From Intent to Manifest: Evaluation of LLM-Driven Generation of Custom Resources in Kubernetes

Lucia Ferrari†, Mattia Zaccarini†, José Santos*, Filippo Poltronieri†,
Mauro Tortonesi†, Cesare Stefanelli†, Filip de Turck*
† Distributed Systems Research Group, University of Ferrara, Ferrara, Italy
* Ghent University - imec, IDLab, Department of Information Technology, Gent, Belgium

***Abstract*—Large Language Models (LLMs) have recently emerged as powerful enablers of intent-driven networking and automated service management in 5G and beyond. The aim of this work is to use LLMs to translate high-level intents into reconfiguration commands, operational support actions, or deployment decisions. Recent works have highlighted the feasibility of LLM-driven management and configuration, but these approaches operate primarily at the control and management plane level. The translation of orchestration intents into valid, system-specific configuration files remains an open challenge, particularly in cloud-native environments where declarative manifests are essential for automation. Therefore, this paper investigates the use of LLMs for generating Diktyo AppGroup Custom Resources (CRs) in Kubernetes (K8s) environments. Diktyo enables network-aware placement via grouping of workloads in K8s, but its adoption is hindered by the complexity and verbosity of the AppGroup configuration schema. Our approach leverages natural language and structured intents provided by cloud developers or operators, which are translated by an LLM into syntactically correct and meaningful AppGroups specifications. Results show that LLMs can bridge the gap between user-level orchestration intents and deployable AppGroup specifications, ensuring schema compliance and improving deployment efficiency. Models such as `qwen2.5-coder:14b` and `gpt-oss:20b` can correctly generate AppGroup CRs in most cases, achieving between 93 – 100% accuracy, significantly reducing the cognitive load for developers. This work advances the vision of an autonomous, intent-driven cloud-native orchestration, by extending the paradigm into network-aware orchestration in K8s ecosystems.**

***Index Terms*—Large Language Models, Kubernetes, Intent-Based Networking, Network Management.**

I. Introduction

The rapid and continuous evolution of network technologies and distributed systems has had a significant impact on network and service management methodologies [1], [2]. Traditional rule-based and heuristic-based approaches, while effective in handling simple and static network management problems, have shown clear limitations in highly dynamic and complex environments [3]. Therefore, the network and service management research community has increasingly adopted Artificial Intelligence (AI) and Machine Learning (ML) methodologies, with Deep Reinforcement Learning (DRL) [4]–[6] and Transformers [7], [8] emerging as prominent solutions. Despite their success, these approaches often struggle with unstructured data [9], such as network configuration intents, system logs, and multimodal data, which are very common in real-world scenarios [10], [11]. A key limitation lies in their limited capacity to understand contextual information and relationships among heterogeneous data sources, which can result in suboptimal decision-making in dynamic environments.

The recent surge of Generative Artificial Intelligence (GenAI) and, more specifically, Large Language Models (LLMs) in both industry and academia has introduced unlimited new research prospects and challenges in various research fields, such as Radio Access Networking [12], Industry 5.0 [13], Recommendation Systems [14], and Autonomous Driving [15]. The main reason behind this increase lies in their ability to deal with unstructured data, especially natural language [16]. In fact, LLMs are capable of efficiently processing and reliably generating human-like text, understanding complex and structured instructions, and making decisions according to input prompts or data patterns recognized during training [17]. Given these abilities and the huge amount of code publicly available, LLMs have seen an ever increasing employment in a variety of different coding tasks [18], [19].

Based on these principles, previous works have leveraged LLMs to create self-healing cloud and deployment solutions in Kubernetes (K8s) [20], [21]. This paper plans to take this research one step further, analyzing LLMs potential in creating K8s Custom Resources (CRs), integrating their functionalities to mitigate human-related drawbacks in the adoption of Diktyo. Diktyo[1] is a network-aware scheduling framework designed for K8s environments [22]. Thanks to the plugins provided, Diktyo can minimize the end-to-end (E2E) latency of their applications according to service dependencies and the infrastructure topology where containers will be deployed. However, to make Diktyo work properly, adopters have to manually provide a detailed CR manifest, including the definition and eventual dependencies of each service, topology characteristics, the sorting algorithm for the desired scheduling order, and workload details. This manual configuration process is not only time-consuming but also prone to human errors. Moreover, since the manifest structure required by Diktyo is relatively rigid, automating this process would greatly enhance its usability, reliability, and adaptability.

Therefore, this work aims leverages the potential of LLMs to automatically generate suitable Diktyo AppGroup CRs,

[1] https://github.com/diktyo-io

thereby reducing deployment time and minimizing human-induced errors in the process. Specifically, this paper demonstrates LLM capabilities in generating K8s manifests and evaluates the performance of several widely adopted open-source LLMs in handling requests of varying sizes using a few-shot prompting approach.

The main contributions of the paper are the following:

- **LLM-based generation of K8s CRs**: Given the structured nature of K8s CR manifests, this work considers the use of LLMs to automate their generation, thereby improving autonomy and reducing human errors. To this end, this manuscript takes into account Diktyo, a network-aware scheduler for K8s officially approved in the K8s sig-scheduling repository[2], as a CR reference.
- **Comparison between natural language and template-based prompts**: To our knowledge, prior works primarily rely on natural language prompts to assess LLM performance, taking advantage of LLMs suitability for understanding and generating human language. In contrast, this study compares both natural language and template-based prompting approaches to evaluate model adaptability to different input structures.
- **Extensive LLMs comparison**: The experimental evaluation presented in this work includes multiple state-of-the-art open-source LLMs, such as the recent GPT-OSS by OpenAI and Granite by IBM, both publicly available in Hugging Face [23]. This enables a detailed comparison of the performance of different LLM models in generating K8s CR manifests.

