maverick-17b-128e-instruct-2503` consistently perform slightly better with Simple intents, indicating that these models may benefit from less constrained or less structured input. In contrast, the other models do not show a clear preference: they do not systematically perform better with either Simple or Structured intents, and in several cases the scores for the two intent types are exactly the same.

To examine more fine-grained differences in the results obtained across models, we report the complete set of boxplots in **Figure 5.2**. These visualizations allow us to move beyond aggregate means and inspect the full distribution of scores for each metric across all models. By analysing the entire distribution, we can observe aspects such as median, variance, range, and outliers, gaining a more nuanced understanding of how models behave under both Simple and Structured intents. This perspective is particularly valuable because it highlights stability, dispersion, and occasional extreme behaviors that aggregate statistics alone would not reveal, offering a clearer view of the consistency and robustness of each model's generations.

Across all metrics, the overall shape of the distributions remains similar between models, indicating that no model consistently outperforms or underperforms the others. However, when comparing Simple and Structured intents, several meaningful differences emerge.

For BLEU, Edit Distance, codeBLEU, and Key-Value Match, Structured intents exhibit **higher variability** than Simple intents. At the same time, Structured intents tend to reach higher maximum scores, suggesting that they occasionally allow models to produce more accurate manifests, but also introduce greater inconsistency. This increased variability highlights a difficulty for some models in maintaining stable performance across different attempts when the intent becomes more structured.

In contrast, Key-Value Wildcard Match and Key Match show substantially lower variability on average. This behavior is expected: these metrics ignore portions of the manifest that do not affect functionality but would otherwise introduce noise when comparing the raw textual structure. As a result, they are less sensitive to formatting differences and more focused on the semantic correctness of the generated YAML.

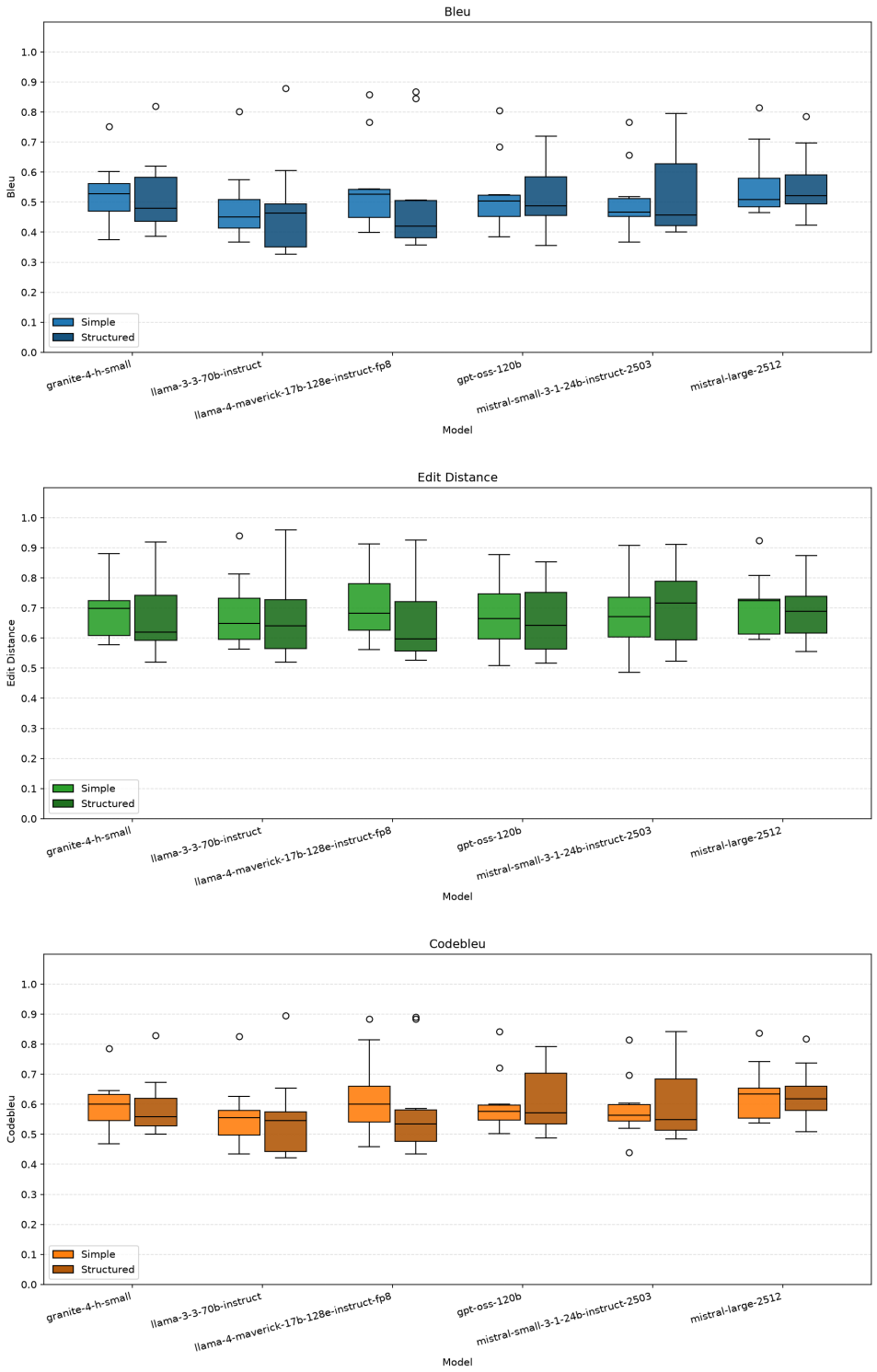
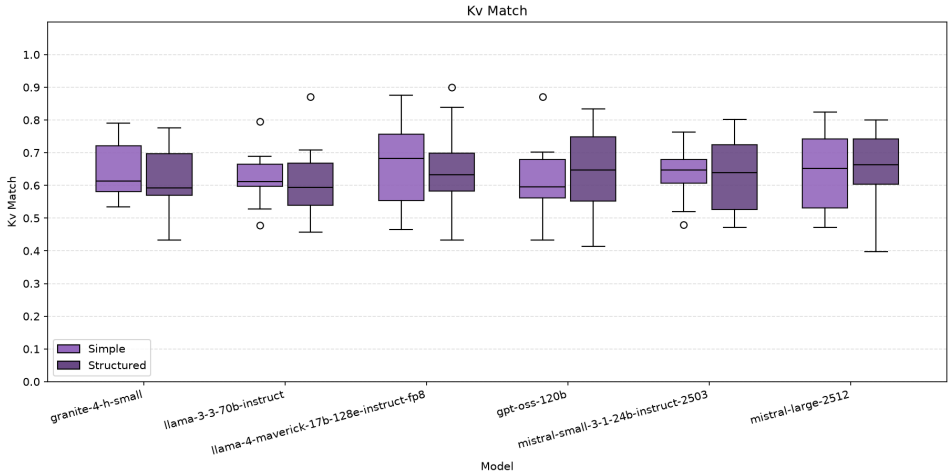
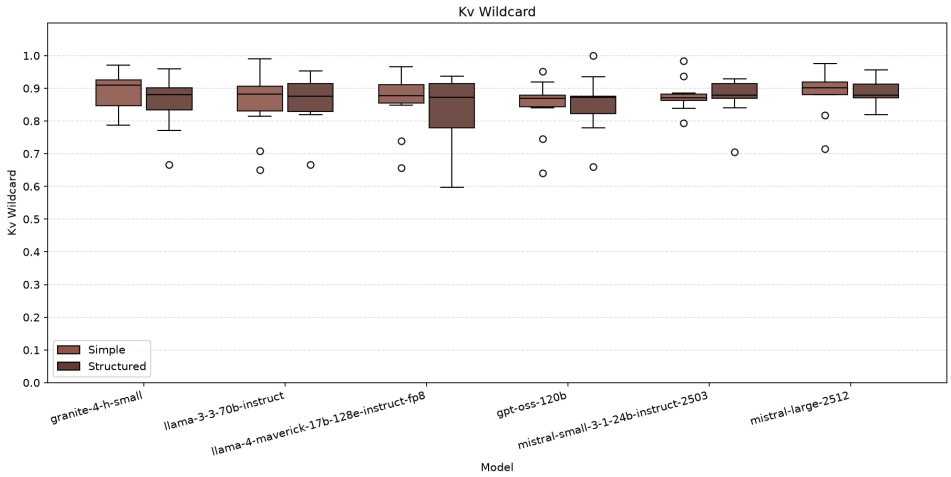


Figure 5.2: Comparison between Simple and Structured prompts for the similarity metrics across the six tested models





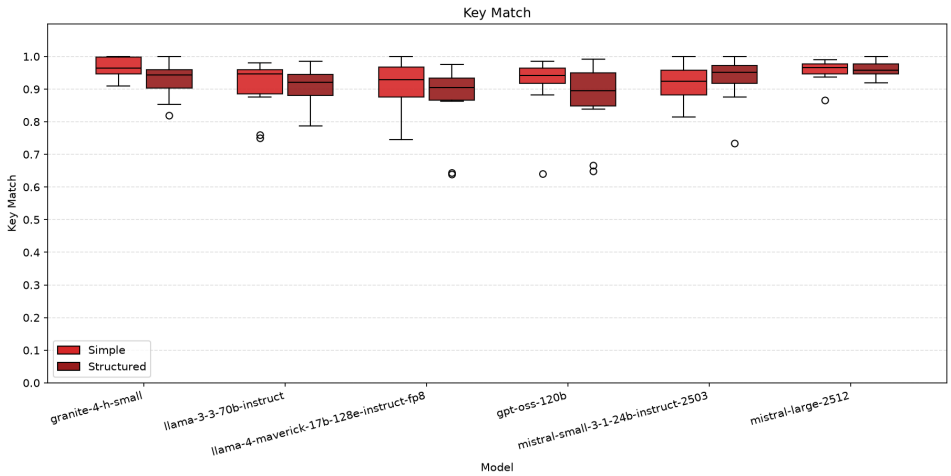


Figure 5.2: (continued)

What highlights the real differences between Simple and Structured intents are two additional metrics that we introduced to evaluate the actual quality of the generated manifests, not in terms of textual similarity, as the previous six metrics do, but in terms of the semantic composition of the objects contained in the manifest. In several cases, we observe that a model chooses to generate a different set of Kubernetes object kinds compared to the reference: sometimes producing more objects, sometimes fewer, and occasionally replacing one kind with another. These deviations affect the functional correctness of the manifest and cannot be captured by text-level or YAML-aware similarity metrics alone.

To study this phenomenon more effectively, we introduce two complementary metrics:

- ***Precision***: measures how many of the generated kinds are actually expected in the reference. A low precision indicates that the model is producing extra or unexpected kinds that do not belong to the target manifest.
- ***Recall***: measures what fraction of the expected kinds in the reference are present in the generated manifest. A low recall indicates that the model is missing some required kinds.

Together, these metrics allow us to evaluate whether the model generates the correct set of Kubernetes objects for a given intent, independently of formatting, ordering, or textual similarity. They provide a functional perspective on manifest quality and reveal differences that are not visible through the previous metrics.

In **Figure 5.3**, we clearly see that the average values of both precision and recall follow almost the same pattern across all models.

For Simple intents, both metrics frequently reach a perfect score of 1, with very low variability. This is expected: in most generations, the models produce exactly the same set of Kubernetes object kinds as the reference, and only a small number of outputs introduce additional or missing kinds.

In contrast, Structured intents lead to noticeably lower scores and significantly higher variability. This indicates that when the intent becomes more detailed or more constrained, some models struggle to consistently identify the correct set of object kinds.

Among all models, `gpt-oss-120b` is the most affected by the structure of the intent: it shows the highest variability and the lowest precision and recall values. On the other hand, `mistral-small-3-1-24b-instruct-2503` and `mistral-large-2512` are the least impacted, especially in terms of precision. Their mean and variance remains low and similar for both Simple and Structured intents, making them the most resilient and robust models with respect to intent structure.

