From Source Code to Kubernetes Manifests: Guiding LLMs with Structured Metadata

Matteo Franzil*†, Ulises Emiliano Sosa*, and Domenico Siracusa*
*Department of Information Engineering and Computer Science, University of Trento, Trento, Italy
†Center for Cybersecurity, Fondazione Bruno Kessler, Trento, Italy
Email: {matteo.franzil, domenico.siracusa}@unitn.it, ulises.e.sosa@proton.me

***Abstract*—Deploying applications on Kubernetes (K8s) often requires writing and maintaining complex manifests that accurately reflect the application's architecture and requirements. This task requires significant expertise, time, and effort, especially for heterogeneous architectures composed of multiple microservices, interdependencies, and configuration options. In recent years, Large Language Models (LLMs) have been proposed to automate cloud application lifecycle phases, including manifest generation. However, obtaining accurate manifests requires specific domain knowledge about both the application and K8s concepts, which can be difficult to convey to LLMs effectively. Attempting to address this by feeding the entire codebase quickly exceeds the model's context window, while using simple natural language instructions often leads to incorrect or incomplete manifests. Filling this gap, we present a novel pipeline that leverages structured metadata to guide LLMs in generating accurate and deployable K8s manifests. Our approach directly scans application repositories, deterministically extracting metadata about each microservice and its dependencies. This information is structured into an Intermediate Representation (IR), a concise summary of the application's architecture and requirements, which is then used to prompt LLMs for manifest generation. We evaluate our pipeline on a dataset of 40 microservice architectures, demonstrating significant improvements in deployment success rates: from 15% with direct prompting to 47.5% with IR guidance, and up to 62.5% when incorporating minor manual overrides. While the results highlight the effectiveness of structured metadata in enhancing LLMs' performance, they also underscore the need for human guidance when dealing with complex and heterogeneous scenarios.**

***Index Terms*—Large Language Models, Kubernetes, Infrastructure as Code, Configuration Management, Network Automation, Generative AI**

I. Introduction

K8s is the de-facto standard for orchestrating containerized applications at scale [1]. Users employ a declarative Application Programming Interface (API) to manage deployment, scaling, and lifecycle operations while abstracting the underlying infrastructure [2]. Resources are created and managed through *manifests*, i.e., YAML files that define the desired objects and their configurations. While this approach is powerful and flexible, it also introduces significant challenges in manifest creation and maintenance.

Efficiently working with manifests requires deep knowledge of K8s concepts and the application itself, especially when creating them from scratch. As microservices increase in number and size, manifests can become convoluted, lengthy, and difficult to maintain. Considering (i) the numerous configuration options available for each K8s resource, (ii) the intricate interdependencies among multiple services and their requirements, and (iii) YAML's sensitivity to indentation and formatting, even a small mistake can lead to deployment failures and hours of debugging. Tools like Helm [3] and Kustomize [4] help manage complexity through templating and abstraction, yet significant manual effort is still required to ensure correctness and consistency with application and environment needs [5].

To automate this process and reduce the manual burden, cloud and infrastructure engineers have recently turned to LLMs to assist in manifest generation [6]. Their unmatched ability to generate context-aware code in seconds has enabled new forms of automation in software development workflows. Generating K8s manifests, however, requires combining knowledge of the application architecture with K8s concepts and practical deployment choices (e.g., resource limits, types of services), a task that remains challenging for LLMs when not provided with sufficient context.

One could naively attempt to address this by directly prompting an LLM with the entire codebase of the target application. However, this approach faces two major challenges. First, even state-of-the-art LLMs have limited context windows, making it infeasible to feed them thousands of lines of code. Second, real-world repositories often contain heterogeneous artifacts, outdated or partial documentation, and inconsistent requirements. This often leads to manifests that appear correct but fail to deploy or behave improperly at runtime. On the other end of the spectrum, prompting the model with only a vague natural language description of the application may work for popular applications, but quickly falls short in scenarios where the model lacks pre-existing knowledge or when specific configuration choices are required.

In other generative tasks, techniques such as chain-of-thought prompting [7], iterative feedback loops [8], retrieval-augmented generation [9], and post-generation automated validation [10] have shown success in improving the quality of generated outputs. In the specific context of K8s manifest generation, prior work has focused on n-shot and prompt engineering [7] or templating [11]. However, neither approach starts from the actual source code of the target application, effectively limiting their applicability in real-world scenarios.