The remainder of the paper is organized as follows: the state-of-the-art on LLMs evolution and usage in different use cases is described in the next section. Sec. III highlights the importance of including LLMs in the creation of K8s CRs, describing the functionalities of Diktyo in K8s and the potential contribution of LLMs. Sec. IV details the methodology adopted, describing the choice to employ different request formats and the evaluation pipeline designed. Sec. V describes the evaluation setup, followed by the results in Sec. VI. Sec. VII concludes the paper.

II. Background & Related Work

Existing works have demonstrated the potential of LLMs for intent-based networking and network management, where high-level user intents are translated into reconfiguration commands, operational support actions, or Virtual Network Function (VNF) deployment strategies. Fontana et al. [24] presented an LLM-powered chatbot capable of reconfiguring 5G modems based on user intents, reducing the burden on network operators. Gowtham et al. [25] introduced GraphGPT, an LLM-driven management framework that leverages graph reasoning to support intent-based control of next-generation networks. Mekrache et al. [26] extended this paradigm to operational support systems, proposing OSS-GPT, which integrates LLMs into 6G management workflows. In addition, Nam et al. [27] showcased the applicability of LLMs for cloud environments, demonstrating how VNF deployment in OpenStack can be automated by intent translation. The authors in [28] introduced ECO-LLM, a middleware for Compute Continuum optimization that uses LLM to dynamically place microservices according to changes in the application workload. The use cases analyzed demonstrated the significant potential of ECO-LLM in generating, integrating, and applying code on-the-fly without manual intervention. Finally, [29] proposed Kubeteus, a framework for the autonomous generation of network policies designed for cloud-native environments. Through Kubeteus, Kim et al. leverage natural language processing techniques and software components, such as the fine-tuning automator, policy validator, and policy enforcer, to produce reliable network policies and, at the same time, continuously fine-tune the model according to the intent of the administrator and the current state of the cluster. The experimental evaluation showed the significant potential of Kubeteus, with fine-tuned LLMs heavily outperforming their related baseline. These efforts have primarily focused on both control- and management-plane tasks in 5G/6G networks or cloud environments, while leaving the translation of complex orchestration intents into deployable, system-specific configurations largely unexplored.

To address this gap, we investigate the use of LLMs to automatically generate Diktyo AppGroup Custom Resource Definitions (CRDs) for K8s deployments [22]. Diktyo enables group-based application scheduling and network-aware orchestration, but writing its AppGroup configuration is an error-prone task and requires detailed knowledge of the custom resource manifest[3]. This paper shows how LLMs can bridge the gap between user-level orchestration intents and valid AppGroup specifications, thus reducing configuration complexity, lowering the barrier for adoption, and improving deployment efficiency in cloud-native environments.

III. System Overview

This section first describes how Diktyo operates as part of the K8s platform, and then introduces our envisioned LLM-driven generation pipeline for Diktyo AppGroup CRs.

A. Diktyo within the Kubernetes platform

Diktyo is a network-aware scheduling framework designed to enhance application deployment and management within K8s clusters [22]. By introducing the concept of *AppGroups*, Diktyo enables the definition of interdependent workloads that share logical and communication relationships. Each AppGroup is described through a CRD that specifies which workloads belong to the application, their network requirements, and their dependencies on other workloads. Diktyo extends the K8s control plane through a custom controller, referred to as the AppGroup Controller, which continuously monitors AppGroup CRDs and dynamically accounts for both network and system-level constraints when deploying these

[2]https://github.com/kubernetes-sigs/scheduler-plugins

[3]https://github.com/diktyo-io/appgroup-api/blob/main/manifests/crd.yaml

groups of pods within the cluster. This tight integration enables network-aware workload placement in K8s, while maintaining compatibility with existing components (e.g., other scheduling plugins). Diktyo effectively provides the scheduler with contextual knowledge of the application graph and its underlying network topology, allowing for more efficient and network-aware orchestration decisions.

B. Towards LLM-Driven generation for Diktyo AppGroups

As mentioned previously, manually producing the CRD instances required by Diktyo to operate properly can be a complex and error-prone task. As illustrated in Listing 1, a typical AppGroup CRD not only includes high-level configuration details, such as the topology sorting algorithm, but also a wide range of fine-grained parameters. Each workload specification must define attributes such as the *selector*, *apiVersion*, and *namespace*, along with resource requirements, service dependencies, and inter-workload communication policies.

To alleviate the burden for cloud operators, we propose a LLM-driven generation framework capable of automatically producing valid AppGroup CRs based on high-level user intents or system objectives. Figure 1 illustrates the envisioned architecture of our system, where the LLM agent interfaces directly with the K8s environment through a controller layer. Upon receiving natural language or structured prompts describing the desired application topology or performance goals, the LLM agent generates corresponding CRs that encode the required specifications for Diktyo. These CRs are then validated, instantiated, and monitored in real time through the K8s API server. The framework can be further extended to support feedback-driven optimization, where deployment metrics (e.g., latency, bandwidth) are fed back into the LLM agent to iteratively refine subsequent CRs generations. This integration establishes a novel closed-loop orchestration paradigm, where LLM-based reasoning augments the declarative nature of K8s and the network-awareness of Diktyo, paving the way toward autonomous and adaptive cluster management.

IV. Methodology

This work aims to evaluate whether LLMs can translate high-level orchestration intents into valid K8s AppGroup manifests, specifically Diktyo AppGroup CRs, as writing such manifests manually is error-prone and requires deep familiarity with the CRD. To test whether LLMs can solve this task autonomously, we propose a methodology that incorporates different open-source models available on the market in the Diktyo architecture. By considering two different types of request and defining a pipeline to evaluate each LLM after a few-shot prompt, this section provides an overview of the methodology adopted, including a retry mechanism to guide models in case of erroneous generated output.