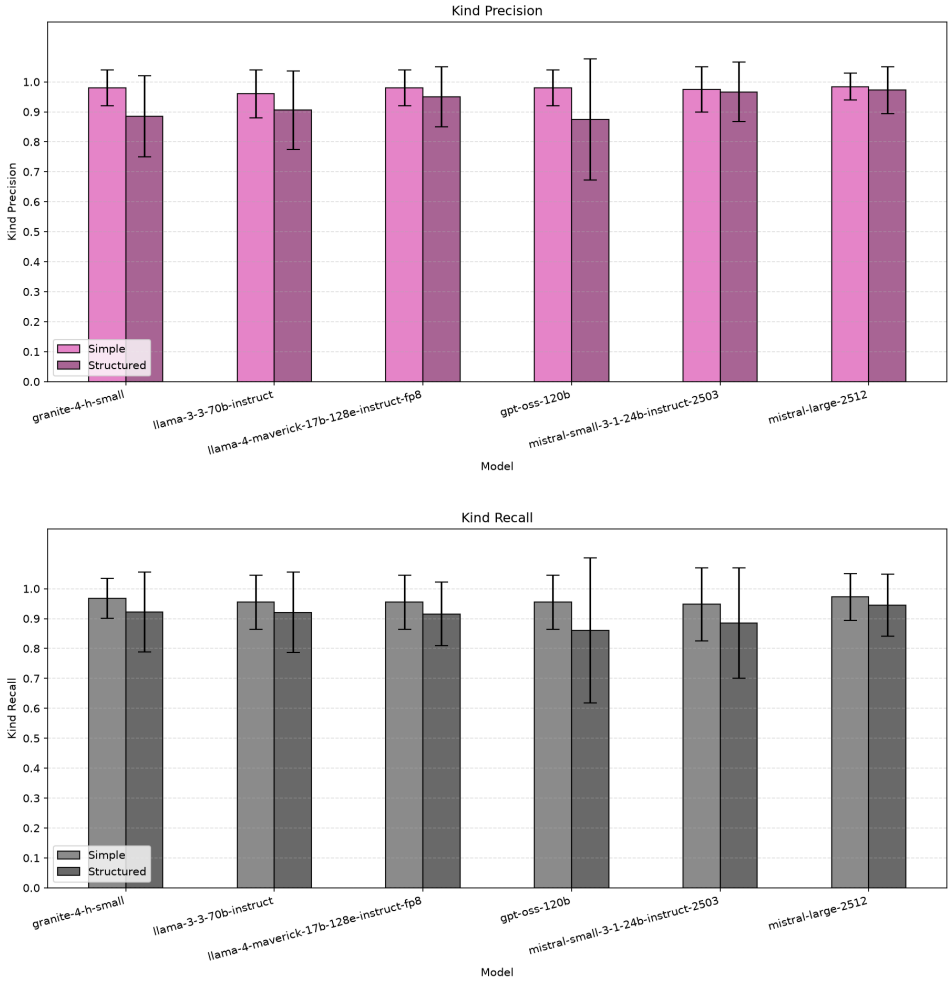


Figure 5.3: Comparison between Simple and Structured intents for Precision and Recall across the six tested models

Summary Overall, we observe that the overall similarity between manifests generated from Simple and Structured intents does not change dramatically. What does change, however, is the precision and stability that models can achieve: Simple intents allow all models to perform well in terms of precision and recall, meaning they correctly identify the set of object kinds required by the task, whereas Structured intents introduce greater variability and occasional drops in quality, reducing reliability for several models.

Among all evaluated models, `mistral-large-2512` consistently obtains the highest similarity scores and shows the lowest variability across examples and agent attempts. It is therefore the most reliable model in terms of both overall scoring and the accurate identification of the correct set of Kubernetes object kinds.

5.3.3 Analysis of Temperature Effects

In this second scenario, we investigate whether changing the model's temperature affects its performance. As discussed in **Chapter 2.2**, a high temperature increases variability by making the model more likely to select less probable tokens, whereas a low temperature makes the model more deterministic, consistently choosing the most probable token at each step.

To evaluate this effect, we test each model using three different temperatures:

- ***0.1***: low and highly deterministic
- ***0.7***: medium level, commonly used as a default
- ***1.4***: high and strongly stochastic

For every temperature, we reuse the same ten scenarios from the dataset, each paired with two distinct intents as in the previous analysis, resulting in **20 generations per model per temperature**.

It is important to highlight that our agent, as previously described, is constrained by a maximum of seven attempts to correct and regenerate a valid manifest. In some cases, the agent fails to produce a correct manifest within these seven attempts, causing the generation to terminate unsuccessfully. When this occurs, no manifest is produced and therefore no similarity metric can be computed; these failed generations must be discarded from the analysis. Although such failures are relatively infrequent overall, they can still influence the final results.

The following **Table 5.3** reports, for completeness, the number of successful generations for each model and temperature setting.

Model	Temp 0.1	Temp 0.7	Temp 1.4
granite-4-h-small	20	20	20
llama-3.3-70b-instruct	17	20	17
llama-4-maverick-17b	19	20	20
gpt-oss-120b	20	20	20
mistral-small-3.1-24b	19	20	1
mistral-large-2512	18	18	19

Table 5.3: Number of generations successfully completed for each model and temperature setting, out of a maximum of 20

The results show that, in most cases, the models achieve the maximum number of successful generations. However, the temperature that is consistently the most problematic is **1.4**. This outcome aligns with the expected behavior of models operating at high temperature: increased stochasticity leads to a higher likelihood of producing less relevant tokens, which in turn results in manifests that are more imprecise or structurally inconsistent.

Temperature 0.1 also exhibits lower performance in some models compared to 0.7. This reduction can be attributed to the highly deterministic nature of low-temperature generation.

A particularly relevant observation for the subsequent analyses concerns the behavior of `mistral-small-3-1-24b-instruct-2503` at temperature 1.4: only a single manifest was successfully generated. Examination of the intermediate outputs reveals a consistent pattern. The model initially produced a structurally valid manifest, but after a few lines it began to deviate, inserting textual descriptions of the manifest or comments about corrections it believed should be applied. In practice, it appeared to "forget" the original objective, progressively drifting away from the required structure and intent.

For this reason, we decided not to include this result in the following analyses, in order to avoid misinterpreting the model's behavior at this temperature. A single successful generation is insufficient to provide a reliable indication of the model's performance under such highly stochastic conditions. Consequently, for this specific model no data are available at temperature 1.4, and the corresponding value is omitted from the following plots.

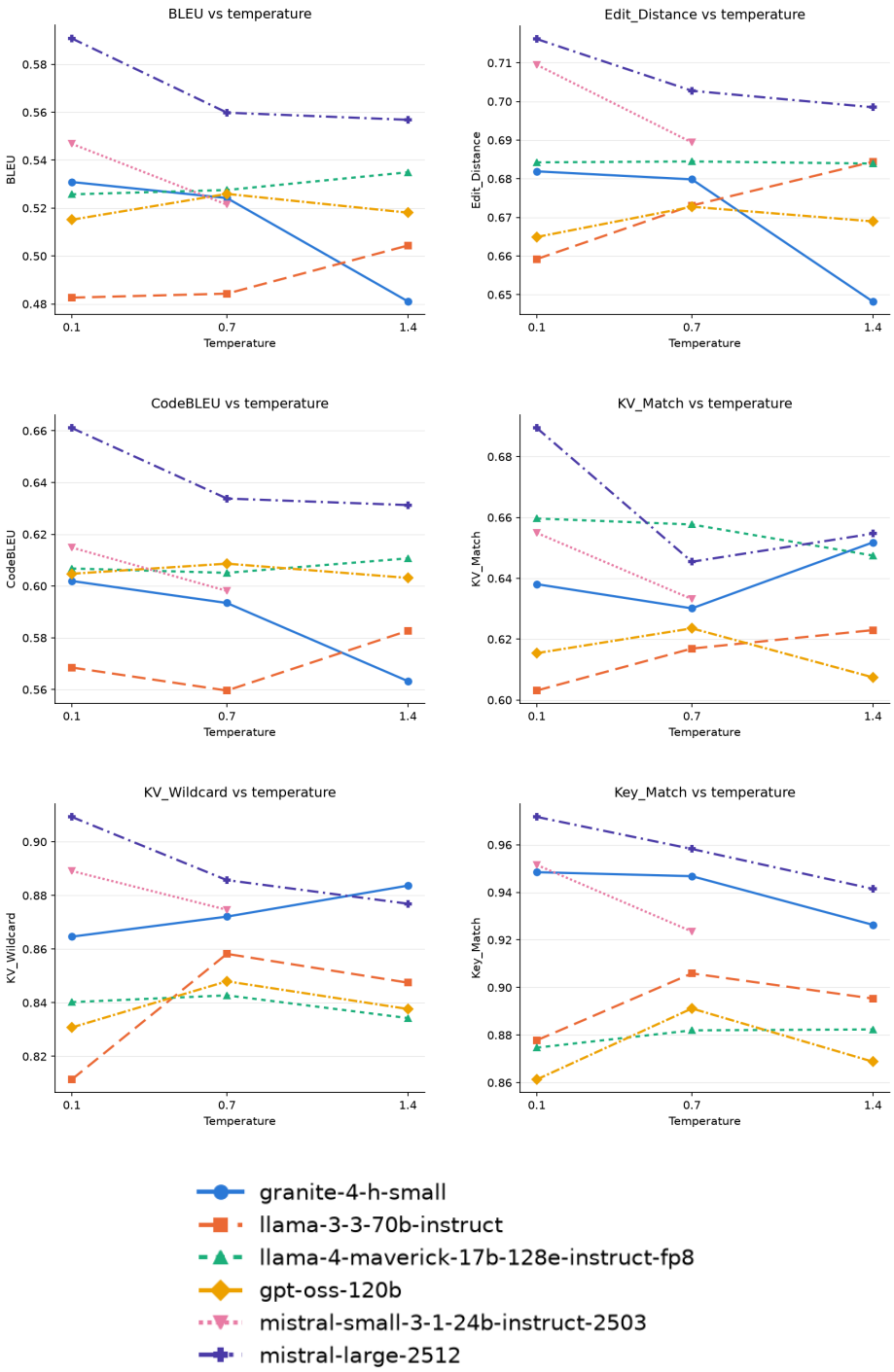


Figure 5.4: Comparison of the six models across temperatures using different similarity metrics

In **Figure 5.4** we report all the similarity-metric plots for every model and temperature setting.

Across all metrics, both low and high temperatures tend to produce greater variability, whereas intermediate temperatures generally yield more stable and consistent scores. This behavior aligns with expectations: extreme sampling temperatures increase stochasticity, while moderate values promote more controlled generation. Overall, most metrics exhibit a **decline in performance as temperature increases**. This effect is particularly evident for text-level metrics, but it also appears in the YAML-aware metrics for the majority of models.

The six models display distinct sensitivities to temperature. Some maintain relatively stable performance across all settings (e.g. `llama-4-maverick-17b-128e-instruct-fp8` and `gpt-oss-120b`), while others show a sharp degradation at higher temperatures (e.g `mistral-large-2512` and `mistral-small-3-1-24b-instruct-2503`). A few models even improve on some metrics while worsening on others (e.g `granite-4-h-small` and `llama-3-3-70b-instruct`). These differences highlight variations in sampling robustness and calibration across architectures.

As observed in the previous scenario, certain models consistently outperform others across multiple metrics and temperatures. In particular, `mistral-large-2512` achieves the highest scores across nearly all metrics. However, it is also the only model that fails to generate few manifests at every temperature (**Table 5.3**). A plausible explanation is that its internal consistency checks are more conservative than those of the other models, causing it to reject or truncate generations. This behavior naturally leads to higher average scores, but also to a lower overall number of valid manifests. Because this mechanism is directly influenced by the intent provided to the agent, refining or tailoring the intent for this specific model can help mitigate the issue.

Summary Overall, this scenario demonstrates that temperature exerts a measurable influence on model behavior across all similarity metrics. Extremely low and high temperatures consistently introduce greater variability and reduce reliability, whereas intermediate values provide the most stable operating regime. Although the general trend across models is a decline in performance as temperature increases, each model exhibits this behavior with different intensity, typically fluctuating around a comparable mean value, with only a few isolated exceptions.

Among all evaluated models, `mistral-large-2512`, once again, achieves the highest similarity scores across all temperature settings, confirming its suitability for this type of generation task when similarity to the target manifest is the primary objective. These findings highlight the importance of selecting an appropriate temperature and indicate that a mid-range value remains the most effective choice, offering a balance between accuracy and low variability.

5.4 Economic Evaluation

After assessing how the agent generates manifests, how closely these align with the reference outputs, and how variations in intent structure and temperature influence the final results, we now shift our focus to a complementary but equally critical dimension: the **economic efficiency of the system**.

While qualitative correctness and structural robustness determine whether the agent is suitable for the intended task, the practical deployment of any LLM-based pipeline also depends on its computational cost. In real-world settings, especially when agents are executed repeatedly or integrated into production workflows, time and token consumption directly translate into monetary cost, latency, and resource allocation constraints. For this reason, evaluating the economic footprint of each model is essential to understanding the feasibility and sustainability of the agent.

In this section, we address two central questions:

- ***Execution Time***: How long does each model take to complete its task within the agent pipeline?
- ***Token Consumption***: How many total tokens does each model require to produce a manifest?

