BRAT: An Intent-to-Kubernetes Translation via LLM Fine-Tuning

BRAT: an Intent-to-Kubernetes Translation via LLM Fine-Tuning

Antonino Angi*† Liubov Nedoshivina † Alessio Sacco * Stefano Braghin † Mark Purcell †
* *Department of Control and Computer Engineering*, Politecnico di Torino, Italy
† IBM Research, Dublin, Ireland

***Abstract*—The widespread adoption of cloud computing technologies enabled notable consolidation of computational resources, with Kubernetes (K8s) as the default open-source container orchestration platform that automates application management. Since properly configuring K8s requires writing manifest files, *i.e.,* YAML-structured files that define the containerized environment, the advent of Large Language Models (LLMs) can automate this process by relaxing user input to an intent expressed in natural language. However, fine-tuning an LLM poses unique challenges and reveals gaps in domain-specific knowledge, as the generated output must be both semantically and syntactically correct, and the area lacks a high-quality dataset. To address these challenges, we propose BRAT, a pipeline that generates K8s manifests from user-defined intents using a fine-tuned LLM. First, we created a dataset of intent-K8s pairs, leveraging advanced LLM and applying *n*-shot learning. The adoption of those models revealed that different models require personalized prompting, increasing costs by up to 66%. Then, leveraging the newly created dataset, we fine-tuned `Granite-3B`, chosen for its lightweight capabilities, to generate accurate manifest files, resulting in an average improvement of 110% over the baseline.**

***Index Terms*—Kubernetes, service orchestration, LLM**

I. Introduction

Traditional cloud computing operations often involve complex manual configurations, particularly in service deployment of containerized environments (*e.g.,* microservices). Kubernetes (K8s) is considered the default open-source container orchestration platform for automating software deployment, as well as scaling and further management. It operates by a *manifest*, a structured file that lists the main deployment features.[1] However, tasks such as defining deployment features, network policies, and services require significant expertise and can be challenging even for experienced users. In response, intent-based approaches often powered by Large Language Models (LLMs), have emerged as a promising solution [1], [2]. By enabling high-level intents expressed in natural language (*intents* from now on), this approach has the potential to simplify configuration tasks, make them more accessible, and speed up the deployment of network and application configurations.

While LLMs have shown versatility across different domains [3], [4], there are techniques, such as prompting or fine-tuning, that can make LLM domain-specific. Although, the former uses customized prompts to guide the model towards more accurate outputs and does not require training, it often requires extensive prompt adjustments and human intervention [1], leading to a slower, less scalable, and automated solution. In contrast, fine-tuning involves training the model on domain-specific data to refine its understanding and performance. This approach is particularly advantageous for tasks that require structured representation, such as generating K8s manifests, as it trains the model to produce outputs with specific syntax (*i.e.,* YAML or JSON) and semantic constraints of the domain. By reducing reliance on human intervention or extensive prompt engineering with few-shot examples, fine-tuning not only improves efficiency but also decreases input token requirements over time, leading to significant cost reductions [5], [6]. However, to fine-tune an LLM effectively, a sufficiently large database of intent-manifest pairs is required, which is not always available or easy to obtain. Without sufficient data, fine-tuning becomes challenging, as the model lacks the context needed to generate accurate, domain-relevant configurations [7], [8]. The scarcity of such a dataset, combined with the highly structured form of the service deployment domain, can impact the fine-tuning process. At the same time, given the task specificity, it is important to provide means to ensure that the fine-tuned model generates outputs that are not only syntactically correct, but also valid in terms of YAML structure and K8s-specific requirements. Finally, the computational resources required for fine-tuning large-scale models are relevant in both hardware requirements and costs, highlighting the need for comprehending selecting models that balance efficiency and scalability.

In this paper, we present BRAT, a solution to **B**uild a dataset of K8s manifest, **R**efine the created dataset by checking each manifest's integrity, **A**utomate the process, and **T**ranslate the introduced natural language into a structured K8s manifest.

In BRAT, we aimed to generate a dataset of K8s manifests, which were fed into different LLM (*i.e.,* `Mixtral-8x7B` [9], `Llama3-8B` [10], and `Llama3-70B` [11]) to produce corresponding intents using an increasing number of few-shot examples. To evaluate these intents, we then asked the same adopted LLM to re-generate the manifests from the intents using the same few-shot examples, as shown in Fig. 1. This process resulted in a dataset of reconstructed manifests, which were first evaluated for structural validity in YAML and then compared against the original manifests to assess the accuracy of their corresponding intents.