A. Requests

Thirty different AppGroup requests were created manually with varying numbers of workloads, topology sorting algorithms, and dependencies. Each request was categorized into a class based on the number of rows in the manifest of the ground truth (i.e., the response). Specifically, examples with fewer than 50 rows were labeled as *small*, examples between 50 and 180 rows long as *medium*, and the remaining ones as *large*. This resulted in a balanced dataset with 10 examples per class.

To examine LLMs' ability to interpret requests, we expressed them in two different formats. The natural language

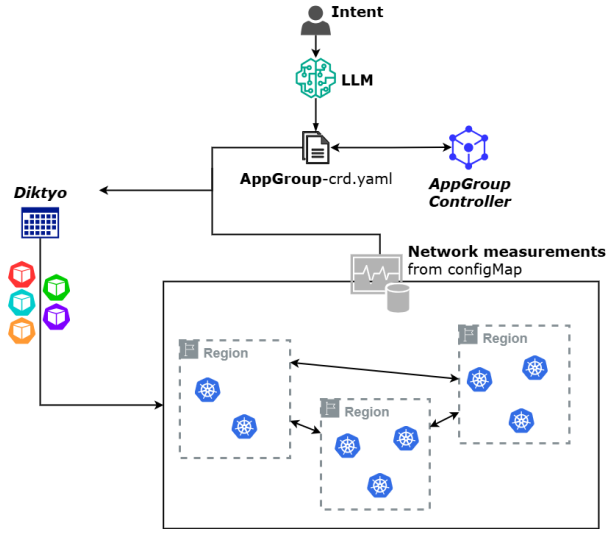


Fig. 1. The proposed LLM agent within the K8s ecosystem.

Listing 1 Example of a Diktyo CR instance.

```yaml
apiVersion: appgroup.diktyo.x-k8s.io/v1alpha1
kind: AppGroup
metadata:
  name: web-app
  namespace: default
spec:
  numMembers: 2
  topologySortingAlgorithm: KahnSort
  workloads:
    - workload:
        kind: Deployment
        name: backend-deployment
        selector: backend-deployment
        apiVersion: apps/v1
        namespace: default
      dependencies:
        - workload:
            kind: Deployment
            name: database-deployment
            selector: database-deployment
            apiVersion: apps/v1
            namespace: default
          minBandwidth: "500Mi"
          maxNetworkCost: 50
    - workload:
        kind: Deployment
        name: database-deployment
        selector: database-deployment
        apiVersion: apps/v1
        namespace: default
```

Listing 2 Example of a natural language request.

```
I need an AppGroup CRD named web-app in
the default namespace. Use 2 members,
with KahnSort for topology sorting.
The workloads to define are:
backend-deployment and database-deployment.
Also, backend-deployment depends on
database-deployment (min bandwidth 500 Mi,
max network cost 50)
```

Listing 3 Example of a template-based request.

```
Generate a Kubernetes AppGroup CRD:
name: web-app
namespace: default
- 2 members
- TopologySortingAlgorithm: KahnSort
- Workloads: backend-deployment,
    database-deployment
- Dependencies: backend-deployment ->
    database-deployment (500Mi, 50),
```

format adopts an open-ended, descriptive style in which the user freely specifies the desired configuration. This format mirrors real-world interaction scenarios, where requests are expressed in informal, human-readable sentences. It challenges the model to extract structured information, infer implicit relationships among components, and translate the narrative into a schema-compliant YAML manifest. Conversely, the template-based format enforces a rigid and explicit structure. Each request follows the same predefined sequence of fields—title, application group name, namespace, topology sorting algorithm, workloads, and dependencies with associated bandwidth and network cost constraints. This structured layout reduces linguistic ambiguity and primarily tests the model's ability to accurately map well-formed input to the target schema, rather than interpret free-form text. The listings 2 and 3 show complete examples of requests expressed in natural and template-based languages, respectively.

B. LLM Prompts

To identify the most effective way to structure the prompts to the LLMs we conducted multiple experiments. In the zero-shot setting, where only the request was provided, the models frequently failed to follow the expected schema, producing incomplete or non-compliant manifests. We then tested multi-shot prompts: including one example greatly improved structural adherence to the expected output, while adding a second example did not bring further benefits. The final design is a few-shot prompt (Listing 4), where the model is provided with the Diktyo AppGroup CRD manifest together with a complete example consisting of a request and its corresponding ground truth response. This context allows the model to align the input request with the expected structure of an AppGroup specification.

C. Evaluation Pipeline

With 30 requests and 30 outputs per LLM, manually checking each response would be impractical and error-prone. To address this, Figure 2 shows the automated evaluation

Listing 4 Few-shot prompt.

```
You are a YAML generation assistant for K8s.

You are provided with the main AppGroup CRD
manifest that defines the structure and the
attributes that are allowed during
the specification of an AppGroup YAML file.
The CRD is as follows:
{yaml_content}

Your task is to generate a valid AppGroup
YAML file based on user input using the CRD
provided above.
The AppGroup YAML file must be valid
according to the CRD schema.
Remember that each workload must also be
defined outside the dependencies section.
Do not add any comments, explanations, or any
other text in the YAML file!
USER REQUEST:
{example_req}

RESPONSE:
{example_resp}

USER REQUEST:
{user_request}

RESPONSE:
```

pipeline developed, which consists of six validation steps that a generated manifest must pass to be considered correct.