To answer these questions, we analyze all results collected across the full suite of tests presented in the previous sections. The goal is to provide a clear understanding of how each model behaves not only in terms of accuracy and reliability, but also in terms of computational cost, ultimately guiding the selection of the most effective model for real deployment.

5.4.1 Execution Time

First, we examine the time consumption of tests performed on our agent.

As described in **Chapter 4**, the agent operates through a sequence of five consecutive steps, each responsible for a specific stage of the manifest generation pipeline. This sequence can be repeated whenever an error is detected at any stage, allowing the agent to refine its output until a correct and complete manifest is produced.

The agent is allowed a **maximum of seven iterations**. Depending on where an error occurs, a new iteration may involve only the generation step or may require re-executing the entire pipeline up to the final validation. Consequently, the total execution time is strongly influenced by both the number of attempts required and the specific step at which the failure arises.

To understand how efficiently each model navigates this iterative process, we begin by analyzing the average number of attempts each model needs to successfully complete a full cycle.

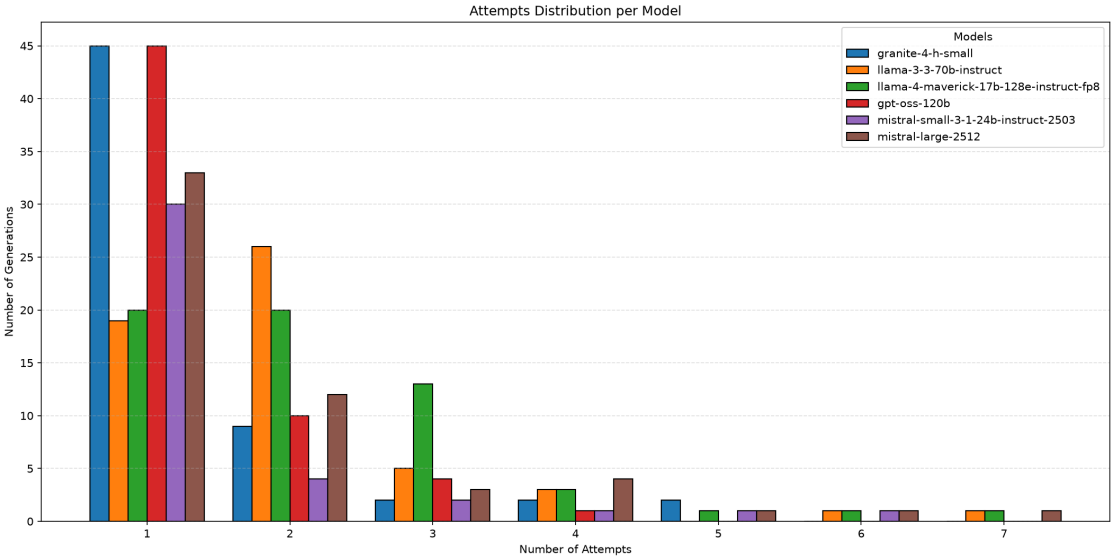


Figure 5.5: Distribution of generation attempts across models

As shown in **Figure 5.5**, the distribution of attempts is largely consistent across models. In most cases, the agent succeeds in producing a correct and functional manifest on the first attempt, indicating that the initial generation step is generally reliable and that the agent rarely needs to trigger the full correction cycle.

The only notable deviation from this pattern is observed in `llama-3-3-70b-instruct`, where the most frequent number of attempts is two rather than one. This suggests a slightly lower ability to produce a valid configuration on the first try, requiring an additional correction step more often than the other models.

Overall, the distribution highlights a strong self-correction capability of the agent: the number of generations decreases sharply as the attempt count increases, meaning that errors, when they occur, are typically resolved early in the cycle. From the perspective of attempt efficiency, the fastest models appear to be `granite-4-h-small` and `gpt-oss-120b`. Both exhibit a high proportion of successful first-attempt generations and almost no cases requiring four or more iterations.

A further observation concerns the comparison between Simple and Structured intents. As discussed in the previous section, these two prompting styles lead to markedly different behaviors. The plots in **Appendix C** show a shift in the attempt distribution: with Simple intents, generations are strongly concentrated in the first attempt position, whereas Structured intents produce a noticeably more dispersed distribution, with higher frequencies at larger attempt counts. This effect

is particularly evident for `llama-3-3-70b-instruct`.

Under Simple intents, its pattern aligns with the other models, with most generations completed in one attempt. Under Structured intents, however, the earlier trend reappears, and is significantly amplified, showing a dominant peak at two attempts and a heavier tail toward higher attempt counts.
These results reinforce the conclusion that, even when considering non-structural metrics such as attempt count, models tend to perform better with less rigid, more natural textual intents.

Taken together, these patterns offer a preliminary indication of how model stability may influence the agent's computational cost. Models that consistently complete the task within one or two attempts are expected to require fewer tokens and shorter execution times, whereas models that more frequently trigger additional iterations may incur higher latency and greater token consumption.

Returning to the focus of this section, we now examine how the total execution time varies across models to assess whether, as one would reasonably expect, there is a correlation between the number of attempts required and the overall time spent by the agent.

We count the **elapsed time** for each generation from the start of the request to the end of the complete generation, taking into account all the steps in the agent cycle. In **Figure 5.6** there is the aggregate results for each model.

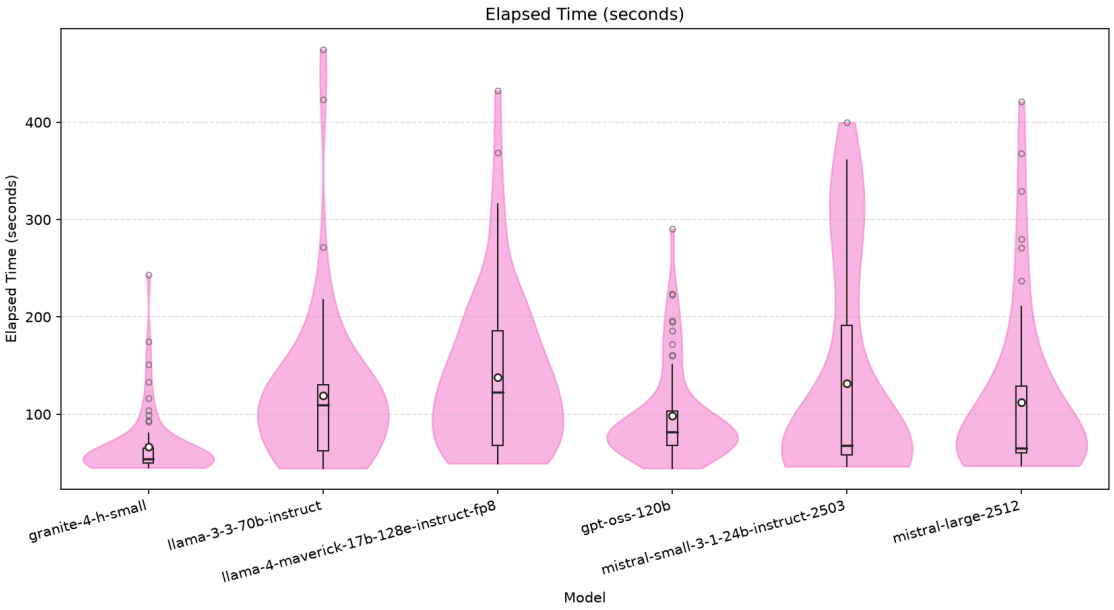


Figure 5.6: Distribution of response time for each model

The distribution of execution times confirms the trend observed in the attempts chart. The fastest and most stable models are `granite-4-h-small` and `gpt-oss-120b`, which show average elapsed times of approximately 50 seconds and 100 seconds, respectively. Their curves are sharply concentrated around low values, reflecting both efficient first-attempt success and minimal need for additional correction cycles.

In contrast, the distributions of the remaining models are noticeably more dispersed. This wider spread is caused by a higher number of generations requiring multiple attempts, which naturally increases the total time needed to complete the task. Nevertheless, even for these models, the peak of the distribution remains located at relatively low execution times, indicating that most generations still terminate early in the cycle.

A final consideration concerns how we measure the agent's elapsed time. Each stage of the pipeline has a different execution cost, and, given how the system is architected, the Kubernetes validation phase is unquestionably the most time-consuming. This step is intrinsically expensive because the validator must:

- boots a fully functional Kubernetes cluster inside a container runtime,
- applies all manifests generated by the agent,
- captures and reports any deployment or configuration errors,
- tear down the cluster and the underlying runtime once validation is complete.

This lifecycle (`start → apply → delete`) is significantly more time consuming than the other components of the pipeline. To quantify how this phase impacts the overall runtime, **Figure 5.7** reports two distinct values:

- ***Total execution time*** of the entire agent pipeline
- ***Cluster lifecycle time***, the time specifically required to start the cluster, apply all manifests, and delete the cluster.

The Kubernetes validation consistently accounts for at least **70%** of total runtime across all tested models. The remaining time is distributed fairly among the other stages. This separation highlights two essential points.

First, the validator is the primary source of latency, and any optimization effort must begin here. Second, this step is fundamental to the agent's purpose: it is the only phase that validates the actual operational correctness of the generated manifests. In other words, the validator is expensive because it is doing real work, and its cost is justified by the reliability guarantees it provides.

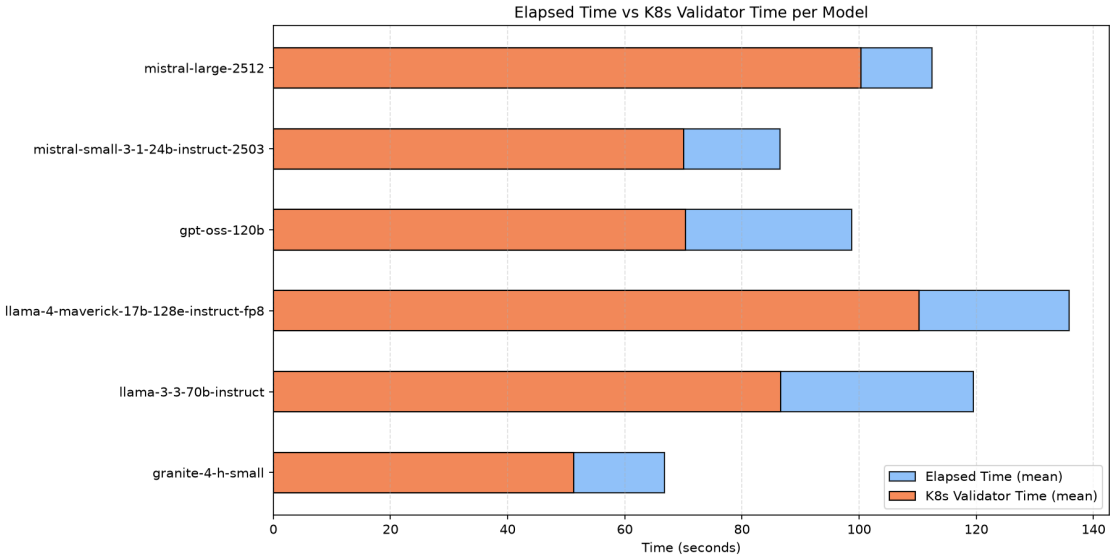


Figure 5.7: Comparison between the mean total elapsed time and the mean duration of the Kubernetes validator phase

In conclusion, distinguishing total execution time from cluster lifecycle time makes the performance profile of the agent transparent. It shows that while LLM-related operations contribute to variability, the Kubernetes validator is the structural bottleneck and simultaneously the core enabler of trustworthy, end-to-end manifest validation.

5.4.2 Token Consumption

After exploring how models perceive and handle time, we can now look at how they process tokens, the fundamental units of text that language models operate on. A token is the smallest chunk of text the model recognizes chosen because it carries semantic weight within the model's learned vocabulary.

When a model interprets a request and produces a response, it works with two main categories of tokens. **Input tokens** are the tokens consumed from the user's prompt, system instructions, and conversation history. **Output tokens** are the tokens the model generates, both for its internal reasoning and for the final response you see. Together, these tokens determine how the model interprets text, how much context it can retain, and how it ultimately constructs meaningful answers.

To evaluate how many tokens a model typically consumes when producing a response file, we measure the total token usage across all LLM calls involved in the generation pipeline.

In our architecture, each response is produced through three sequential LLM interactions:

- Scope Consistency check
- Manifest Generator
- Semantic Consistency check

Each step contributes its own input and output tokens. The total token count therefore represents the complete computational footprint of the entire pipeline. Formally, for a pipeline composed of n steps, the total number of tokens is:

$$\text{total_tokens} = \sum_{i=1}^{n} (\text{input_tokens}_i + \text{output_tokens}_i)$$

This formulation makes explicit that token usage is accumulated across all stages of the process, capturing both the text the model ingests and the text it produces throughout the entire workflow, including any cycles of autocorrection.