[1]For example see *https://kubernetes.io/docs/concepts/workloads/pods/*

Specifically, we found that increasing the number of examples can mislead the model and result in poorer accuracy for the analyzed `Llama3` models, while also introducing additional computational overhead and increasing input tokens usage. On the other hand, too few examples for `Mixtral-8x7B` lead to underperformance, as the model lacked sufficient context to generate accurate structured output. Our experiments revealed that the number of examples provided for *n*-shot learning has a significant and complex impact on the quality of the generated intents and reconstructed manifests, probably due to the heterogeneity of the K8s manifests [2]. Thus, LLMs need to be carefully evaluated to determine the optimal number of examples that balances computational efficiency with accuracy, ensuring the best similarity score to the original manifest. After creating the intent-manifest dataset, we fine-tuned `Granite-3B`, chosen for its lightweight and resource-efficient capabilities [12], improving its generation performance compared to the baseline.

To summarize, BRAT consists of two main contributions. First, a depth *n*-shot prompting analysis for the generation of an ad-hoc dataset for fine-tuning, and second, an assessment of the impact of fine-tuning `Granite-3B` for the generation of K8s manifests from intents. Thus a pipeline to lower the barrier for the practical usage of this still novel technology.

II. Related Work

In the era of Generative AI and LLMs, many studies have explored integrating these advanced models to generate network configurations [13]. An example of this combination [14] demonstrates a framework to translate intents, specified in natural language, to network configurations adapted for a Border Gateway Protocol (BGP) using different LLM (*e.g.,* GPT-4, Llama2). However, the hallucination problem of LLM, that potentially impacts the overall model's performance was not investigated there. While LLMs are powerful and versatile, their adaptability across domains can result in decreased performance when applied to specific tasks [15], [16], [17]. For this reason, researchers have started integrating techniques, known as *prompting*, into their solutions to better guide the model and produce more adapted responses. An example of this integration [18] showcases Appleseed, an intent-based system to train an LLM using few-shot examples with the goal of generating a set of executable Python programs adaptable to different use cases (*e.g.,* AWS, Google Cloud). This system, however, depends on a specific Python library, which could limit its flexibility and adaptability to other scenarios. Moving towards an intersection of LLM with Intent-based Networking for cloud-native scenarios, researchers have also focused on generating structured cloud configurations (*e.g.,* in YAML and JSON) used to automate service deployment across distributed infrastructures. The adoption of these configurations has shown seamless integration on containerized environments, microservices, and K8s-based clusters or Ansible-based automation tools [19]. An example of this integration [2] is a benchmark evaluation for generated YAML files starting from natural language inputs. This benchmark was tested on a hand-crafted dataset using different LLMs, both open-accessed, *e.g.,* `Llama2`, `Wizardcoder-v1.0` [20], and proprietary, *e.g.,* `GPT-4`, `PaLM`. Another relevant study [21] demonstrates an architecture for decomposing the intents into their Cloud/Edge and Radio Access Network (RAN) elements (*e.g.,* id, version, provider), which will then be translated by the adopted LLM to match the corresponding infrastructure-level structure and populate a Network Service Descriptor (NSD). A different approach [1] considers generating Kubernetes manifest by only performing prompt engineering and preparing customized prompts for the adopted LLM (*e.g.,* `GPT-4`, `Llama2-13B`, `Mixtral-7B`). However, the paper also shows that manual intervention might be needed for some LLMs to refine the generated manifests, which in the long term could slow the generation process, especially in large structured manifests.

Intersecting these advancements and the flexibility of adopting natural language as input and the stability of structures like JSON or YAML as output, in this paper, we present BRAT, a pipeline for creating a manifest-description dataset for fine-tuning an LLM (*i.e.,* Granite-3B), allowing users to generate Kubernetes manifests directly from natural language intents. Different from the other solutions, in BRAT we also emphasize how the choice and number of *n*-shot examples can significantly affect the generated dataset, showing how increasing examples can sometimes degrade the quality of outputs rather than improving them.