It is important to note that the pipeline evaluates the generated manifests against the corresponding ground truth specifications, not directly against the original requests. Since requests are often expressed in natural language, they are not machine-verifiable [30], [31]. Therefore, manually written ground truth manifests serve as the reference for correctness. The pipeline begins with a check for exact equivalence between the generated manifest and the ground truth. If they match, the manifest is immediately accepted, and the remaining steps are skipped. However, this situation is rare, as LLMs often introduces minor differences due to their stochastic nature, which do not affect the correctness of the produced output. In such cases, the manifest continues through the full validation process and can still be considered correct.

The steps of the evaluation pipeline are as follows:

1) ***Step 1: YAML file validation***. The manifest is parsed to ensure that it is a well-formed YAML, with the correct syntax and indentation.
2) ***Step 2: Custom Schema validation***. The manifest is validated against the Diktyo AppGroup CRD schema, using the library `kubeconform`[4] to verify field presence, data types, and structural compliance.
3) ***Step 3: Topology Sorting and ApiVersion validation***. The declared `TopologySortingAlgorithm` and `ApiVersion` are verified against the ground truth.
4) ***Step 4: Workloads validation***. The number of workloads defined in the manifest must match the value of

[4]https://github.com/yannh/kubeconform

`numMembers`, which must be the same as the ground truth.

5) ***Step 5: Dependencies validation***. The set of declared dependencies is checked against the ground truth: all referenced workloads must exist, and the resulting dependency graph must be equivalent.
6) ***Step 6: Requirements validation***. The properties of each dependency edge, which are `minBandwidth` and `maxNetworkCost`, are validated against the ground truth.

If the generated manifest passes all steps, it is considered correct. While no automatic pipeline can be entirely flawless, our approach has proven highly reliable, as we did not encounter any cases where an incorrect manifest was erroneously accepted during experimental evaluations.

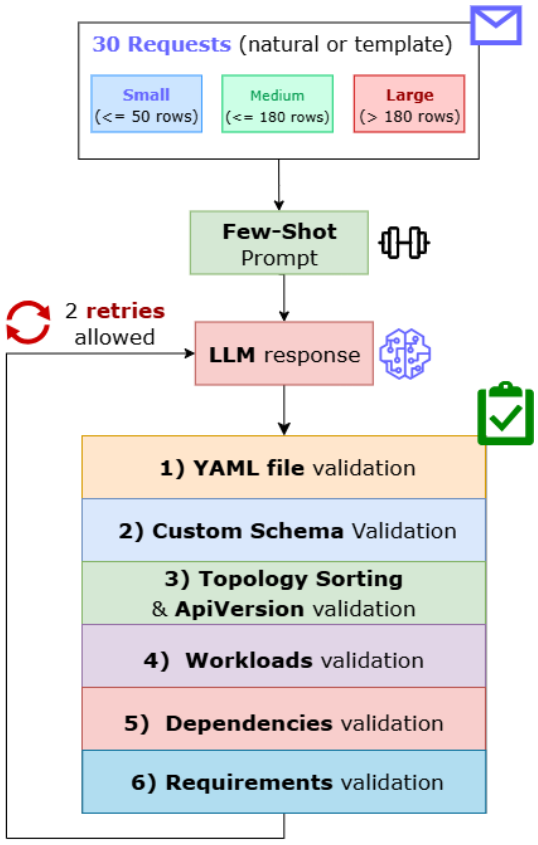


Fig. 2. Overview of the proposed methodology.

D. Retry Mechanism

To improve the quality of LLMs responses and the robustness of the approach, we implemented a retry mechanism, also shown in Figure 2. Whenever a response is certified invalid by the evaluation pipeline, the LLM is prompted again with additional guidance. In particular, the updated prompt first includes a few-shot example showing an original request, the incorrect response of the model, the detected error, and the corrected manifest. After this reminder, the prompt provides the current user request together with the previous invalid response of the model, annotated with the errors that must be fixed. This iterative strategy is applied up to two times, allowing a maximum of three generations per example.

V. Evaluation Setup

All experiments were conducted on an NVIDIA L40S GPU (48 GB VRAM) hosted in an High Performance Computing (HPC) cluster. The evaluation dataset consists of 30 requests evenly divided into three complexity classes (small, medium, large), as described in Section IV. We evaluated 7 recent open-source LLMs specialized for code generation, covering a broad spectrum of parameter sizes. The selected models are the following:

- **`Qwen2.5-Coder` (0.5B):** the smallest member of the `Qwen2.5-Coder` family, designed for lightweight local inference[5].
- **`Granite4-micro` (3B):** the smallest model in the Granite 4 family, featuring 3B dense parameters[6].
- **`Qwen2.5-Coder` (7B):** The medium-sized `Qwen2.5-Coder` checkpoint offering stronger coding performance.
- **`Qwen2.5-Coder` (14B):** the largest variant of `Qwen2.5-Coder` used in this work, targeting improved reasoning and longer input handling.
- **`DeepSeek-Coder-v2` (16B):** a Mixture-of-Experts (MoE) version of `DeepSeek-Coder`, with 2.4B activated parameters[7].
- **`Granite4-small-h` (32B):** the largest model in the Granite 4 family, with 32B parameters, of which 9B are activated[8].
- **`Gpt-oss` (20B):** OpenAI's open-weight 20B model, targeting general assistant capabilities with 3.6B activated parameters[9].