This also reveals a direct correlation between the number of attempts performed by the agent and the total number of tokens consumed. When a model requires multiple retries to produce a valid configuration file, each additional attempt contributes extra input and output tokens. As the number of iterations increases, so does the overall computational footprint.

Let us now examine how our models behave under this expanded scenario.

The results in **Figure 5.8** reveal several noteworthy patterns. Overall, token usage remains remarkably consistent across models: the mean number of tokens consumed throughout the full pipeline lies between 4000 and 5500 and the minimum values range from roughly 850 to 1400 tokens.

In the previous analysis focused on the number of attempts, we observed a clear performance gap: `granite-4-h-small` and `gpt-oss-120b` consistently required fewer retries, while both `llama` models struggled more. One might therefore expect this distinction to manifest in token consumption as well.

Surprisingly, this is not what we observe. The token distributions do not show a clean separation between the two groups. Instead, a different and unexpected pattern emerges. `granite-4-h-small`, `llama-3-3-70b-instruct`, and `llama-4-maverick-17b-128e-instruct-fp8` tend to consume more tokens, particularly in complex scenarios, whereas `gpt-oss-120b`, `mistral-small-3-1-24b-instruct-2503`, and `mistral-large-2512` exhibit lower token usage and a much tighter distribution.

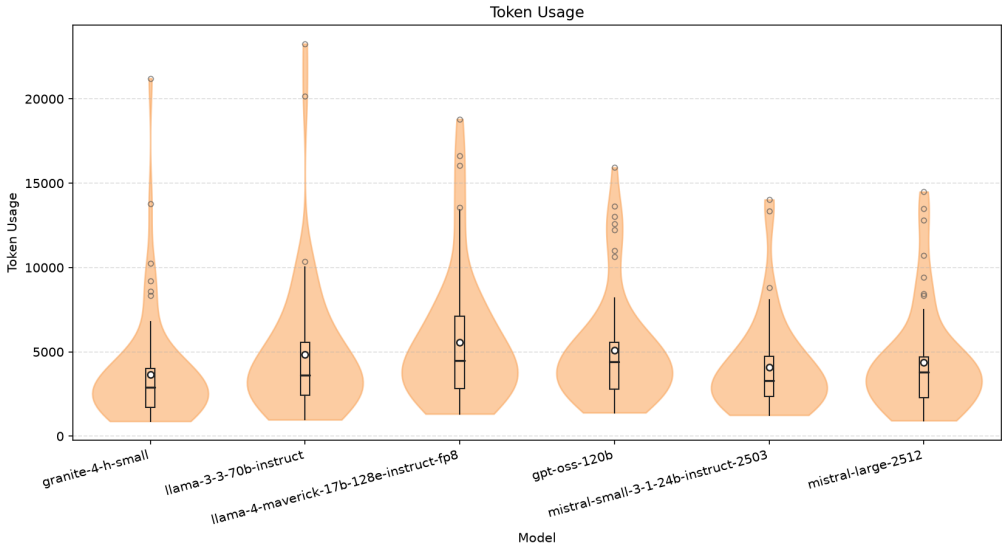


Figure 5.8: Distribution of total token consumption for each model

This divergence may appear incidental at first glance, but it aligns with a key shared characteristic within each group. `granite`, `llama-3`, and `llama-4` are models equipped with explicit reasoning capabilities, whereas `gpt`, `mistral-small`, and `mistral-large` do not employ an active reasoning mechanism. Reasoning involves generating intermediate steps of thought to improve the final output, which naturally increases the number of tokens produced, and this is exactly what we observe in our data.

In our specific case, this divergence is substantial: the difference between the average maximum token usage of reasoning-enabled models and non-reasoning models approaches 30%, highlighting a clear quantitative separation between the two families.

A final aspect to evaluate in this section is the effective **price per token** across the different models. LLM providers typically apply separate rates for input tokens, usually cheaper, and output tokens, which tend to cost more. However, for the purposes of this thesis, we focus on the total number of tokens processed throughout the entire flow, without distinguishing between input and output.

Consequently, the prices reported in **Table 5.4** represent an average cost per token, obtained by combining and normalizing the provider's input and output token rates into a single mean value. These values are sourced from *Artificial Analysis* [23], a platform that aggregates and compares pricing, performance, and technical characteristics of numerous commercial LLM providers. It offers a unified view of model costs, enabling consistent cross-provider comparisons.

For each model, the price range reported here corresponds to the *minimum* and **maximum cost per million tokens** among all providers listed on the website, ensuring that the analysis reflects the full variability of market offerings.

Model	MIN	AVG	MAX
granite-4-h-small	**0.06**	0.15	0.25
llama-3.3-70b-instruct	0.12	1.18	**2.25**
llama-4-maverick-17b	0.10	0.80	1.50
gpt-oss-120b	**0.06**	0,40	0.75
mistral-small-3.1-24b	**0.06**	0.31	0.56
mistral-large-2512	0.60	0.80	1.00

Table 5.4: Average min and maxi cost per million tokens across providers (USD)

Several observations emerge. First, models such as `granite-4-h-small`, `gpt-oss-120b`, and `mistral-small-3.1-24b` exhibit very low minimum prices (around $0.06 per million tokens), indicating that some providers offer highly competitive rates for these architectures. This makes them attractive for large-scale or cost-sensitive workflows. Conversely, models such as `llama-3.3-70b-instruct` show a much wider price range, with maximum values exceeding $2 per million tokens. This variability suggests significant differences in pricing strategies among providers, likely influenced by factors such as infrastructure, optimization level, or additional service guarantees.

The average prices also highlight meaningful distinctions: mid-range models like `llama-4-maverick-17b` and `mistral-large-2512` tend to stabilize around $0.8–$1 per million tokens, reflecting a balance between performance and operational cost.

Overall, the table illustrates that pricing is not solely correlated with model size. Instead, it depends on a combination of provider-specific optimizations, hardware choices, and market positioning. This variability reinforces the importance of evaluating multiple providers when selecting a model for production workflows, especially in scenarios where token volume is substantial.

5.5 Cost-Performance Tradeoff

Having examined quality of the generations, execution time, token consumption, and pricing independently, we now integrate these dimensions to assess the overall economic efficiency of each model. The results reveal that no single metric is sufficient to characterize model efficiency. Instead, each model occupies a distinct position within a multidimensional tradeoff space.

The goal of this final analysis is to determine which model offers the best balance between computational cost, stability, and quality of the generated manifests, guiding model selection for real-world deployment.

`granite-4-h-small` consistently emerges as one of the most economically efficient models. It exhibits **fast execution times**, **low token consumption** and a **competitive price**. Despite being a reasoning-enabled model granite maintains a tight distribution and avoids the large spikes observed in other reasoning models. Its stability under both Simple and Structured intents further reduces the likelihood of costly retries. Overall, granite offers one of the **best cost-performance ratios** in the entire evaluation.

`gpt-oss-120b` shares many of granite's strengths, forming the second pillar of high-efficiency models. It has **high first-attempt success rate**, **fast execution time** and **moderate token usage**. This model is also the model most sensitive to intent structure in terms of precision and recall, but this semantic variability does not significantly impact its economic footprint.

`mistral-small-3.1-24b-instruct-2503` demonstrates a balanced and robust economic profile. It's **minimal sensitive to intent structure**, it has **low token consumption** and also a very **competitive price**. Its main limitation is the sharp degradation at high temperature (1.4), where only one generation succeeds over twenty tests. However, at realistic operating temperatures (0.1–0.7), mistral-small remains stable, predictable, and cost-efficient.

`mistral-large-2512` is the most semantically accurate model across all similarity metrics and temperatures. Economically, it occupies a middle position respect to the other models because it has **lower token usage than reasoning models** and **medium execution times**. Its main drawback is the slightly lower number of successful generations at some temperatures. This behavior increases the risk of failed generations, which in real deployments may translate into additional retries and higher cost. Nevertheless, when similarity and correctness are the primary objectives, mistral-large offers the **best quality-to-cost ratio**.

`llama-3.3-70b-instruct` is the model with the highest economic variability. It's **less stable in generation attempts**, more **sensitive to intent structure**, it has high token usage across tests and a very wide price range. Its sensitivity to both intent structure and temperature makes its economic footprint unpredictable. While it may offer advantages in complex reasoning tasks, these benefits come at a substantial monetary and computational premium. For cost-sensitive or latency-critical pipelines, llama-3.3-70B is the **least efficient option**.

`llama-4-maverick-17b-128e-instruct-fp8` occupies a balanced but not optimal position. It has **moderate execution times** but **high token usage** and **pricing in the mid-range**. Its performance is stable across temperatures and similarity metrics, but it does not excel in any single economic dimension. It is a reasonable choice when moderate cost and moderate performance are acceptable, but it is outperformed by granite, gpt-oss, and mistral-large in pure efficiency.

5.5.1 Overall Tradeoff Summary

Combining all these analyses, we can distill and simplify the key findings to present the results more clearly:

- ***Most cost-efficient models***:
  `granite-4-h-small`, `gpt-oss-120B`, `mistral-small-3.1-24B`
  These models minimize execution time, token usage, and monetary cost.
- ***Best quality-to-cost model***:
  `mistral-large-2512`
  Highest similarity scores with moderate cost.
- ***Least economically efficient model***:
  `llama-3.3-70B-instruct`
  High token usage, unstable behavior, and the largest price range.
- ***Middle-ground model***:
  `llama-4-maverick-17B`
  Balanced but not optimal in any dimension.

In practical deployments,where thousands of manifests may be generated daily, the differences observed here translate into substantial variations in operational cost, latency, and resource allocation. Selecting the appropriate model therefore requires balancing economic efficiency with the desired level of semantic accuracy and robustness.

Chapter 6

Comparative Evaluation

Following the presentation of our system architecture and the empirical evaluation of its performance, we now compare our approach with some of the methods discussed in **Chapter 3**. This chapter analyzes the degree to which our agent aligns with or diverges from existing solutions and assesses whether it offers measurable improvements over prior techniques. The goal is to situate our contribution within the research landscape and evaluate its effectiveness relative to established methodologies.

6.1 Academic Approaches

We now compare our proposal with the most closely related studies [19], [20], [2], [24], highlighting the major differences and the aspects in which our approach provides enhancements.

Table 6.1 outlines the key distinctions between prior work and our proposal. Below, we provide a detailed discussion of these differences and explain how our solution addresses the limitations identified in existing approaches.

A first fundamental distinction concerns the scope and **complexity of the generated configurations**. Existing works primarily focus on producing simple, single-file manifests, often limited to basic Kubernetes objects such as Pods, Deployments, or Services.

In contrast, our objective is to design and evaluate an agent capable of generating complete, multi-resource Kubernetes configurations, including interdependent components, namespaces, volumes, RBAC rules, and multi-file setups. To assess this robustness, our agent is explicitly tested across different levels of complexity, ensuring resilience even when handling large, heterogeneous manifests, an aspect not explicitly explored in prior studies.

Study	Manifest Complexity	Validation Strategy	Auto-correction	Human-in-the-Loop	K8s Validation
Angi et al.[19]	Low	Separated	None	Required	Not specified
Sacco et al.[20]	Low	Partially integrated	Partial	Required	Not specified
Beuter et al.[2]	Medium	Partially integrated	Partial	Required	Dry-run
Dubey et al.[24]	Low	Separated	None	Required	Dry-run
Our Proposal	**High**	**Completely integrated**	**Yes**	**None**	**Real application**

Table 6.1: Overview of Compared Approaches

A second key difference lies in how **validation** is incorporated in the study. Prior works generally perform syntax or semantic checks a posteriori, disconnected from the generation pipeline, meaning validation does not influence iterative refinement.

Our solution integrates syntax validation, semantic consistency, and scope consistency directly within the agentic workflow. Each validation step can trigger corrective actions, enabling the agent to refine the manifest until all checks are satisfied. This tightly coupled design ensures higher reliability and prevents error propagation across iterations.

Closely related to this point, we deliberately exclude **human-in-the-loop** interventions during the correction process. While some existing proposals rely on manual feedback to guide the refinement of the manifest, our goal is to evaluate how resilient a fully autonomous agent can be when equipped with a complete set of internal validation and correction mechanisms. This design choice allows us to measure the true effectiveness of the agentic pipeline without external assistance.

Regarding **autocorrection**, only two of the referenced studies implement a feedback loop capable of adjusting the generated manifest. The remaining works simply evaluate correctness after generation, without using validation results to improve the output. Our agent instead employs a more structured, multi-stage correction loop, where each failure triggers a targeted refinement step.