III. BRAT Methodology

This section illustrates the main steps that compose BRAT (see Fig. 1 for the overview). As mentioned previously, the main idea of BRAT is to obtain a LLM-*generator* (*Generator* from now on) that is capable of *generating* K8s manifest from a high-level user intent (*i.e.*, intents expressed in natural language). An example of a possible intent could be: *We are creating a simple Kubernetes Pod named nginx. This Pod runs a single container, also named nginx, which pulls the latest version of the nginx image. There is no extra metadata like labels or annotations included here*. To do so, we need to fine-tune the model on a dataset of intent-to-K8s manifest pairs. This process is justified by the need to reduce costs, particularly in enterprise environments, where relying on ad-hoc prompts and few-shot examples can increase input token usage. With an appropriate fine-tuning, the model can produce domain-specific outputs more efficiently, minimizing reliance on prompt engineering and thus reducing the overall computational and economic overhead associated with additional input tokens. Hence, the first step of BRAT consists of the creation of such a dataset with the help of an intermediate LLM-*descriptor* (*Descriptor* from now on). The dataset is then used for fine-tuning the *Generator* that takes intent as an input and generates a valid K8s manifest.

At the time of the experiment, we were unable to find a relevant dataset of K8s manifests, either with or without intents. This limitation motivated us to break down the first step of BRAT into three sub-tasks: *(1)* generate a set of

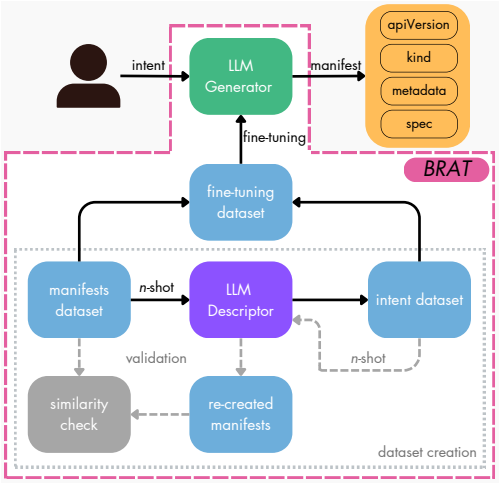


Fig. 1: Overview of BRAT's Architecture.

valid K8s manifests from a set of given templates that forms a Kubernetes Manifest Dataset; *(2)* prompt the intermediate *Descriptor* (*i.e.,* `Llama3-70B`, `Llama3-8B`, and `Mixtral-8x7B`) to generate an intent from a manifest that results in a Kubernetes Intent Dataset; *(3)* evaluate the quality of these intents using the same *Descriptors* used before, but with the goal of re-creating the original manifests when the intent is given as input. These re-created manifests are compared to the initial manifests to assess their accuracy.

A. Kubernetes Manifest Dataset

We construct the Kubernetes Manifest Dataset by combining values derived from examples of possible manifests, partially collected from High-Performance Computing (HPC) workload definitions and standard templates, organized into specific categories (*e.g.,* certification, monitoring). The example manifests have been collected in a production cluster (37 nodes) used to run various types of workloads related to scientific computation (e.g., computational chemistry, material discovery, and energy research). The data sample consists of $86,605$ pod manifests, using 1-7 containers, with various deployment characteristics including node affinity (93%) of jobs need GPU or special hardware), and network policies.

This process is divided into three distinct phases: template generation, value extrapolation, and manifest generation.

Template generation. We begin by generating a template for each K8s manifest category, preserving the original manifest structure (*e.g.,* `apiVersion`, `kind`, `spec`) while replacing all values with HELM-style placeholders. HELM was chosen for its hierarchical notation, which naturally fits the nested structure of K8s manifests [22]. To generate the templates, BRAT recursively traverses each manifest using a Depth First Search (DFS) strategy, replacing leaf values with their corresponding hierarchical paths (*e.g.,* `spec.containers.image`). During this process, unnecessary fields such as `status` and `annotations` are removed, as they are not required for deployment.

Value extrapolation. For each manifest category, we extract the unique values associated with every object by recursively traversing the manifest structure using DFS. At each step, the parent object name is concatenated with the child field to form a hierarchical key (*e.g.,* `spec.containers.name`). When a leaf node is reached, its value is stored in a dictionary, resulting in structures of the form $< string, list >$, where the key represents the hierarchical field name and the value contains all observed values for that field. This representation directly matches the previously generated HELM templates.