This mix of lightweight models with only a few hundred million parameters and large-scale models with up to 20 billion parameters allows us to assess how model size impacts the ability to generate valid AppGroup manifests, while also ensuring that the comparison reflects the performance of state-of-the-art coding-oriented LLMs. All models were executed using the *Ollama* framework[10] with quantized weights. The generation temperature was fixed at 0, and a constant random seed was used, ensuring deterministic and reproducible outputs. Each model was tested with both request styles: natural language requests and template-based requests. During the evaluation, models were allowed up to two retries if their initial output failed the evaluation pipeline. Multiple metrics were collected for each generation: request type, request class (*small*, *medium*, *large*), correctness of the final manifest, step reached in the evaluation pipeline, number of executions performed, the step at which the previous attempt failed, and the generation time (in seconds). This setup ensures fair

[5]https://huggingface.co/Qwen
[6]https://www.ibm.com/granite/docs/models/granite
[7]https://huggingface.co/deepseek-ai/DeepSeek-Coder-V2-Lite-Instruct
[8]https://www.ibm.com/granite/docs/models/granite
[9]https://openai.com/it-IT/index/gpt-oss-model-card/
[10]https://ollama.com

TABLE I
THE ACCURACY FOR NATURAL LANGUAGE REQUESTS.

Model	Accuracy (%)	Accuracy with retrial (%)
qwen2.5-coder:0.5b	0.0	6.7
granite4:micro	10.0	36.7
qwen2.5-coder:7b	83.3	83.3
qwen2.5-coder:14b	90.0	93.3
deepseek-coder-v2:16b	46.7	50.0
granite4:small-h	70.0	90.0
gpt-oss:20b	**93.3**	**100.0**

TABLE II
THE ACCURACY FOR TEMPLATE-BASED REQUESTS.

Model	Accuracy (%)	Accuracy with retrial (%)
qwen2.5-coder:0.5b	0.0	6.7
granite4:micro	13.3	40.0
qwen2.5-coder:7b	56.7	66.7
qwen2.5-coder:14b	86.7	90.0
deepseek-coder-v2:16b	56.7	63.3
granite4:small-h	76.7	93.3
gpt-oss:20b	**100.0**	**100.0**

comparison across models and supports the reproducibility of the results.

VI. EVALUATION RESULTS

This section presents the experimental evaluation of the selected models in the task of translating high-level intents into schema-compliant K8s `AppGroup` manifests. We analyze model accuracy, failure patterns, and execution times in both natural language and template-based request formats, highlighting the impact of model scale and retrial mechanisms. Unless otherwise stated, *accuracy with retrial* aggregates successes obtained either on the first attempt (0), or after one (+1) or two (+2) additional corrections; each model is allowed up to two retrials per request.

A. Accuracy

The accuracy results of the experiments with natural language requests are reported in Table I, while the results of the template-based requests are shown in Table II. In both request formats, the best-performing model is `gpt-oss`, which consistently achieves an accuracy of 100% after retrial in both cases. This outcome is consistent with expectations, as `gpt-oss` is also the largest model in our evaluation, with 20 billion parameters. The second-best performer is `qwen2.5-coder` (14B), which achieves an accuracy of 90.0% in the first attempt and improves to 93.3% after retrial, confirming that larger models benefit from stronger generalization and reasoning capabilities for structured YAML generation. Notably, `granite4:small-h` also performs competitively, reaching 90.0% accuracy with natural language and 93.3% with the template-based requests after retrial, despite having fewer active parameters than their fully dense counterparts.

At the opposite end of the spectrum, the smallest model, `qwen2.5-coder` (0.5B), performs poorly in both scenarios, with 0.0% accuracy and only 6.7% after retrial. This highlights the limitations of lightweight models when applied to complex intent translation tasks requiring schema compliance. A similar performance is observed for `granite4:micro`, further confirming this hypothesis. All evaluated models seem to benefit from the retrial mechanism. In our reporting, the "with retrial" columns aggregate successes obtained after either one or two additional attempts; we omit the per-attempt breakdown for clarity. In natural language requests, `granite4:micro` improves from 10.0% to 36.7% accuracy, and `granite4:small-h` improves from 70.0% to 90.0%.

When comparing input formats, the results remain similar: template-based requests provide a slight advantage for certain models (e.g., `deepseek-coder-v2`, `granite4:small-h` and `granite4-micro`), while natural language requests favor others (e.g., `qwen2.5-coder` 7B and 14B). We analyzed the accuracy by request class to examine how manifest size affects model performance, as shown in Figure 3. Smaller manifests consistently emerge as the easiest to generate across both natural language and template-based prompts, with most models achieving near-perfect accuracy after retrial. Even models with limited capacity, such as `qwen2.5-coder:0.5b` and `granite4:micro`, obtain 20 and 80% accuracy, respectively, for small manifests, while all models above 7B parameters converge to 100% in both prompting strategies.

Medium-sized manifests introduce greater structural and contextual complexity, exposing clear differences between parameter scales and prompting methods. Models below 10B parameters show a remarkable degradation (e.g., `qwen2.5-coder:0.5b` and `granite4:micro` drop below 60%), while larger models, particularly `qwen2.5-coder:7b`, `qwen2.5-coder:14b`, and `granite4:small-h`, maintain high reliability, exceeding 70–100% most of the time in both natural language and template-based requests. Large manifests remain the most challenging across all evaluated systems, amplifying the effects of context length, schema depth, and cross-field dependencies. Performance declines significantly for smaller models, with accuracies falling below 50% in both prompt types, while only `gpt-oss:20b` preserves perfect accuracy across all request classes. Notably, `qwen2.5-coder:14b` and `granite4:small-h` preserve accuracy above 80% for the natural language, suggesting that medium-to-large parameter models can effectively generalize even under extended context and compositional demands.

Overall, allowing up to two retrials increases accuracy by as much as 26%, with the actual gain varying by model and by whether requests are natural language or template-based. In short, retrials help broadly, but the size of the boost depends on the model and input format.