Given the speed at which LLM capabilities are improving, one might expect that simply using larger models with bigger

context windows would eventually solve these issues. However, relying on larger models is not a sustainable solution due to costs, diminishing returns, and environmental impact [12]. We argue that blindly scaling up model size is not a viable path forward, and that carefully curating and structuring the information provided to the model can improve performance without requiring significant resources or engineering effort.

Building on this observation, we propose a novel pipeline designed to automatically gather, structure, and enrich metadata from Docker-based microservice repositories before presenting it to an LLM. Prior to any interaction, we perform static analysis of the repository to extract relevant metadata about each microservice, such as its dependencies, environment variables, exposed ports, and resource requirements. This information is then structured into an Intermediate Representation (IR), a concise, hierarchical and structured summary of the application's architecture and requirements. The IR is then used to prompt the LLM for manifest generation.

We developed a proof-of-concept implementation using the Anthropic Claude 3.5 model, evaluating it on a dataset of 40 microservice stacks collected from public repositories of various sizes and complexities. The pipeline automatically scans the code, constructs the IR, generates the manifests, and evaluates their deployability and correctness. Our findings both shed light on and cast a shadow over the capabilities of LLMs in generating complex infrastructure manifests. On one hand, they demonstrate the effectiveness of providing structured metadata over both large amounts of code and natural language descriptions. On the other hand, they underscore the limitations of current LLMs in fully automating such tasks. In complex and heterogeneous scenarios, even a small mistake or a misconfigured option can lead to deployment failures. Thus, at the current state of the art, human oversight remains irreplaceable in critical deployment scenarios.

The rest of the paper is organized as follows. Section II describes the process of extracting metadata from repositories and constructing the IR. Section III discusses how the data is then used to prompt LLMs for manifest generation and how it is evaluated. Section IV shows how we validated the approach on a dataset of 40 microservice architectures and discusses the results. Section V briefly mentions related work in LLM-based code generation and K8s configuration automation. Finally, Section VI concludes the paper.

II. Extracting Metadata from Repositories

The process of generating K8s manifests using LLMs can be broadly divided into two main stages: (i) metadata extraction and IR construction, and (ii) manifest generation using LLMs. These stages are detailed in the pseudocode of Figure 1, while a graphical example is provided in Figure 2.

A. Docker and Compose File Analysis

First, the target repository is scanned to identify microservice directories. For each microservice, Dockerfiles and Docker Compose files are parsed to extract relevant configuration information; these files are the primary sources of information for the extraction process. If a Dockerfile is found, a dedicated extraction module parses it to identify runtime-relevant directives. Rather than including every directive, which would be noisy and overwhelming for complex images, we restrict extraction to those that have a direct impact on container behavior within a K8s cluster.

```
1: IR ← ∅
2: for all repository ∈ repositories do
3:     for all subdirectory ∈ repository do
4:         parse Dockerfile (sec. II-A)
5:         parse Compose file (sec. II-A)
6:         extract environment file (sec. II-B)
7:         infer framework-derived metadata (sec. II-B)
8:         resolve metadata conflicts (sec. II-B)
9:     end for
10:    IR ← IR ∪ metadata
11: end for
12: manifests ← ∅
13: for all microservice ∈ IR do
14:    manifest ← LLM_query(prompt, IR) (sec. III)
15:    manifests ← manifests ∪ manifest
16: end for
17: (optional) evaluate and refine manifests (sec. III-B)
18: return manifests
```

Fig. 1: High-level pseudocode of the pipeline.

Table I summarizes the selected Dockerfile directives and their corresponding mappings to K8s resource specifications. For example, `CMD` and `ENTRYPOINT` define the primary process of the container and map to the `command` and `args` fields in the Pod specification. Similarly, `VOLUME` directives guide the generation of `volumeMounts` and Persistent Volume Claims (PVCs)[1]. Build-time instructions such as `RUN` or `COPY` are intentionally ignored, as they influence image construction but have no direct impact on K8s orchestration.

As a practical example, Figure 3 illustrates the metadata tree extracted from a .NET Worker Dockerfile. The extraction module identifies the base image, entrypoint command, and working directory, constructing a hierarchical representation that captures the essential runtime configuration of the microservice.

When Dockerfiles are accompanied by Docker Compose files, the extraction module processes them to capture architectural metadata. Compose files define how services interconnect, depend on one another, and expose functionality, making them a rich source of information about the application's topology. Table II summarizes the selected Docker Compose directives and their corresponding mappings to K8s resource specifications. For example, `depends_on` entries create a Directed Acyclic Graph (DAG) representing startup ordering and logical dependencies, which inform the synthesis of `initContainers` or distributed readiness checks.