Finally, a crucial difference concerns **runtime validation** of Kubernetes functionality. All existing approaches rely on `kubectl --dry-run` to validate manifests. Although this command ensures that the manifest is syntactically well-formed and that individual objects conform to Kubernetes API schemas, it does not persist resources in the cluster. This limitation becomes problematic for complex configurations: for example, a namespace created via dry-run is not stored, so subsequent objects referencing it will fail to resolve dependencies. Dry-run validation is therefore insufficient for multi-resource or interdependent manifests.

To overcome this limitation, our agent performs real runtime validation by applying the generated manifest to a temporary cluster. This ensures that all objects are created, dependencies are resolved, and cross-resource references are validated in a real execution environment. By capturing errors that only emerge at runtime, our approach provides a significantly more reliable assessment of manifest correctness.

In the following sections, we present a detailed comparative analysis between the results of our evaluation and some techniques reported in these existing approaches to highlight their main differences and possible alternatives.

6.1.1 Fine-tuning Techniques

Sacco et al. (2025) [20] investigate how different fine-tuning strategies affect the quality of Kubernetes manifest generation, highlighting both performance gains and practical limitations. The authors note that fine-tuning is often **resource-intensive**, **time consuming**, and requires **large specific datasets**, conditions that are not always feasible in all real-world environments. Their pipeline employs a relatively lightweight autocorrection mechanism, limited to syntactic and semantic checks, and evaluates model performance using three metrics also adopted in our study: BLEU, Edit Distance, and KV-Wildcard Match.

Across all evaluated models, their results show a consistent trend: vanilla models perform worst, LoRA fine-tuned models improve, and full fine-tuned models achieve the highest scores. This confirms that fine-tuning is beneficial for the task of manifest generation.

Although we cannot reproduce their fine-tuning results on our dataset to enable a detailed comparison, we can still perform a qualitative assessment based on the results reported in their paper. Despite differences in datasets and experimental setups, their findings highlight substantial variations in absolute performance.

Table 6.2 indicates that, even without applying fine-tuning, our system achieves higher accuracy, lower variability, and greater stability across all metrics.

Metric	Their Approach	Our Approach
BLEU Score	0.35–0.45	0.48–0.56
Edit Distance	0.20–0.30	0.61–0.67
KV-Wildcard Match	0.15–0.50	0.83–0.89

Table 6.2: Comparison of min and max scores obtained between the fine-tuning approach and our vanilla method

The primary factor behind this is that a well-designed iterative correction mechanism can compensate for the absence of fine-tuning, achieving competitive results with the ranges reported by the fine-tuning approach. This suggests that, for structured tasks such as Kubernetes manifest generation, task-specific autocorrection pipelines may represent a more efficient and cost-effective alternative to traditional fine-tuning, especially when large curated datasets are not available.

6.1.2 Few-shot Learning Technique

Both *Angi et al. (2025)* [19] and *Dubey et al. (2024)* [24] investigate how structured prompting strategies, specifically few-shot prompting, affect the quality of generated configurations. This approach is significantly less expensive in terms of computation time compared to full fine-tuning, and for this reason their reported results are more directly comparable to our metrics.

In both studies, the generation process does not include any form of automatic self-correction loop; validation is performed only after generation, as a separate step. Our goal is to evaluate whether our autocorrection loop mechanism achieves comparable or superior outcomes relative to few-shot prompting, both in terms of overall success rate and in terms of manifest similarity performance.

Accordingly, we revise our agent pipeline to let the LLM generate the intent in a single pass, eliminating any iterative loop or consistency verification. A complete description of the pipeline and set up is reported in the **Appendix D**.

We then evaluate this simplified architecture by adapting the original prompt in three different configurations:

- ***zero-shot***: the baseline prompt, also used in the full agent experiments
- ***one-shot***: prompt augmented with a single intent–manifest example
- ***three-shot***: prompt augmented with three examples

We limit the few-shot setting to three demonstrations because prior work [19] reports performance degradation in some models when more than three examples are provided and also to limit the complexity of the prompts.

All subsequent results are obtained by evaluating three of the models used in earlier experiments (`gpt-oss-120b`, `granite-4-h-small` and `mistral-large-2512`), using the complete dataset of **ten intent–manifest reference pairs**. For consistency with the rest of this study, each model is tested with two distinct intents pair per scenario and an intermediate temperature of 0.7. Under these conditions, each model generates 20 manifests in total.

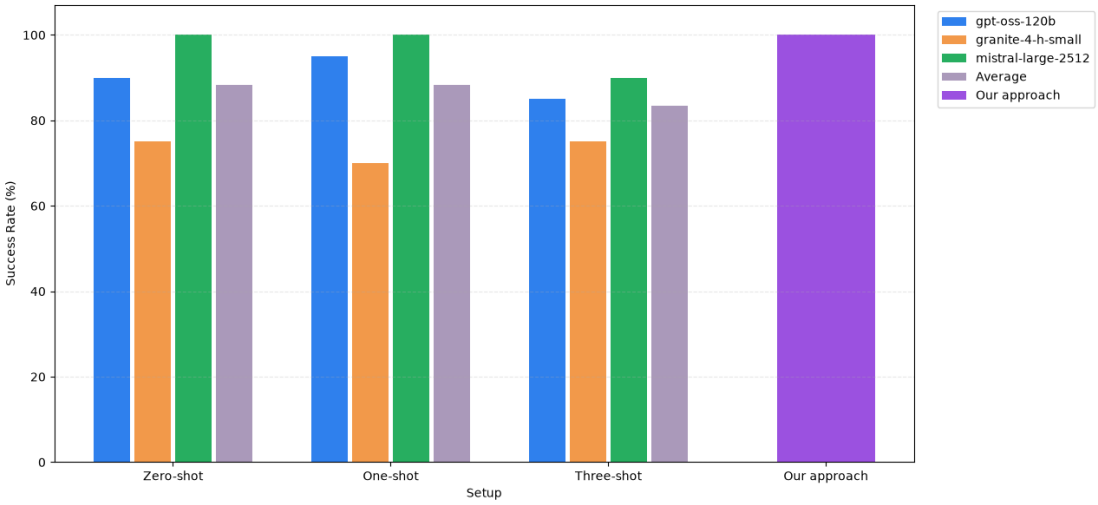


Figure 6.1: Success Rate Comparison Across Prompting Strategies

First of all we examine the rate of success of each model and comparing it with our original proposal. **Figure 6.1** summarizes the performance of the different models under the various prompting configurations and reports also the average rate between all the models.

A run is considered successful only when both validations—syntactic (via `yamllint`) and functional (via `kubectl apply`), as in the original pipeline—are passed. For our autocorrection-based approach, we report the aggregate results of the same three models used throughout the evaluation.

The success rate of the models using the simplified prompting-based method varies substantially across tests. Only a small subset of models consistently produces fully correct manifests, and overall the average success rate remains below 90%. This highlights a clear gap between direct few-shot prompting and the robustness provided by the autocorrection loop.

Increasing the number of examples does not consistently improve success rates. In several cases, the three-shot configuration performs on par with—or even worse than—the one-shot setup, confirming prior observations about diminishing returns in few-shot prompting. This trend is consistent with the hypothesis that longer prompts may introduce context overload for certain models, reducing their ability to maintain structural consistency across the generated manifest.

A further limitation of few-shot prompting concerns the **selection of the examples** to be included in the prompt. Because our generation target spans a wide and heterogeneous set of Kubernetes resources and user intents, a small number of demonstrations cannot adequately represent all possible scenarios. As a consequence, the examples included in the prompt may fail to provide the model with the structural cues required for certain intents. This constraint is intrinsic to few-shot methods and becomes particularly evident in broad or open-ended tasks; such approaches tend to be more effective when the target domain is narrow and representative examples can be chosen to fully capture its variability.

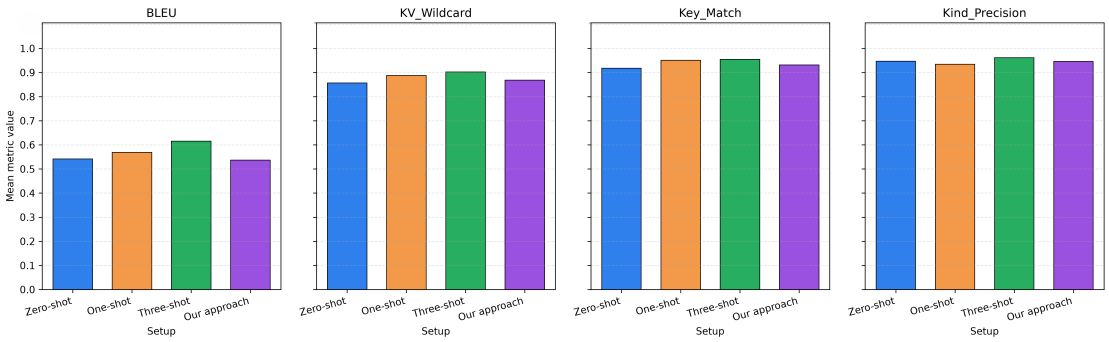


Figure 6.2: Manifest Similarity Comparison Across Prompting Strategies

Now we analyze whether the different prompting strategies lead to measurable differences in the **manifest similarity metrics**. We focus on the most informative metrics to avoid making the discussion unnecessarily heavy.

Figure 6.2 considers the aggregated results for each setup, without distinguishing between models. Overall, the differences across prompting configurations are limited. Surprisingly, the three-shot setup consistently performs slightly better than the others across all metrics, although the gap remains modest.

The largest variation appears in the BLEU score; however, as discussed in previous chapters, BLEU is strongly affected by the ordering of instructions and is therefore not the most reliable metric for structured configuration files.

We should note that our autocorrection approach could reasonably outperform the prompt pipeline, especially because its semantic consistency checks ensure that the generated manifest fully satisfies the user's request. This advantage should be particularly visible in YAML-aware metrics such as key-match and kv-wildcard-match. However, this improvement does not materialize, and a plausible explanation lies in the intrinsic behavioral differences between the two approaches. The autocorrection loop is designed to guarantee validity and semantic consistency with the user's intent. It generates and modifies the manifest until it becomes syntactically correct and semantically deployable. Few-shot prompting instead behaves differently: by exposing the model to one or more reference manifests—directly derived from the ground truth—it implicitly **biases the generation** toward reproducing the style, layout, and key structure of the examples. This imitation effect can naturally increases similarity scores.

Approach	**Avg Cost** (tokens)	**Success Rate** (%)
Zero-shot	1325.90	88.33
One-shot	1729.08	88.33
Three-shot	2591.20	83.33
Our Agent	**2786,90**	**98.67**

Table 6.3: Cost vs. Success Rate Comparison across prompting strategies

Finally, the results reported in **Table 6.3** offer a comprehensive view of the cost–quality trade-off across the evaluated prompting strategies. By jointly examining total token consumption (*input + output*) and success rate, we obtain a more informative assessment of each method's reliability and operational efficiency.

For our agent, in this comparative evaluation we consider only the **tokens used during the generation phase**. We intentionally exclude the tokens consumed in the initial prompt validation and in the final manifest verification. Including those additional steps would distort the comparison, since the other approaches rely on a single generation call, whereas our pipeline performs three. Restricting the analysis to the generation phase therefore allows a fairer and more interpretable comparison of how each strategy behaves when producing the manifest.

Even under this restricted accounting, we must acknowledge that the higher token usage of our agent is an inherent consequence of its design. Each invocation may require up to seven iterative attempts before yielding a valid manifest.

As a result, the measured cost reflects not a single generation—as in the other prompting strategies—but potentially multiple generation attempts. This leads to a higher token count, but it also explains the agent's markedly superior robustness.

Within this framework, our autocorrection agent clearly emerges as the most dependable approach. Despite its higher token expenditure, it achieves a success rate of nearly **99%**, outperforming all direct prompting baselines by a considerable margin. Conversely, few-shot prompting remains comparatively **inexpensive** but exhibits limited robustness. Its success rates consistently remain below 90%, and the three-shot configuration performs particularly poorly: it consumes the largest number of tokens among the prompting strategies while yielding the lowest success rate. In this case, the increased cost is not justified by the resulting performance.

Overall, the results highlight a clear trade-off: few-shot prompting offers **lower cost** but **limited robustness**, whereas our agent achieves **near-perfect correctness** at a higher computational cost. At the same time, manifest-similarity metrics tend to favor the n-shot strategies, since exposure to ground-truth examples encourages the model to mimic their structure and layout. Taken together, these results show that few-shot prompting optimizes for structural resemblance, while our agent prioritizes functional validity, making the choice between the two approaches dependent on whether cost efficiency or manifest correctness is the primary objective.