B. Failure Patterns

Given that natural language queries better approximate real-world operator interactions, the remainder of the discussion focuses primarily on the natural language results. Figure 4

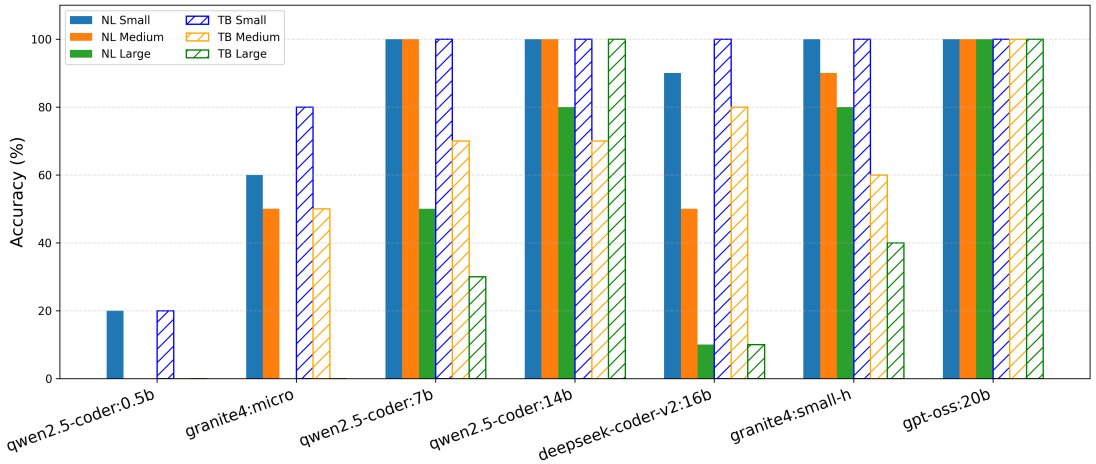


Fig. 3. Accuracy by request class after retrials with natural language and template-based requests.

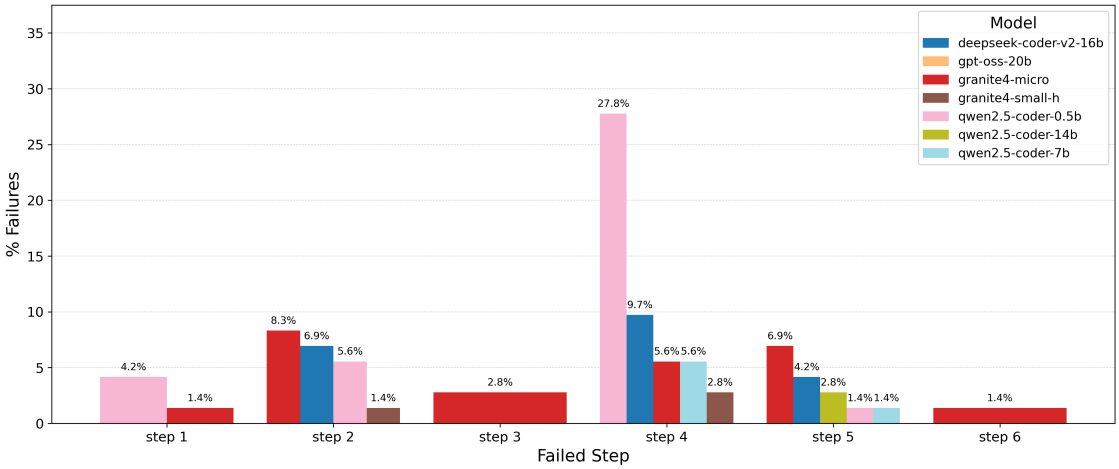


Fig. 4. Failure distribution across the validation steps for all considered models.

summarizes the distribution of model failures in the six validation stages of the evaluation pipeline. More specifically, we aggregated all the errors produced by every model and computed the proportion of failures that occurred at each step relative to the total number of observed failures. This approach not only highlights which stages are most error-prone overall, but also reveals the diversity of failure sources across models of different sizes and architectures.

The results show that the highest concentration of errors occurs in *Step 4* (workload validation), which results in 51.5% of all failures. This step represents the most demanding phase of the pipeline from a semantic perspective, where models frequently miscount workloads. The second most frequent source of errors is *Step 2* (schema validation), responsible for 22.2% of total failures. These errors typically result from missing required fields, incorrect nesting, or type mismatches within the YAML manifests, showing that strict structural compliance remains difficult for several models. Dependency validation errors in *Step 5* also represent a notable percentage (16.7%), reflecting the difficulty of generating consistent cross-workload references and maintaining coherent dependency graphs. In contrast, early-stage issues such as *Step 1* (YAML syntax validation) are comparatively rare, representing only 5.6% of all failures and occur primarily in smaller models. *Step 3* (topology sorting and `apiVersion` validation) contributes only to 2.8% of the failures, indicating that most models can reproduce these simple configuration parameters correctly. Finally, *Step 6* (requirements validation) accounts for just 1.4% of total failures, suggesting that once dependency graphs are valid, models generally maintain consistent requirement properties such as `minBandwidth` and `maxNetworkCost`.

In general, these findings emphasize that the main weaknesses of the evaluated models lie in the deeper semantic and contextual reasoning phases, particularly in workload and dependency validation, rather than in surface-level syntax or metadata requirements. A comparable distribution is observed for template-based requests, with the same dominant failure modes and relative frequencies. This indicates that the main source of difficulty lies in the intrinsic complexity of workload specification and dependency reasoning, rather than in the phrasing or structure of the input query. Because qualitative trends are indistinguishable across formats, we focus the discussion on the natural language setting to avoid redundancy; the corresponding template-based results lead to the same

TABLE III
AVERAGE EXECUTION TIME PER MODEL (INCLUDING RETRIALS) AND AVERAGE ACCURACY (WITH RETRIAL).