[1]A PVC is a request for storage by a user that abstracts the underlying storage implementation.

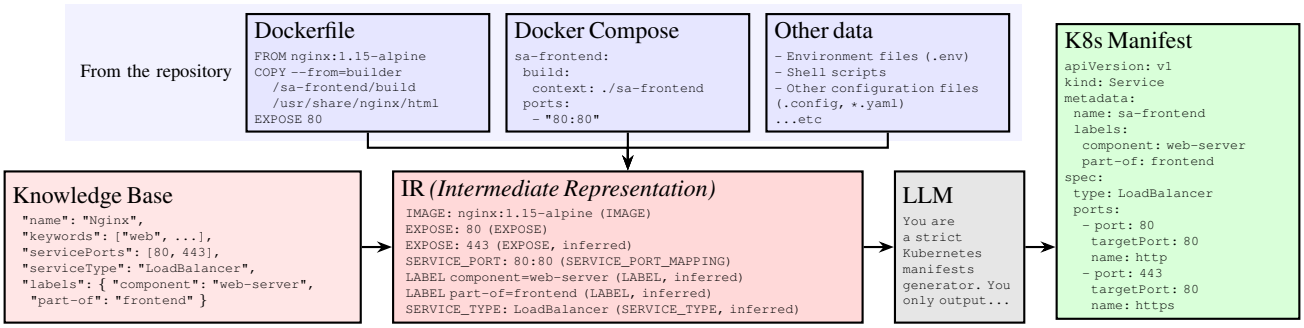

Fig. 2: An example of the pipeline applied to an Nginx-based microservice. The Dockerfile and Docker Compose files are parsed first, then the knowledge base enriches the metadata, and the resulting IR is used to prompt the LLM.

Directive	Kubernetes Mapping
CMD	`command`, `args` in Pod spec
ENTRYPOINT	(same as `CMD`)
EXPOSE	`containerPort` in Pod spec, `Service` ports
ENV	`env` in Pod spec, `ConfigMap`/`Secret`
VOLUME	`volumeMounts` and PVCs
WORKDIR	`workingDir` in Pod spec
USER	`securityContext.runAsUser` in Pod spec
HEALTHCHECK	Readiness and liveness probes

TABLE I: Dockerfile directives mapped to Kubernetes resource specifications.

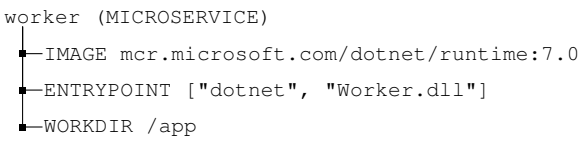

Fig. 3: Example of metadata tree extracted from a .NET Worker Dockerfile.

Figure 4 illustrates how a Compose file enriches the metadata tree of the `worker` service from the previous example. The extraction module adds `DEPENDENCY` nodes for `redis` and `db`, each with their respective health check conditions, as well as a `NETWORK` node indicating membership in the `back-tier` network.

B. Enrichment and Conflict Resolution

The extraction module also identifies environment variables declared in Dockerfiles, Compose files, or auxiliary files such as `.env`, `.config`, and shell scripts, explicitly scanning the repository for such files.

Variables explicitly declared with directives like `ENV` are directly extracted and classified. When shell scripts are referenced, they are further analyzed to identify variable assignments, export statements, and other potential sources of configuration data (e.g., sourcing other scripts, opening sockets, reading from files).

Unfortunately, some metadata may not be explicitly present in the repository. To obtain a fully functional manifest, certain information must be inferred based on standards and best practices rather than explicit declarations. For example, web servers are expected to listen on standard ports 80 and 443. When this information is not declared, the extraction module infers it based on the identified framework (e.g., Nginx, Apache, Express.js) and enriches the metadata tree accordingly. For example, Figure 2 shows how the Knowledge Base provides default values for the service ports, labels, and type service (`LoadBalancer`) for an Nginx-based microservice. The knowledge base has been curated from official documentation and best practices for various frameworks and is stored in a structured format for easy lookup and integrations if needed.