6.2 Alternative Frameworks

To complete our analysis, in addition to the academic methods discussed in **Section 6.1**, we also examine industrial tools that address the same problem through different design approaches. Among these, ***kagent*** is an emerging framework designed for the automatic management of Kubernetes clusters through modular AI agents. Although it is not part of the scientific literature, it represents a relevant benchmark for evaluating the maturity, completeness, and architectural choices of the system developed in this thesis.

6.2.1 Kagent

kagent [25] is an **open-source project** under the Cloud Native Computing Foundation (CNCF) that provides a Kubernetes-native framework for building **AI agents**. Its goal is to simplify the development, deployment, and management of AI-driven automation directly within Kubernetes clusters. The framework is designed to be easy to understand and use, while offering a flexible and extensible architecture for constructing agents that interact with cloud-native systems.

Core concepts

- **Agents** The fundamental building block of kagent. Each agent is defined as a Kubernetes Custom Resource (CRD) that specifies a system prompt, a set of tools, and an LLM configuration
- **LLM Providers** kagent supports multiple language-model providers, including OpenAI, Azure, Anthropic, Google Vertex AI, Ollama, and other custom providers accessible through AI gateways. Providers are represented through the `ModelConfig` resource
- **MCP Tools** Agents can connect to any MCP (Model Context Protocol) server exposing tools. kagent includes an MCP server offering tools for Kubernetes, Istio, Helm, Argo, Prometheus, Grafana, Cilium, and others. Each tool is defined as a Kubernetes Custom Resource and can be shared across multiple agents.

Design Characteristics A distinctive aspect of kagent is that both agents and tools are managed as Kubernetes Custom Resources using standard kubectl workflows. The framework is fully declarative: agents and tools are defined through YAML manifests, making the system highly extensible and adaptable to different AI-automation use cases. Users can rely on the built-in tools or define new ones directly within the cluster.

The typical use case for kagent involves monitoring and observing workloads in Kubernetes using AI agents. By leveraging specialized tools—or by creating new ones—kagent enables a unified and intelligent overview of cluster behavior, centralizing operational insights.

kagent is a relatively new framework and is currently under active development. Its rapid evolution and modular design make it an interesting point of comparison for evaluating the capabilities and design choices of the agentic system proposed in this thesis.

Architecture The high-level architecture of kagent consists of multiple components running inside and outside of Kubernetes cluster (Figure 6.3).

- **Controller** It's a Kubernetes components that knows how to handle custom CRDs for creating and managing AI agents in the cluster. It observes changes to resources and reconciles the desired state by creating or updating the corresponding runtime components.
- **Engine** The core component of kagent. It runs the agent's conversation loop and supports two runtimes: *Python ADK* (default), built on the Google ADK framework and integrating with CrewAI, LangGraph, and OpenAI; *Go ADK*, a lightweight implementation offering faster startup and lower resource usage.

- **CLI** A command-line interface that connects to the engine and allows users to manage resources and interact with agents.
- **Dashboard** A browser-based UI for managing and interacting with AI agents. It provides the simplest entry point for exploring kagent.

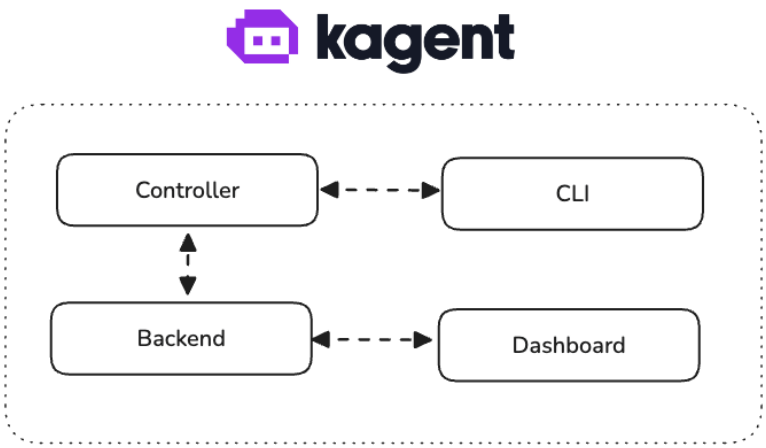


Figure 6.3: kagent Architecture

6.2.2 Comparative Analysis

In this section, we evaluate both the architectural differences and the resulting outputs produced by comparing kagent with our framework.

Design Differences

The choice to compare our agent and kagent framework is motivated by a shared capability: both approaches leverage large language models to interact with Kubernetes and has the possibility of create agents to generate configuration files in the form of YAML manifests. Despite this possible common goal, the two systems differ substantially in terms of architecture, specialization, and validation mechanisms.

The agent proposed in this thesis is specifically designed for Kubernetes manifest generation. Its execution flow is explicitly defined through a state graph and consists of several consecutive phases: request-scope checking, manifest generation, syntactic validation, operational validation, and semantic validation against the original intent. This design enables a controlled cycle of generation, verification, and correction, ensuring that the produced configuration is both valid and aligned with the requested functionality.

kagent, on the other hand, adopts a more general and Kubernetes-native architecture. The framework runs directly inside the cluster and uses Kubernetes resources to manage and observe the agents deployed within it. Unlike our system, which defines a single agent with a fixed role, kagent allows users to create custom agents tailored to specific needs by selecting tools, modifying prompts, and configuring the resources involved. This flexibility enables a broader range of use cases but also shifts responsibility for fine-grained behavior and validation to the user.

A key difference between the two approaches therefore lies in the level of control over the generation process. In our system, the custom pipeline for Kubernetes configuration generation explicitly governs all factors relevant to the correctness and functionality of the produced manifest. In contrast, kagent provides greater generality and tighter integration with Kubernetes, but detailed validation steps must be configured separately and may not be available with the same level of granularity.

Experimental Setup

To compare the two frameworks, we created a new agent within the kagent system specifically designed to generate Kubernetes configuration manifests to be then used with our dataset.

Among the available MCP kubernetes tools, two are particularly relevant for our evaluation:

- **Resource-generation tool** generates Kubernetes resources from a natural language description, aligning directly with the objective of our system
- **Apply-to-cluster tool** applies a YAML file to the cluster to assess its functionality within the target environment

For the experiments presented in the following sections, we configured a dedicated kagent agent that uses these tools. Although the generated manifest is applied to the cluster and potential errors are detected, this agent does not perform any iterative refinement: the generation process is strictly one-shot, with a single invocation of the LLM and no correction cycle.

To ensure a meaningful comparison, we use the same complete dataset described in **Chapter 5.1**: 10 different scenarios, each one with 2 different intent descriptions. All experiments are executed on the same Kubernetes cluster environment and under identical runtime conditions.

Regarding the LLMs used in those experiments, we selected two distinct models: `gpt-oss-120b`, already used in our agent's evaluation, and `gemma4:31b`. This choice allows us to include both a larger model and a smaller one, producing a set of results that is more directly comparable to those obtained with our agent.

The benchmark metrics reported for our agent correspond to the aggregated results across all previously tested models, considering only a temperature of 0.7.

Results Comparison

A first difference that immediately emerges when executing the agent created within kagent is the presence of an intermediate **clarification phase**. After receiving the initial intent, and before invoking the resource-generation tool, the agent frequently asks the user a set of follow-up questions related to various aspects of the requested configuration. Examples include the container image to be used in a Deployment, the desired database version, or the number and type of replicas required for a specific resource.

This intermediate step clearly improves the correctness of the generated manifest, as it guides the model toward a more precise interpretation of the user's intent. It also represents a valuable design insight for future improvements to our agent: incorporating a lightweight clarification stage could enhance performance while preserving the degree of generative autonomy that characterizes our approach.

A first evaluation step focuses on determining whether the configurations generated by kagent are valid both **syntactically** and from a **Kubernetes runtime** perspective. All manifests produced by kagent were found to be syntactically correct; however, the same level of success does not hold when these manifests are applied to a real cluster using the kubectl apply tool. When executed against a running cluster, only 80% of the generated manifests are accepted and instantiated correctly. The remaining **20% fail** due to issues such as invalid values, incompatible selectors, or resource-specific constraints that only emerge at runtime.

These results provides an important early indicator of the difference between integrating validation directly inside the agentic generation loop and performing validation only after generation. In our previous experiments with our agent, where validation is fully embedded into the iterative refinement pipeline, 98.7% of all generated manifests successfully pass the complete validation workflow, including syntax checks, semantic consistency, and real cluster application.

The contrast between those results highlights that agent-integrated validation loops significantly increase reliability, enabling the system to detect errors early and refine the manifest iteratively. By doing so, the agent becomes capable of autonomously correcting structural inconsistencies and ensuring that the final output aligns with real Kubernetes execution semantics.

Metric	kagent Results	Our Agent Results
BLEU	0.514	0.523
CodeBLEU	0.592	0.600
Edit Distance	0.686	0.684
KV Match	0.624	0.635
KV-Wildcard Match	0.841	0.863
Key-Match	0.888	0.917
Kind Precision	0.922	0.951

Table 6.4: Comparison of similarity metrics between manifests generated by kagent and our agent

Considering only the manifests that successfully pass Kubernetes runtime validation in the previous step of the analysis, we now examine their correctness with respect to the reference configurations in our dataset, and compare these results with those obtained using our agent.

Table 6.4 reports a selection of the most significant similarity metrics introduced and described in **Chapter 5.3.1**.

The comparison shows a substantial equivalence between the results obtained with kagent and those produced by our agent. This is confirmed by the textual and syntactic similarity metrics, as indicated by the close values of **BLEU** and **CodeBLEU**. However, our agent achieves slightly higher scores in the structural and YAML-aware metrics, particularly **Key-Match** and **Kind-Precision**. These improvements suggest a more accurate preservation of the keys and resource types present in the reference manifests, as well as a reduced tendency to introduce extra or unintended resources.

Despite these differences, the gap between the two systems remains small. Therefore, in terms of manifest correctness, the two approaches can be considered broadly equivalent.

Discussion

Overall, the comparison shows that kagent and our agent achieve comparable results in terms of similarity between the generated manifests and the reference configurations present in the dataset. kagent benefits from a more general architecture and from its native integration with Kubernetes, whereas our system is more specialized and explicitly designed to verify and refine generated manifests. This specialization results in a considerably higher runtime success rate and slightly better structural similarity scores.

These findings suggest that the main advantage of our approach does not lie in producing radically different manifests—as also observed in the comparative analysis with few-shot learning techniques—but rather in increasing the reliability of the generation process. Therefore, kagent represents a flexible general-purpose framework, while our agent provides a more controlled and reliable solution for the specific task of Kubernetes manifest generation.

Chapter 7

Conclusions

The increasing complexity of cloud-native environments has made the configuration and management of modern infrastructures a challenging task, especially when applications consist of multiple interdependent services and resources.

Kubernetes provides the flexibility and scalability required to operate such systems, but its declarative configuration model also introduces a significant level of complexity. Designing correct manifests requires knowledge of Kubernetes objects, dependencies, networking, storage, and operational constraints, while even small configuration errors can prevent successful deployment or lead to unexpected and dangerous behavior.

In this context, Large Language Models have emerged as a promising technology for simplifying configuration tasks by translating natural-language requirements into structured artifacts. However, the experiments and analyses presented throughout this thesis show that the ability to generate plausible YAML files is not sufficient to guarantee reliable Kubernetes configurations. A generated manifest may be syntactically correct but incompatible with a Kubernetes cluster, or it may be successfully deployed while still failing to satisfy the user's requirements. The main challenge is therefore not only generation itself, but the ability to integrate generation, validation, and correction into a coherent and reliable process.

This thesis addressed this problem by proposing an intent-driven agentic approach for the automatic generation of Kubernetes manifests, demonstrating that such an approach makes the process more reliable and less dependent on manual intervention. The core idea was to move beyond single-shot generation and treat configuration creation as an **iterative process** in which the generated result is continuously evaluated and refined. The objective was therefore not simply to generate Kubernetes manifests, but to design a workflow capable of progressively improving them until they satisfy multiple levels of correctness.

This iterative mechanism proved to be the key element of the proposed architecture. The experimental results show that the main advantage of the agent does not merely consist in generating configurations that are more similar to a reference manifest, but in substantially increasing the probability of obtaining a configuration that is actually valid and usable. By integrating validation into the generation process, the agent is able to detect errors that would otherwise remain unnoticed and autonomously correct them through successive iterations. The resulting workflow shifts the role of the LLM from a simple generator to a component of a controlled decision-making process.