Model	Natural (s)	Template (s)	Avg. accuracy (%)
qwen2.5-coder:0.5b	**2.50**	**2.89**	6.7
granite4:micro	4.77	4.21	38.4
qwen2.5-coder:7b	9.01	9.00	75.0
qwen2.5-coder:14b	14.80	15.58	91.7
deepseek-coder-v2:16b	7.02	6.73	56.7
granite4:small-h	12.43	12.59	91.7
gpt-oss:20b	15.93	16.40	**100.0**

TABLE IV
RESOURCE CONSUMPTION PER MODEL.

Model	CPU (%)	RAM (GB)	GPU Util. (%)	GPU Mem. (MiB)
qwen2.5-coder:0.5b	2.10	1.04	48.79	1464.60
granite4:micro	2.11	1.02	78.84	3419.05
qwen2.5-coder:7b	2.12	1.22	81.32	5373.73
qwen2.5-coder:14b	2.12	1.40	87.98	10055.33
granite4:small-h	2.12	1.23	83.63	19704.84
deepseek-coder-v2:16b	2.12	1.19	78.63	10343.78
gpt-oss:20b	3.00	2.32	85.34	13114.76

conclusions and are omitted for brevity.

C. Execution Time and Resource Consumption

Table III reports the *average* end-to-end generation time per request for each model with natural language and template-based requests (including retrials). The table also shows the average accuracy after retrials, computed across both request formats. As expected, latency generally increases with model size and complexity, reflecting the computational cost of larger parameter counts. The smallest model, `qwen2.5-coder:0.5b`, is the fastest, completing a request in 2.50 s for natural language and 2.89 s for template inputs, but at the expense of accuracy. Among mid-size models, `granite4:micro` achieves moderate latency (4–5 s per request) due to its efficient hybrid design, while `deepseek-coder-v2:16b` maintains a strong balance between response speed (around 7 s) and accuracy, benefiting from its sparse MoE activation. The `qwen2.5-coder` series shows a clear scaling trend, with the 7B and 14B variants requiring 9.01 s and 14.80 s, respectively, for natural language generation. Larger models such as `granite4:small-h` and `gpt-oss:20b` show the longest execution times, reaching 12.4 s and 15.9 s, respectively, for natural requests. A similar pattern is observed for template-based requests, with times ranging from 2.89 s to 16.40 s. Overall, end-to-end latency (including retrials) spans 2.50–16.40 s across models and request formats, and the fastest model is `qwen2.5-coder:0.5b` at 2.50 s and 2.89 s. In comparison, the slowest is `gpt-oss:20b` at 15.93 s and 16.40 s, while average accuracy ranges from 6.7% to 100.0%. Overall, these results highlight the expected trade-off between inference latency and reasoning capability: smaller models offer faster responses but limited performance on complex manifest generation tasks, whereas larger models provide superior accuracy at the cost of longer generation times.

Lastly, Table IV compares the resource utilization profiles of the evaluated LLMs during the experiments. All models exhibit low CPU usage (around 2–3%), indicating that GPU resources dominate. Memory consumption scales with model size, with GPU utilization exceeding 80% for models beyond 7 billion parameters. Notably, GPU memory usage grows almost linearly with model scale, ranging from approximately 1.4 GB for the smallest (0.5B) model to nearly 20 GB for the largest (32B) model. These results highlight the clear trade-off between model capacity and computational footprint, emphasizing the need for resource-aware model deployment strategies in constrained environments.

VII. CONCLUSION AND FUTURE WORK

This paper explored the use of LLMs as enablers of intent-driven orchestration in cloud-native environments. Focusing on the generation of Diktyo AppGroup CRs in K8s, we demonstrated that LLMs can effectively bridge the gap between high-level orchestration intents and system-specific configuration schemas. The proposed LLM-driven pipeline translates natural language or structured intents into syntactically correct and semantically meaningful AppGroup specifications, simplifying the configuration process and enhancing deployment efficiency of the Diktyo scheduler. The evaluation showed that models such as `qwen2.5-coder:14b` and `gpt-oss:20b` achieved between 93–100% accuracy in generating valid CRs, highlighting the feasibility of adopting LLMs as assistants for orchestration intents. These results demonstrate the potential of LLMs to reduce human error, lower cognitive overhead for developers, and accelerate the deployment of workloads in K8s with the Diktyo scheduler. Despite these promising results, current models can still produce hallucinations or schema-inconsistent CRs, especially when presented with ambiguous or incomplete intents. Therefore, future work will explore Reinforcement Learning (RL) with schema-based feedback, prompt optimization, human-in-the-loop, and LLM-as-a-judge validation mechanisms, creating fully automated pipelines capable of properly working without the need for an explicit ground truth. Moreover, we plan to extend the developed LLM agent beyond Diktyo to support other cloud-native use cases, representing a natural evolution toward fully autonomous, intent-driven management of complex edge-cloud ecosystems. Lastly, we would like to emphasize that this work demonstrates that LLMs can act as a key enabler of intent-to-orchestration translation in K8s, marking a significant step toward realizing the vision of self-managing and intelligent cloud-native infrastructures for 6G and beyond.

ACKNOWLEDGMENT

This work has been partially supported by the Spoke 1 "FutureHPC & BigData" of the Italian Research Center on High-Performance Computing, Big Data and Quantum Computing (ICSC) funded by MUR Missione 4 - Next Generation EU (NGEU). José Santos is funded by the Research Foundation Flanders (FWO), grant number 1253226N.

REFERENCES

[1] G. Karsai *et al.*, "Distributed and managed: Research challenges and opportunities of the next generation cyber-physical systems," in *2014 IEEE 17th International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, 2014, pp. 1–8.