Directive	Kubernetes Mapping
services	`Deployment`, `Service` resources
depends_on	Init containers, readiness checks
networks	Service connectivity, internal/external exposure
volumes	PVC resources
ports	`NodePort`, `LoadBalancer` Services

TABLE II: Docker Compose directives mapped to Kubernetes resource specifications.

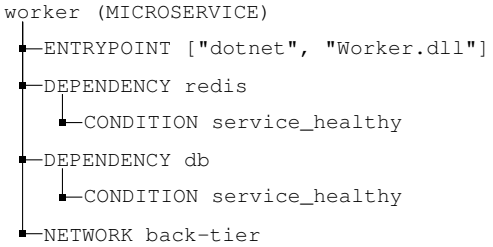

Fig. 4: Example of metadata tree enriched by Docker Compose for the `worker` service. `IMAGE` and `WORKDIR` nodes omitted for brevity.

The opposite scenario may also occur: environment variables, services, or even volumes may be defined multiple times across different files, leading to potential conflicts. We adopt a conflict resolution strategy based on a pre-trained sentence transformer model, specifically `all-MiniLM-L6-v2` [13], [14]. This model computes dense vector embeddings of conflicting metadata elements and measures their cosine similarity. If the similarity score exceeds a dynamic threshold τ, the elements are considered semantically relevant, and the entry with highest confidence (e.g., more explicit declaration or recent modification date) is retained.

III. Evaluating the Generated Manifests

Once the IR is constructed, it serves as the primary input to the LLM for manifest generation.

A. Configurations

We evaluate three different configurations of the manifest generation process to assess the impact of the IR and manual corrections on deployment success rates.

1) **Without IR (Baseline)**: In this configuration, the LLM is directly prompted using the unprocessed text gathered from the Dockerfile, Compose and contextual files (e.g. README *txt *.cfg, *ini, etc) to generate K8s manifests. No structured metadata or IR is provided. This configuration serves as the baseline for comparison.
2) **With IR**: The LLM is prompted using the structured IR generated from the repository rather than the raw codebase. This configuration tests the effectiveness of the IR in guiding the LLM to produce accurate manifests.
3) **With Overrides**: In addition to the IR, the user can supply additional parameters to override or correct specific metadata entries—potentially identified as problematic in previous runs (obtained through failed deployments logs or manual inspection). This configuration tests how additional user feedback can further improve manifest quality.

All prompts are designed using a template filled dynamically with the extracted metadata to ensure consistency across configurations, differing only in the inclusion of the IR and overrides where applicable. They include clear instructions on the desired output format and best practices for K8s manifest creation. No limitations are placed on the types of resources that can be generated, but it is explicitly instructed to only output valid YAML manifests without any additional text.

B. Validation Criteria and Metrics

Once the manifests are generated, they undergo a series of validation steps to ensure their correctness and security. We evaluate the generated manifests under three different criteria:

Deployability. We evaluate whether the generated manifests are ready to be deployed on a real K8s cluster. We run Skaffold [15], a tool that automates the deployment process and provides feedback on errors encountered during deployment. We track whether the manifests can be successfully *rendered*[2], *deployed*, and whether the services reach a healthy state.

Human Effort. We measure the amount of manual intervention required to correct the generated manifests and achieve successful deployment. Each generated manifest is manually evaluated, and the number of changes (Git-style additions, deletions, modifications) needed to fix issues is recorded. We call this metric the *Number of Corrections (NoC)*.

Semantic Alignment and Runtime Behavior. Finally, we assess whether the deployed application behaves as intended based on the documentation provided in the repository. We do so in two ways: (i) by manually testing the application's features to ensure they align with documented expectations, and (ii) by using an *LLM-as-a-judge* approach, where we prompt the same model to compare the expected behavior described in the documentation with the observed behavior after deployment, identifying any discrepancies. This dual approach provides a comprehensive view of the functional correctness of the deployed application.

[2]Rendering refers to the process of performing substitutions and template evaluations to produce final YAML files ready for deployment.

IV. Experimental Results

We evaluate our pipeline on an experimental testbed and a dataset of 40 Docker-based microservice repositories collected from public sources.

A. Testbed Setup

On a testbed running Ubuntu 24.04 LTS, we implemented the pipeline using Python 3.12 and various libraries for Dockerfile parsing, YAML handling, embedding computations, and K8s interactions. The LLM backend used was Anthropic's Claude 3.5 Sonnet, accessed via the Anthropic API with a temperature of 0 to reduce randomness in token selection. The local Kubernetes (version v1.33) cluster was provisioned using Minikube v1.36. Finally, we used Skaffold v2.16.1 for deployment automation and Kubescape v3.0.8 for security analysis as described in Section III.