The experiments conducted on a heterogeneous set of Kubernetes workloads confirm the effectiveness of this design. Across the evaluated scenarios, the proposed agent achieved a average **success rate** consistently higher than 91%, demonstrating a high degree of reliability in producing manifests that pass the complete validation workflow. This result is particularly relevant when compared with direct prompting strategies. Few-shot prompting generation achieved a success rate between 83% and 88%, and, although these approaches in some cases require fewer tokens and are therefore computationally cheaper, they do not provide the same ability to detect and correct errors during generation.

A further result concerns the **choice of the underlying LLM**. The evaluation showed that model characteristics have a significant impact on the behavior of the agent, and that there is no single model that is optimal across all dimensions. Larger models such as `mistral-large-2512` and `gpt-oss-120b` generally provided strong generation quality and stable behavior, while smaller models such as `granite-4-h-small` and `mistral-small-3.1-24b` offered more favourable economic profiles. Among the evaluated models, `mistral-large-2512` consistently achieved the highest similarity scores and the lowest variability, making it the strongest candidate when correctness and quality are the primary objectives. At the same time, `granite-4-h-small` and `gpt-oss-120b` emerged as particularly efficient alternatives when execution time, token consumption, and cost are given greater importance.

The **economic evaluation** also provides an important insight into the practical applicability of the proposed approach. The higher reliability achieved by the agent comes at the cost of increased token consumption, and consequently of price, since each validation and correction cycle introduces additional model interactions. The experiments therefore show that the benefits of the agent should not be evaluated in isolation from its computational cost. Instead, the choice of model and workflow should depend on the specific requirements of the target scenario, balancing correctness, robustness, latency, and economic efficiency.

Overall, the results obtained in this thesis confirm that an agentic architecture can provide a **more reliable alternative** to conventional single-shot approaches for Kubernetes manifest generation. The experiments show that the effectiveness of an LLM-based configuration system depends not only on the capabilities of the underlying model, but also on the mechanisms used to control, validate, and refine its output. By combining these components into a unified workflow, the proposed approach demonstrates that **general-purpose LLMs can be used to address a highly structured and domain-specific generation task** without relying exclusively on fine-tuning or large domain-specific training datasets.

7.1 Future Works

Although the results obtained are promising, there remain several improvements and analyses that could further enhance the completeness, reliability, and efficiency of the generation process. A limitation to consider is that the results should be interpreted within the scope of the experimental setup, which strongly influences the outcomes. The evaluation was conducted on a limited dataset of Kubernetes scenarios, and although security was identified as an important requirement in the analysis of existing approaches, the current implementation does not yet include a dedicated security-validation stage. These limitations do not undermine the main findings of the work, but they highlight relevant areas in which the proposed architecture can be further extended.

A first possible extension is the introduction of a lightweight **clarification stage** before manifest generation. The comparison with kagent showed that asking the user for additional information when the initial intent is ambiguous can improve the precision of the generated configuration. Integrating such a mechanism into the current architecture could reduce the number of assumptions made by the model while preserving the autonomous nature of the agent.

A second direction concerns the extension of the validation pipeline with dedicated **security checks**. Future versions of the system could integrate security validators capable of detecting unsafe defaults, insecure or vulnerable configurations, and violations of Kubernetes security best practices. This would allow security to become an explicit part of the acceptance criteria, together with syntactic, runtime, and semantic correctness.

Security considerations should also include an assessment of whether the agent itself is secure and not vulnerable to adversarial attacks. This aspect was only briefly explored and could not be fully developed due to time constraints.

Another important direction is the **expansion of the evaluation dataset**, ideally using resource sets and configurations that are actually employed in production environments and therefore validated beyond simple correctness against a test cluster. A larger benchmark including more complex workloads, additional Kubernetes resources, and stronger inter-resource dependencies would provide a broader evaluation of the generalization capabilities of the proposed approach. This would also make it possible to investigate more systematically how configuration complexity affects the number of correction iterations, execution time, and overall cost.

Finally, future work could investigate **adaptive strategies for model and prompt selection**. Since the experiments showed that different models behave differently depending on the characteristics of the input and the generation settings, the agent could dynamically select the most appropriate model, temperature, or prompting strategy for each request. For example, smaller models could be used for simple configurations or patches, while larger models could be employed for more challenging requests. Such an extension could further improve the cost-performance trade-off while maintaining the reliability achieved by the current architecture.

Appendix A

Model-Specific Prompt Set Used in the Architecture

This appendix provides an example of the actual prompts used in the system, illustrating their structure and constraints. Because the full prompt set is extensive, we include only one model-specific instance, specifically the one used with the `granite-4-h-small` model, which is sufficient to demonstrate the constraints and behaviors adopted in our experiments.
Each prompt corresponds to a distinct node in the workflow (scope validation, manifest generation, semantic validation), and reflects the modular and model-adapted design described in **Section 4.3**.

```
You are an AI assistant acting as a security and scope validation gate for a Kubernetes
    YAML generator.

Instructions:
Evaluate whether the user request is specifically related to Kubernetes manifests,
    Kubernetes resources, Helm charts, deployment configuration, container orchestration,
     or closely related infrastructure configuration tasks.
  - Return only one of the following responses:
     VALID
     INVALID: <reason>
  - Do not provide explanations, reasoning, markdown, examples, or additional text.
  - Reject requests unrelated to Kubernetes or deployment configuration.
  - Reject requests involving malware, exploits, viruses, credential theft, destructive
    activity, or harmful content unrelated to Kubernetes administration or deployment.
  - Keep invalid reasons concise but descriptive.
  - Never output anything other than the allowed response formats.
  - Accept list of user needs for an application, even if they are not clearly expressed in
    Kubernetes terms, as long as they can be translated into a valid Kubernetes manifest.
```

Listing A.1: Scope Consinstency Prompt for Configuration Intents Validation

```
You are an AI assistant specialized in generating Kubernetes resource manifests.

Instructions:
  – Output only valid Kubernetes YAML manifests.
  – Do not include explanations, reasoning, markdown fences, or comments.
  – Do not start the response with ''' or '''''.
  – If multiple Kubernetes resources are required, separate them using ———.
  – Ensure manifests are syntactically correct and production–ready when possible.
  – Use appropriate apiVersion, kind, metadata, and spec fields for all resources.
  – Never output anything other than raw YAML.
```

Listing A.2: Manifest Generation Prompt for Kubernetes Configuartions

```
You are an AI assistant acting as a semantic validation engine for Kubernetes YAML
    manifests.

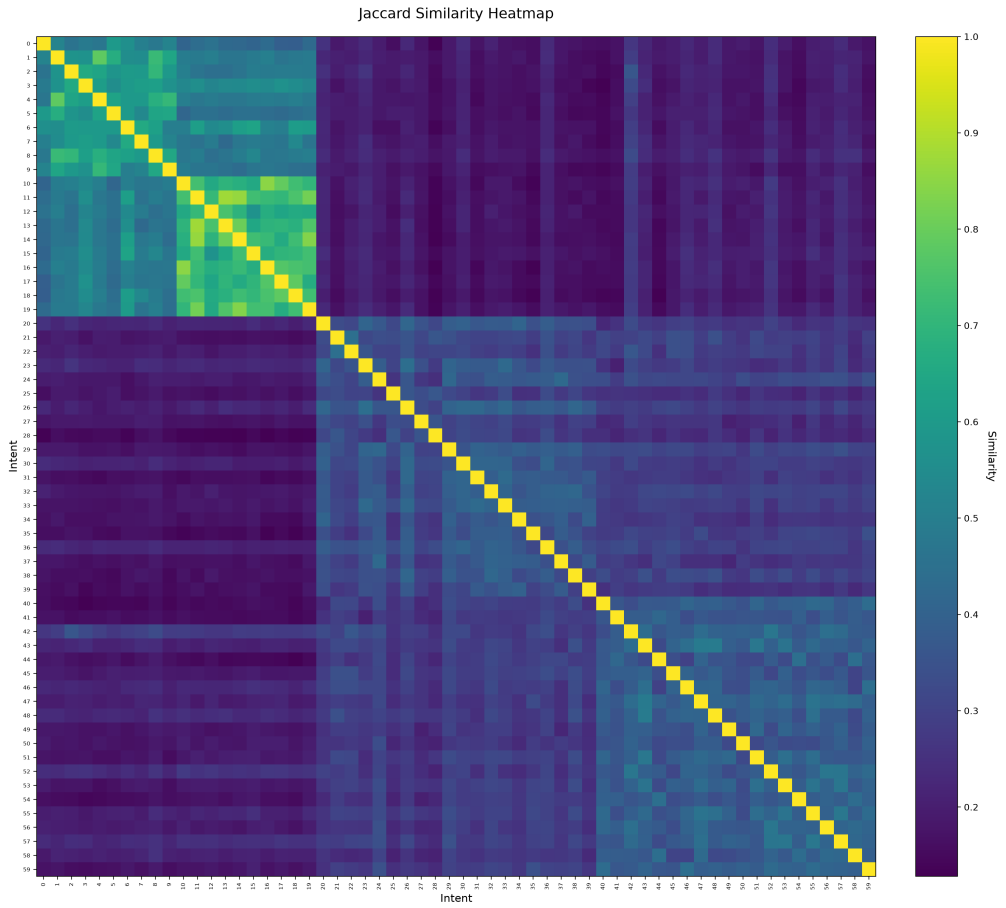
Instructions:
  – Compare the user request against the provided Kubernetes YAML manifest.
  – Determine whether the manifest semantically satisfies the requested task,
   functionality, resource types, configuration intent, and all explicit constraints.
  – Validate that the YAML resources align with the requested Kubernetes behavior and
   deployment requirements.
  – Return only one of the following responses:
      VALID
      INVALID: <reason>
  – Do not provide explanations, reasoning, markdown, examples, or additional text.
  – Mark the result as invalid if:
      Required Kubernetes resources are missing.
      Resource types do not match the request.
      Explicit user constraints are not satisfied.
      The manifest configuration contradicts the requested behavior.
      The YAML is incomplete, inconsistent, or semantically unrelated to the request.
  – Keep invalid reasons concise but descriptive.
```

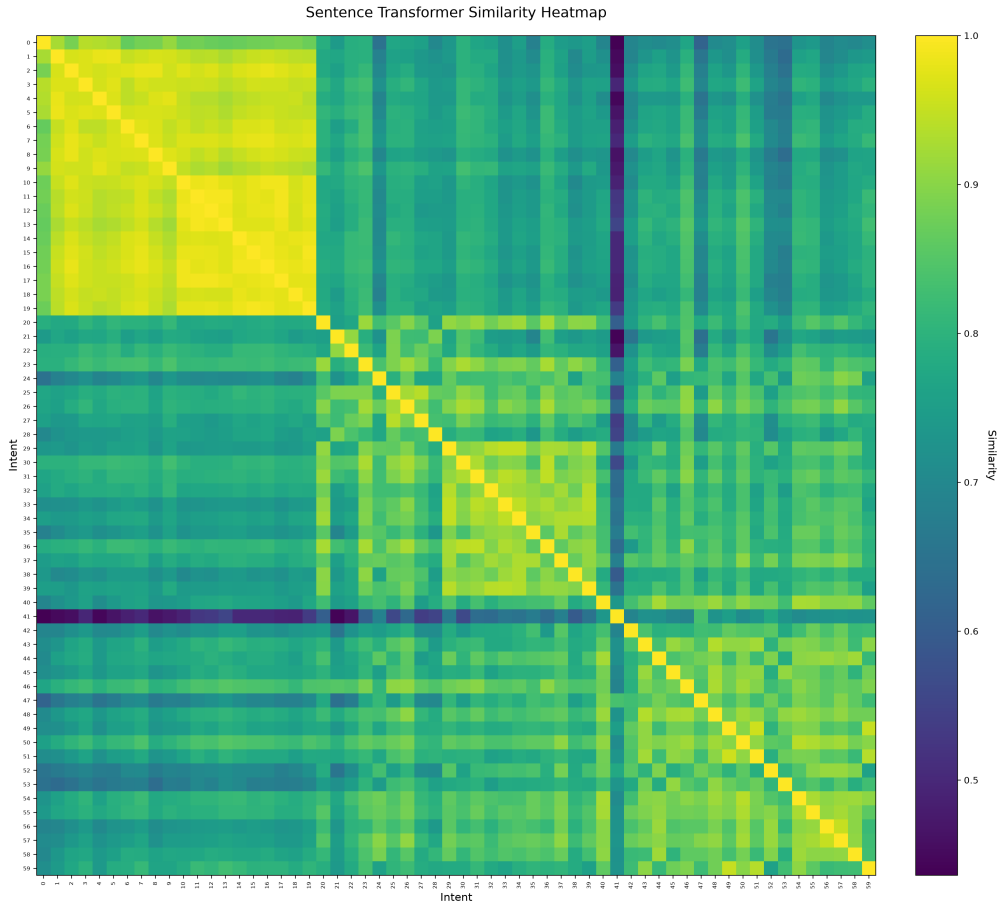
Listing A.3: Semantic Validation Prompt for Kubernetes YAML Manifests

Appendix B

Intents Similarity Heatmaps

This appendix presents the complete set of heatmaps generated for the intent similarity analysis, focusing specifically on the `db-cronjob-example`. These visualizations complete the discussion in **Section 5.2**.