[2] L. F. Bittencourt *et al.*, "The computing continuum: Past, present, and future," *Computer Science Review*, vol. 58, p. 100782, 2025.

[3] J. Santos *et al.*, "Hephaestusforge: Optimal microservice deployment across the compute continuum via reinforcement learning," *Future Generation Computer Systems*, vol. 166, p. 107680, 2025.

[4] F. Poltronieri *et al.*, "Reinforcement learning vs. computational intelligence: Comparing service management approaches for the cloud continuum," *Future Internet*, vol. 15, no. 11, 2023.

[5] A. Tellache *et al.*, "Multi-agent reinforcement learning-based network intrusion detection system," in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*. IEEE, 2024, pp. 1–9.

[6] J. Santos *et al.*, "Efficient microservice deployment in kubernetes multi-clusters through reinforcement learning," in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*. IEEE, 2024, pp. 1–9.

[7] R. Sahraoui *et al.*, "Energy-aware vnf-fg placement with transformer-based deep reinforcement learning," in *2024 IEEE/IFIP Network Operations and Management Symposium (NOMS 2024)*, 2024.

[8] B. Bazaluk *et al.*, "Towards a transformer-based pre-trained model for iot traffic classification," in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*. IEEE, 2024, pp. 1–7.

[9] Y. Shen *et al.*, "Reinforcement learning in natural language processing: A survey," in *Proceedings of the 2023 6th International Conference on Machine Learning and Natural Language Processing*, ser. MLNLP '23. New York, NY, USA: Association for Computing Machinery, 2024, p. 84–90.

[10] C. Dobbins *et al.*, "Signal processing of multimodal mobile lifelogging data towards detecting stress in real-world driving," *IEEE Transactions on Mobile Computing*, vol. 18, no. 3, pp. 632–644, 2019.

[11] S. Ikram *et al.*, "A transformer-based multimodal object detection system for real-world applications," *IEEE Access*, vol. 13, pp. 29 162–29 176, 2025.

[12] M. A. Habib *et al.*, "Harnessing the power of llms, informers and decision transformers for intent-driven ran management in 6g," *IEEE Transactions on Network Science and Engineering*, pp. 1–20, 2025.

[13] G. Dimitrakopoulos *et al.*, "Industry 5.0: Research areas and challenges with artificial intelligence and human acceptance," *IEEE Industrial Electronics Magazine*, vol. 18, no. 4, pp. 43–54, 2024.

[14] Z. Zhao *et al.*, "Recommender systems in the era of large language models (llms)," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 11, pp. 6889–6907, 2024.

[15] L. Chen *et al.*, "Driving with llms: Fusing object-level vector modality for explainable autonomous driving," in *2024 IEEE International Conference on Robotics and Automation (ICRA)*, 2024, pp. 14 093–14 100.

[16] R. Patil *et al.*, "A review of current trends, techniques, and challenges in large language models (llms)," *Applied Sciences*, vol. 14, no. 5, 2024.

[17] A. Celik *et al.*, "At the dawn of generative ai era: A tutorial-cum-survey on new frontiers in 6g wireless intelligence," *IEEE Open Journal of the Communications Society*, vol. 5, pp. 2433–2489, 2024.

[18] J. Wang *et al.*, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, 2024.

[19] T. Coignion *et al.*, "A performance study of llm-generated code on leetcode," in *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 79–89.

[20] N. Beuter *et al.*, "Prompt-driven and kubernetes error report-aware container orchestration," *Future Internet*, vol. 17, no. 9, 2025.

[21] A. Sacco *et al.*, "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.

[22] J. Santos *et al.*, "Diktyo: Network-aware scheduling in container-based clouds," *IEEE Transactions on Network and Service Management*, vol. 20, no. 4, pp. 4461–4477, 2023.

[23] "Hugging face: The ai community building the future." https://huggingface.co/, [Online; accessed on 15-October-2025].

[24] M. Fontana *et al.*, "Leveraging llm-powered intelligent chatbots for intent-based networking in 5g modem reconfiguration," in *2025 IEEE 11th International Conference on Network Softwarization (NetSoft)*. IEEE, 2025, pp. 209–213.

[25] V. Gowtham *et al.*, "Graphgpt: An intent-based management system for next generation networks," in *2025 IEEE 11th International Conference on Network Softwarization (NetSoft)*. IEEE, 2025, pp. 25–30.

[26] A. Mekrache *et al.*, "Oss-gpt: An llm-powered intent-driven operations support system for 6g networks," in *2025 IEEE 11th International Conference on Network Softwarization (NetSoft)*. IEEE, 2025, pp. 155–163.

[27] S. Nam *et al.*, "Llm-based ai agent for vnf deployment in openstack environment," in *NOMS 2025-2025 IEEE Network Operations and Management Symposium*. IEEE, 2025, pp. 1–7.

[28] K. Rao *et al.*, "Eco-llm: Llm-based edge cloud optimization," in *Proceedings of the 2024 Workshop on AI For Systems*, ser. AI4Sys '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 7–12.

[29] B. Kim *et al.*, "Kubeteus: An intelligent network policy generation framework for containers," in *IEEE INFOCOM 2025 - IEEE Conference on Computer Communications*, 2025, pp. 1–10.

[30] A. Fan *et al.*, "Large language models for software engineering: Survey and open problems," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, 2023, pp. 31–53.

[31] F. Jiang *et al.*, "Large language model enhanced multi-agent systems for 6g communications," *IEEE Wireless Communications*, vol. 31, no. 6, pp. 48–55, 2024.

APPENDIX

A. Artifact details

This artifact is available for review at https://github.com/dream-19/Diktyo_LLM.