All the code, scripts, and resources used for the pipeline implementation, evaluation, and result analysis are open source and are available on GitHub[3] and Figshare[4]

B. Microservice Dataset

The evaluation was performed on a curated set of 40 public microservice repositories, totaling 106 microservices across all repositories. These reflect a broad spectrum of real-world applications, spanning multiple languages (e.g., Python, Node.js), database backends (e.g., PostgreSQL, Redis), service topologies, and custom network configurations. They vary in size, with the smallest containing just two services and the largest including up to twelve services. The dataset also includes complex systems such as the Boutique Shop authored by Google [16]. All repositories meet a series of quality criteria: they must contain two or more containerized applications, be well-documented, and work correctly when cloned and executed locally. Furthermore, 62.5% of applications required persistent storage configuration, and 17.5% included explicit health checks, providing a realistic set of deployment challenges for the pipeline to address.

C. Deployability

Figure 5 summarizes the results of the deployability evaluation across the three experimental stages: *Without IR*, *With IR*, and *With Overrides*. The latter two stages are supplemented

[3]The source code of the pipeline is available at https://github.com/Ulises961/manifest-generator. The scripts used for evaluation are available at https://github.com/Ulises961/thesis-data-analysis

[4]The dataset is available at https://figshare.com/s/1ade996d74dd40da5c31. Commit hashes for the repositories: Awesome Compose - 18f59bdb; Wordsmith - 23c61bba; Istio - 575685ec; Pasture - tag 0.2.5

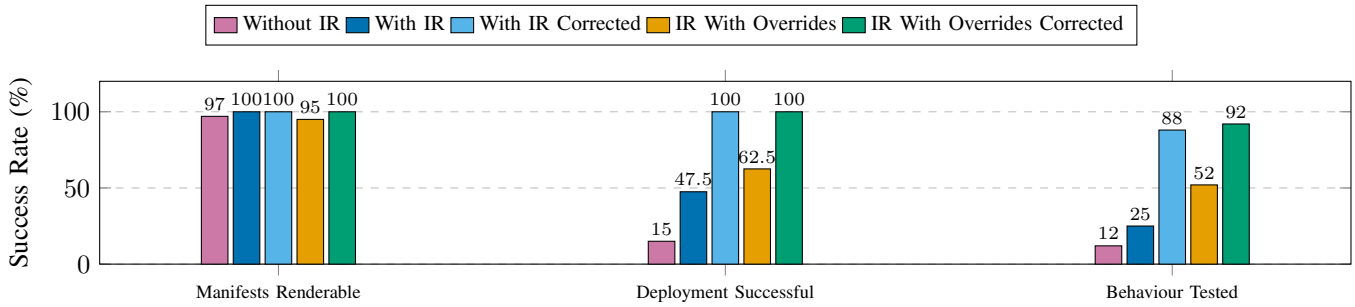

Fig. 5: Success rates for renderability, deployment, and runtime behaviour across different pipeline configurations.

by two additional configurations where manual corrections were applied to the generated manifests to achieve successful deployment, denoted as "With IR Corrected" and "With Overrides Corrected". While syntactic validity remained nearly perfect across all stages, significant differences emerged in deployment success rates and runtime behavior. Plain IR-less generation achieved only a 15% deployment success rate, while the introduction of the IR boosted this to 47.5%. Adding user-specified overrides further increased the success rate to 62.5%. Testing the actual behavior of the deployed applications revealed even starker contrasts, with only 12% of applications functioning correctly in the baseline stage, compared to 25% with the IR and 52.5% with overrides.

Manually correcting the generated manifests (the extent of which is described in Section IV-D) led to a 100% deployment success rate in both IR-based configurations. Nevertheless, as shown in Section IV-E, some applications still exhibited semantic misalignments even after corrections, indicating that certain application semantics remain challenging to infer from static analysis and LLM generation alone.

D. Human Effort

We evaluated the human effort required to correct the generated manifests in the two IR-based configurations. Figure 6 shows how many manifests required a given number of manual edits to achieve successful deployment. With only the IR (top), 29 out of 40 manifests required fewer than 20 edits, with a mean of 23.08 and a median of 3.5, given the presence of several significant outliers. With the IR and manual overrides (bottom), the number of edits dropped significantly, with a median of 0, a mean of 9.82 edits, and more than half of the manifests needing no edits post-override.