The matrices are symmetric because the generated intents (60 for each scenario) are arranged in the same order along both axes. Lighter colors indicate higher similarity values, while darker colors correspond to lower similarity.
This visual representation makes it easier to identify regions of strong semantic cohesion as well as areas with more heterogeneous intent formulations, complementing the quantitative results obtained through metrics such as Jaccard Similarity and Cosine Similarity.

Appendix C

Number of Attempts - Simple vs Structured Intents

This appendix presents two supplementary plots that support the analysis discussed in **Section 5.4.1** about the average number of attempts per model.

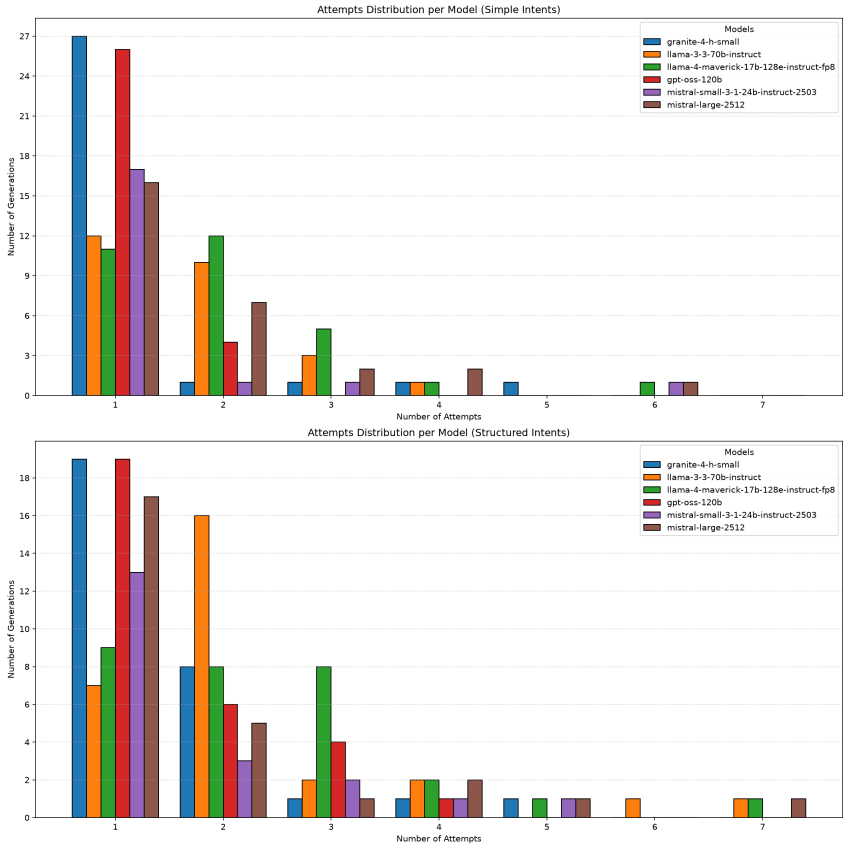


Figure C.1: Comparison of number of attempts per model with Simple and Structured Intents across models

Appendix D

Pipeline Details for N-Shot Prompting Generation

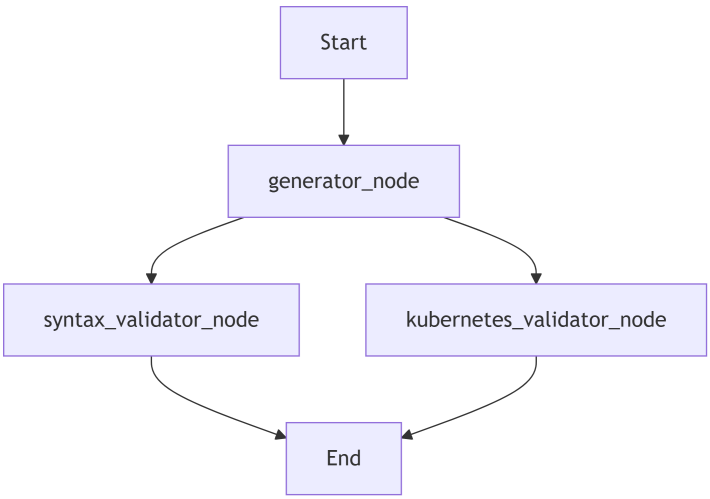


Figure D.1: Representation of the Pipeline Used to Generate Manifests via N-Shot Prompting Strategies

Figure D.1 reports the architecture used to evaluate the performance of manifest generation under zero-shot, one-shot, and three-shot prompting strategies (**Section 6.1.2**). We selected three example–intent pairs with increasing complexity: a simple case involving two Kubernetes objects, a medium-complexity scenario, and a more elaborate one designed to better stress-test the LLM. All examples were derived from the ground-truth dataset used throughout the evaluation.

The first node of the pipeline performs the generation step, where the original task description is augmented with the corresponding set of examples. Once the manifest is generated, it is sent to both validator nodes independently to assess whether the output is correct. The pipeline then terminates.

Bibliography

[1] *Kubernetes Documentation*. The Kubernetes Authors. URL: https://kubernetes.io/docs/home/ (visited on 07/04/2026) (cit. on p. 4).

[2] Niklas Beuter, André Drews, and Nane Kratzke. «Prompt-Driven and Kubernetes Error Report-Aware Container Orchestration». In: *Future Internet* 17.9 (2025). ISSN: 1999-5903. DOI: 10.3390/fi17090416. URL: https://www.mdpi.com/1999-5903/17/9/416 (cit. on pp. 7, 15, 61, 62).

[3] *LLM Foundations Description*. Microsoft. URL: https://learn.microsoft.com/en-us/agent-framework/journey/llm-fundamentals (visited on 07/22/2026) (cit. on p. 8).

[4] Caglar Gulcehre et al. *Reinforced Self-Training (ReST) for Language Modeling*. 2023. arXiv: 2308.08998 [cs.CL]. URL: https://arxiv.org/abs/2308.08998 (cit. on p. 9).

[5] *AI Agents Description*. IBM. URL: https://www.ibm.com/think/topics/ai-agents (visited on 07/08/2026) (cit. on p. 10).

[6] Ranjan Sapkota, Konstantinos I. Roumeliotis, and Manoj Karkee. «AI Agents vs. Agentic AI: A Conceptual taxonomy, applications and challenges». In: *Information Fusion* 126 (2026), p. 103599. ISSN: 1566-2535. DOI: https://doi.org/10.1016/j.inffus.2025.103599. URL: https://www.sciencedirect.com/science/article/pii/S1566253525006712 (cit. on p. 10).

[7] Tula Masterman, Sandi Besen, Mason Sawtell, and Alex Chao. *The Landscape of Emerging AI Agent Architectures for Reasoning, Planning, and Tool Calling: A Survey*. 2024. arXiv: 2404.11584 [cs.AI]. URL: https://arxiv.org/abs/2404.11584 (cit. on p. 10).

[8] LangChain. *LangChain Framework*. Accessed: 2026-07-22. URL: https://www.langchain.com/ (cit. on p. 11).

[9] LiteLLM. *LiteLLM Documentation*. Accessed: 2026-08-03. URL: https://docs.litellm.ai/ (cit. on p. 13).

[10] Sathvik Joel, Jie JW Wu, and Fatemeh H. Fard. *A Survey on LLM-based Code Generation for Low-Resource and Domain-Specific Programming Languages*. 2025. arXiv: 2410.03981 [cs.SE]. URL: https://arxiv.org/abs/2410.03981 (cit. on p. 14).

[11] David Delgado, Lola Burgueño, and Robert Clarisó. *A framework for assessing the capabilities of code generation of constraint domain-specific languages with large language models*. 2026. arXiv: 2603.05278 [cs.SE]. URL: https://arxiv.org/abs/2603.05278 (cit. on p. 14).

[12] Bailin Wang, Zi Wang, Xuezhi Wang, Yuan Cao, Rif A. Saurous, and Yoon Kim. *Grammar Prompting for Domain-Specific Language Generation with Large Language Models*. 2023. arXiv: 2305.19234 [cs.CL]. URL: https://arxiv.org/abs/2305.19234 (cit. on p. 14).

[13] Gordon Owusu Boateng et al. «A Survey on Large Language Models for Communication, Network, and Service Management: Application Insights, Challenges, and Future Directions». In: *IEEE Communications Surveys & Tutorials* 28 (2026), pp. 527–566. DOI: 10.1109/COMST.2025.3564333 (cit. on p. 14).

[14] Antonino Angi, Alessio Sacco, and Guido Marchetto. «LLNet: An Intent-Driven Approach to Instructing Softwarized Network Devices Using a Small Language Model». In: *IEEE Transactions on Network and Service Management* 22.4 (2025), pp. 3403–3418. DOI: 10.1109/TNSM.2025.3570017 (cit. on p. 14).

[15] Saurabh Pujar et al. *Automated Code generation for Information Technology Tasks in YAML through Large Language Models*. 2023. arXiv: 2305.02783 [cs.SE]. URL: https://arxiv.org/abs/2305.02783 (cit. on p. 14).

[16] Taher A. Ghaleb and Dulina Rathnayake. *Can LLMs Write CI? A Study on Automatic Generation of GitHub Actions Configurations*. 2025. arXiv: 2507.17165 [cs.SE]. URL: https://arxiv.org/abs/2507.17165 (cit. on p. 14).

[17] Antonio D'Ambrosio, Paolo Platter, Martina Salis, Francesco Simbola, Diego Reforgiato Recupero, and Daniele Riboni. «Lightweight Semantic Similarity Search of Data Products». In: *IEEE Access* 13 (2025), pp. 180647–180661. DOI: 10.1109/ACCESS.2025.3622752 (cit. on p. 14).

[18] Yifei Xu et al. *CloudEval-YAML: A Practical Benchmark for Cloud Configuration Generation*. 2023. arXiv: 2401.06786 [cs.DC]. URL: https://arxiv.org/abs/2401.06786 (cit. on pp. 15, 36).

[19] Antonino Angi, Liubov Nedoshivina, Alessio Sacco, Stefano Braghin, and Mark Purcell. «A Perspective on LLM Data Generation with Few-shot Examples: from Intent to Kubernetes Manifest». In: *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*. Ed. by Georg Rehm and Yunyao Li. Vienna, Austria: Association for Computational Linguistics, July 2025, pp. 345–354. DOI: 10.18653/v1/2025.acl-industry.27. URL: https://aclanthology.org/2025.acl-industry.27/ (cit. on pp. 15, 61, 62, 64, 65).

[20] Alessio Sacco, Cristian Zilli, and Guido Marchetto. «Intent-Based Kubernetes Configuration via LLMs: Current Trends and Open Challenges». In: *2025 IEEE 50th Conference on Local Computer Networks (LCN)*. 2025, pp. 1–7. DOI: 10.1109/LCN65610.2025.11146359 (cit. on pp. 15, 61–63).

[21] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. «Bleu: a Method for Automatic Evaluation of Machine Translation». In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Ed. by Pierre Isabelle, Eugene Charniak, and Dekang Lin. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, July 2002, pp. 311–318. DOI: 10.3115/1073083.1073135. URL: https://aclanthology.org/P02-1040/ (cit. on p. 36).

[22] Shuo Ren et al. *CodeBLEU: a Method for Automatic Evaluation of Code Synthesis*. 2020. arXiv: 2009.10297 [cs.SE]. URL: https://arxiv.org/abs/2009.10297 (cit. on p. 37).

[23] *Artificial Analysis*. Artificial Analysis. URL: https://artificialanalysis.ai/ (visited on 07/30/2026) (cit. on p. 57).

[24] Anuj Dubey, Charu Priya Singh, and Deepak Nadig. «Leveraging Large Language Models for Intent-Based Generation of Cloud-Native Configurations». In: *2024 IEEE International Conference on Advanced Networks and Telecommunications Systems (ANTS)*. 2024, pp. 1–6. DOI: 10.1109/ANTS63515.2024.10898244 (cit. on pp. 61, 62, 64).

[25] *kagent.dev*. CNCF. URL: https://kagent.dev/ (visited on 09/01/2026) (cit. on p. 68).