Manifest generation. To generate new K8s manifests, each template placeholder is replaced with a randomly selected value from its associated list. Placeholders with the same name are substituted independently to ensure heterogeneity within the manifest. Duplicate manifests are detected via hashing and automatically discarded, guaranteeing uniqueness. Finally, a unique metadata name is assigned to each generated manifest using a UUID.

B. Kubernetes Intents Dataset

After the creation of the K8s Manifests Dataset, we focus on describing each generated manifest to obtain corresponding intents by adopting three different *Descriptors*: `Llama3-70B`, `Llama3-8B`, and `Mixtral-8x7B`. After constructing the K8s Manifests Dataset, we generate corresponding intents for each manifest using three *Descriptors*: `Llama3-70B`, `Llama3-8B`, and `Mixtral-8x7B`. Each model is prompted using a combination of a structured context template (via role-based prompting) and few-shot examples, ranging from 0 to 10, to guide intent generation. When the prompt exceeds the model's token limit, we apply chunking with overlapping tokens to preserve context continuity; the resulting outputs are then merged into a single response, ensuring coherence and structural consistency with YAML [23]. To build the few-shot example set, we first generated a small number of sample manifests (Section III-A) and used the *Descriptors* to produce initial intents. These outputs were manually reviewed and curated to form the few-shot database. To mitigate hallucination risks [24], the examples intentionally vary in style and level of detail, promoting generalization and reducing overfitting.

Finally, to ensure outputs closely follow the provided examples, we tuned generation hyperparameters. Given the strong impact of *temperature* on accuracy and determinism [25], we set it to 0.3 to limit randomness while preserving flexibility. Additionally, we configured *top* k (a parameter limiting the model's choices to the k most probable tokens) to 20 and *top* p (a parameter used to select from the smallest set of tokens whose cumulative probability exceeds a threshold p) to 0.8 to further constrain token selection and improve reliability.

C. Fine-tuning Dataset

By combining the two dataset creation steps, we construct the dataset used to fine-tune the *Generator*. Based on the prompting analysis and evaluation of few-shot strategies (Section V), we generated $10{,}000$ K8s manifests and their corresponding intents using `Llama3-8B` as the *Descriptor*. Generation was performed without few-shot examples, relying

solely on carefully designed prompts to reduce input tokens and costs while maintaining high-quality intents, as supported by our evaluation. To ensure data quality, we prompted `Llama3-8B` to regenerate manifests from the generated intents and applied two validation steps: (i) discarding outputs that were not valid YAML or K8s manifests, and (ii) filtering pairs using Levenshtein similarity, retaining only those achieving at least 80% similarity with the original manifests. This process was repeated until $10{,}000$ high-quality pairs were obtained. Finally, intents and manifests were merged into an instruction–output fine-tuning dataset, a format widely adopted for its simplicity, cross-model compatibility, and effectiveness in LLM fine-tuning.

IV. Generator Fine-tuning

The choice of the LLM for the *Generator* is critical, as it directly impacts overall performance. Given the length and structured nature of the intents and K8s manifests, we focused on *instruction*-tuned models capable of handling complex structured text. We targeted models with at least 3B parameters and a context length of 5k tokens to capture long-range patterns while maintaining efficient resource usage. Based on these criteria, we selected `Granite-3B`,which provides a balanced trade-off between accuracy and computational efficiency for generating manifests from complex intents [12].

A. Implementation

To fine-tune the *Generator*, we used the Parameter-Efficient Fine-Tuning (PEFT) library [26] with the Supervised Fine-tuning (SFT) trainer on an instruction-based dataset. This setup enables efficient adaptation of large language models while limiting computational and memory overhead.

Hyperparameters settings. Fine-tuning large models is typically resource-intensive, demanding significant memory and computational time. To balance performance and efficiency, we tuned training hyperparameters using a trial-and-error approach. We trained for 1 to 4 epochs with a batch size of 8, and applied gradient accumulation to reduce memory usage, which we found to be a balanced setting in terms of computational efficiency and model performance for our dataset size. We further adopted LoRA [27] and QLoRA [28], loading the model in 4-bit precision to minimize resource consumption while preserving accuracy, together with the AdamW optimizer. This has been demonstrated to improve memory efficiency and accelerate the fine-tuning process without compromising performance [28]. To ensure output quality, we performed automatic YAML and K8s validation every 500 steps and logged training metrics every 25 steps to monitor training progress and, in case of any problems, early stop training.