In the IR-only configuration, the majority of the edits involved the addition of missing resources such as ConfigMaps and PersistentVolumeClaims, which were not inferred correctly from the static analysis. Conversely, in the IR with overrides configuration, most edits were minor adjustments to resource specifications, such as correcting port numbers or database paths, indicating that the overrides helped in avoiding more substantial corrections.

We further analyzed the relationship between the number of edits and the size of the generated manifests (measured in lines of YAML), finding no significant correlation (Pearson correlation coefficient $r \approx -0.06$). Through manual review, we observed that the distribution of edits appears more influenced by specific application requirements and the presence of ad hoc configurations that are challenging to infer automatically. Figure 7 shows two examples of common errors found in the generated manifests: on the left, a structural error caused by duplicated `selector` fields in a `Service` manifest; on the right, a semantic error where the `Service` type should be `LoadBalancer` instead of `ClusterIP` to allow external access. Other common types of errors included missing environment variables, removal of redundant proxy services (i.e., an Nginx service, which would be redundant in K8s), and incorrect port mappings.

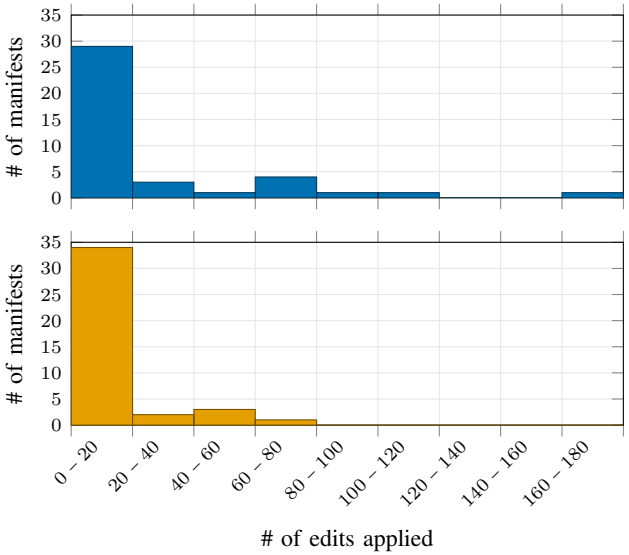

Fig. 6: Distribution of manual corrections required across applications for the "With IR" (top) and "IR with Overrides" (bottom) configurations.

E. Semantic Alignment and Runtime Behavior

While the results show that LLMs benefit from additional context, a significant number of applications still fail to behave as expected. We evaluated the runtime behavior of the deployed applications using both manual testing and an *LLM-*

```
apiVersion: v1
kind: Service
metadata:
 name: result
spec:
 type: LoadBalancer
selector:
app: result
 ports:
 - port: 9229
   targetPort: 9229
   name: debug
selector:
app: result
```

```
apiVersion: v1
kind: Service
metadata:
 name: productpage
spec:
type: ClusterIP
 selector:
   app: productpage
 ports:
 - port: 9080
   targetPort: 9080
   protocol: TCP

[...]
```

Fig. 7: Example of (left) structural and (right) semantic errors found in generated manifests.

Stage	Works Correctly (Human)	Aligned to Intent (LLM)
Without IR	12%	95%
With IR	25%	92%
With IR Corrected	88%	92%
IR + Overrides	52%	98%
IR + Overrides Corrected	92%	95%

TABLE III: Runtime behavior and manifest alignment results across different stages.

as-a-judge approach. Table III summarizes the results across both evaluation methods.

Even after manual intervention, 12% (IR-only) and 8% (IR with overrides) of applications failed to meet their documented functional requirements. Common issues included inherent incompatibility of the application with K8s paradigms (e.g., reliance on local file storage) and subtle misconfigurations (e.g., incorrect environment variable names). Since the goal was to assess manifest generation quality, we did not further investigate these failures, as they often stem from application design choices that are difficult to track.

Interestingly, the *LLM-as-a-judge* evaluated its own generated manifests more leniently than manual testers. Across all stages, the LLM reported high semantic alignment with the documentation, always above 90%. This suggests that the model perceives its outputs as consistent with the provided information even when manual testing reveals discrepancies. Indeed, many manifests deemed aligned by the LLM required substantial modifications to function correctly at runtime. We believe the main limitation lies in the model's inability to predict runtime success based on syntactic analysis alone. As a practical example, several applications that suggested proxies or sidecars were not configured with them by the LLM, leading to deployment failures as the model could not infer whether these architectural decisions applied to the K8s use case.

As a further confirmation of the limitations of the *LLM-as-a-judge* approach, we computed a confusion matrix comparing the LLM's judgements with manual evaluations and obtained a Cohen's Kappa score $\kappa = -0.103$, which indicates that random guessing would have performed better than the model's judgements.

V. Related Work

Automating network and cloud configurations has been a long-standing research area, with various approaches proposed over the years, and the rise of Generative AI and LLMs has accelerated interest in this domain [17]. Angi et al. [11] show how starting from pre-defined YAML templates and using LLMs with n-shot prompting reliably produces valid K8s manifests. However, their scalability with more complex applications and architectures remains untested. Similarly, Kratzke et al. [18] and Beuter et al. [7] propose prompt engineering techniques to generate single K8s manifests using LLMs and direct prompting. They rely on a small subset of well-known applications (e.g., Nginx, Redis), and show that even in these simple cases, the generated manifests often require manual corrections to be deployable.

LLMs have been applied also to other networking domains beyond K8s manifest generation. Cohen et al. [19] present KubeGuard, a system that leverages LLMs to harden K8s configurations and permissions at runtime, improving security posture. Dzeparoska et al. [20] proposed an intent-based networking framework that translates high-level intents into low-level network configurations using traditional rule-based systems. Other works have explored LLMs for generating network configurations [21], [22], threat modeling [23], and network scheduling [24].

VI. Conclusion

This paper presented a novel pipeline for generating deployable K8s manifests from microservice repositories using structured metadata extraction and LLMs. By extracting relevant configuration details from Dockerfiles and Compose files and synthesizing them into an intermediate representation, we increased the deployability of the resulting manifests from 15% to 47.5%. The results are promising but also highlight the challenges that remain in fully automating this process. While the use of structured metadata significantly reduces the number of manual corrections needed, certain semantic requirements and runtime behaviors remain difficult to infer from static analysis alone, and manual intervention is still needed when the application complexity exceeds what can be inferred from the codebase. Future work will explore iterative feedback loops with LLMs, testing on different LLM architectures and datasets, and integrating dynamic analysis techniques to capture more nuanced runtime behaviours.

VII. Acknowledgements

This work has been partially supported by Progetto Integrato 2.1 "Cyber Security dei sistemi energetici" PTR25-27, funded by Ministero dell'Ambiente e della Sicurezza Energetica (MASE).

References

[1] W. O. Owobu, P. Gbenle, and C. E. Alozie, "The evolution and impact of kubernetes in modern software engineering: A review," *International Journal of Academic and Applied Research (IJAAR)*, vol. 9, no. 4, pp. 56–63, Apr. 2025. [Online]. Available: https://www.ijeais.org/ijaar

[2] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016. [Online]. Available: https://doi.org/10.1145/2890784

[3] T. H. Authors, "Helm: The kubernetes package manager," https://helm.sh/docs/topics/charts, 2025, accessed: 2025-11-07.

[4] T. K. Authors, "Kustomize: Customization of kubernetes yaml configurations," https://kustomize.io/, 2025, accessed: 2025-11-07.

[5] A. Zerouali, R. Opdebeeck, and C. De Roover, "Helm charts for kubernetes applications: Evolution, outdatedness and security risks," in *Proceedings of the 2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR 2023)*, ser. Proceedings - 2023 IEEE/ACM 20th International Conference on Mining Software Repositories, MSR 2023. IEEE, May 2023, pp. 523–533. [Online]. Available: https://conf.researchr.org/home/msr-2023

[6] S. Bistarelli, M. Fiore, I. Mercanti, and M. Mongiello, "Usage of large language model for code generation tasks: A review," *SN Computer Science*, vol. 6, no. 673, 2025, rEVIEW ARTICLE.

[7] N. Beuter, A. Drews, and N. Kratzke, "Prompt-driven and kubernetes error report-aware container orchestration," *Future Internet*, vol. 17, no. 9, 2025. [Online]. Available: https://www.mdpi.com/1999-5903/17/9/416

[8] V. Tamminedi, "Automating kubernetes operations with ai and machine learning," *International Journal For Multidisciplinary Research*, 2024. [Online]. Available: https://api.semanticscholar.org/CorpusID:274951891

[9] S. D'Urso, B. Martini, and F. Sciarrone, "A Novel LLM Architecture for Intelligent System Configuration," in *2024 28th International Conference Information Visualisation (IV)*, Jul. 2024, pp. 326–331. [Online]. Available: https://ieeexplore.ieee.org/document/10714278

[10] I. Kolawole and A. Fakokunde, "Machine learning algorithms in devops: Optimizing software development and deployment workflows with precision," *International Journal of Computer Applications Technology and Research*, pp. 247–264, 01 2025.

[11] A. Angi, L. Nedoshivina, A. Sacco, S. Braghin, and M. 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)*, G. Rehm and Y. Li, Eds. Vienna, Austria: Association for Computational Linguistics, Jul. 2025, pp. 345–354. [Online]. Available: https://aclanthology.org/2025.acl-industry.27/

[12] Z. Zhou, X. Ning, K. Hong, T. Fu, J. Xu, S. Li, Y. Lou, L. Wang, Z. Yuan, X. Li, S. Yan, G. Dai, X.-P. Zhang, Y. Dong, and Y. Wang, "A Survey on Efficient Inference for Large Language Models," Jul. 2024. [Online]. Available: http://arxiv.org/abs/2404.14294

[13] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, Hong Kong, China, 2019, pp. 3982–3992. [Online]. Available: https://aclanthology.org/D19-1410/

[14] Sentence-Transformers, "sentence-transformers/all-MiniLM-L6-v2," https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2, 2020, model card, accessed 2025-11-27.

[15] G. LLC, "Skaffold: Easy and repeatable kubernetes development," https://skaffold.dev/, 2025, accessed: 2025-11-07.

[16] ——, "Google cloud microservices demo: Boutique shop," https://github.com/GoogleCloudPlatform/microservices-demo, 2025, accessed: 2025-11-07.

[17] H. Zhou, C. Hu, Y. Yuan, Y. Cui, Y. Jin, C. Chen, H. Wu, D. Yuan, L. Jiang, D. Wu, X. Liu, C. Zhang, X. Wang, and J. Liu, "Large Language Model (LLM) for Telecommunications: A Comprehensive Survey on Principles, Key Techniques, and Opportunities," Sep. 2024. [Online]. Available: http://arxiv.org/abs/2405.10825

[18] N. Kratzke and A. Drews, "Don't Train, Just Prompt: Towards a Prompt Engineering Approach for a More Generative Container Orchestration Management:," in *Proceedings of the 14th International Conference on Cloud Computing and Services Science*. Angers, France: SCITEPRESS - Science and Technology Publications, 2024, pp. 248–256. [Online]. Available: https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0012710300003711

[19] O. S. Cohen, E. Malul, Y. Meidan, D. Mimran, Y. Elovici, and A. Shabtai, "KubeGuard: LLM-Assisted Kubernetes Hardening via Configuration Files and Runtime Logs Analysis," Sep. 2025. [Online]. Available: http://arxiv.org/abs/2509.04191

[20] K. Dzeparoska, J. Lin, A. Tizghadam, and A. Leon-Garcia, "LLM-Based Policy Generation for Intent-Based Management of Applications," in *2023 19th International Conference on Network and Service Management (CNSM)*, Oct. 2023, pp. 1–7. [Online]. Available: https://ieeexplore.ieee.org/document/10327837

[21] D. M. Manias, A. Chouman, and A. Shami, "Towards Intent-Based Network Management: Large Language Models for Intent Extraction in 5G Core Networks," May 2024. [Online]. Available: http://arxiv.org/abs/2403.02238

[22] C. Wang, M. Scazzariello, A. Farshin, S. Ferlin, D. Kostić, and M. Chiesa, "NetConfEval: Can LLMs Facilitate Network Configuration?" *Proceedings of the ACM on Networking*, vol. 2, no. CoNEXT2, pp. 1–25, Jun. 2024. [Online]. Available: https://dl.acm.org/doi/10.1145/3656296

[23] S. Munshi, S. Pathak, S. Ghatode, T. Priyadarshini, D. Chandramouleeswaran, and A. Rana, "ACSE-Eval: Can LLMs threat model real-world cloud infrastructure?" May 2025. [Online]. Available: http://arxiv.org/abs/2505.11565

[24] M. Elkael, M. Polese, R. Prasad, S. Maxenti, and T. Melodia, "ALLSTaR: Automated LLM-Driven Scheduler Generation and Testing for Intent-Based RAN," May 2025. [Online]. Available: http://arxiv.org/abs/2505.18389