Dataset configuration. We split the dataset using a standard $90/10$ train-test ratio. Given the length of the intents (about 1000 tokens on average) and the generated manifests, we set the maximum sequence length to 2048 tokens to comply with LLM constraints while avoiding context loss or overflow issues during tokenization. All components, including the dataset and tuned hyperparameters, were integrated into the SFT trainer to fine-tune the selected *Generator* with minimal computational overhead and high memory efficiency. We evaluated the model using four standard string-based metrics [29]: edit-distance (Levenshtein), cosine similarity, BLEU, and METEOR, enabling a comprehensive assessment of both syntactic and semantic similarity between generated manifests and reference outputs.

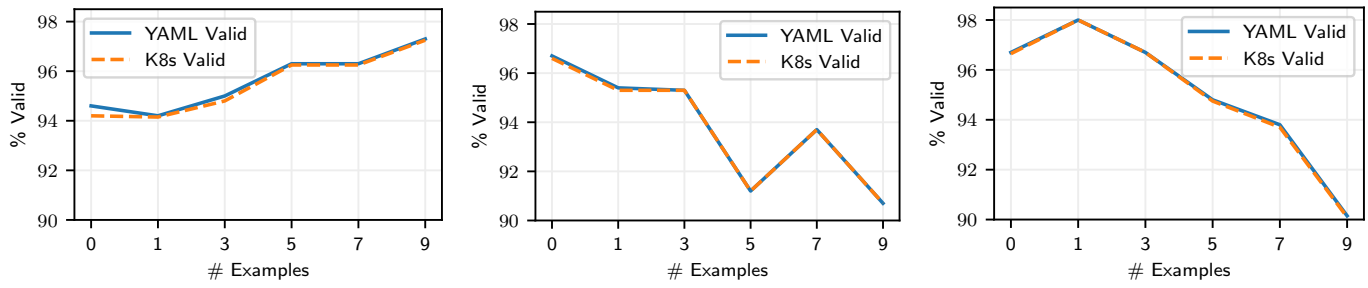
V. Evaluation

This section presents the results obtained with BRAT. We first describe the experimental setup, then report the percentage of correctly generated manifests, and finally analyze the similarity scores achieved during manifest regeneration. This evaluation strategy enabled early identification and resolution of issues in intent generation (*e.g.,* prompting, hyperparameters, and templates) before scaling the fine-tuning process. The experimental settings are organized around two aspects: *Descriptor* prompting analysis with varying n-shot examples, and fine-tuning of the *Generator*.

A. Dataset Creation

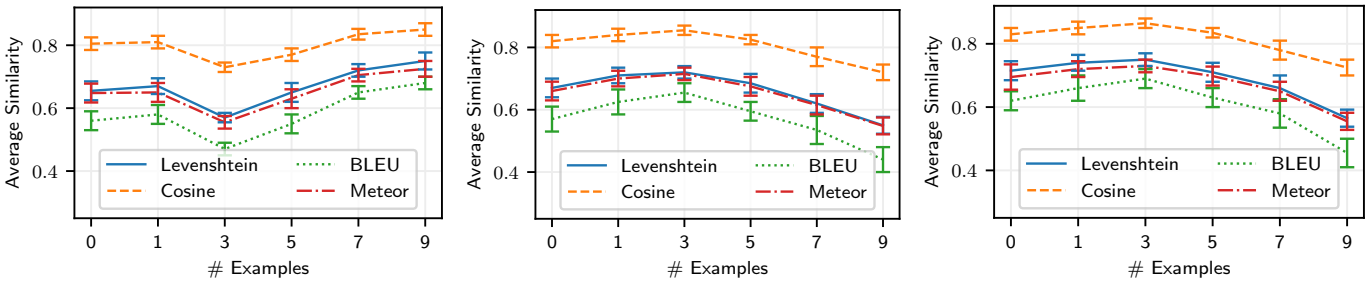
For this analysis, we evaluated three *Descriptors*: `Llama3-70B`, `Llama3-8B`, and `Mixtral-8x7B`, using an increasing number of few-shot examples from 0 to 10. Experiments were conducted in a local environment with minimal hardware requirements due to API-based execution, averaging 98.91 seconds of CPU time on a 5-core processor and only 71.94 MB of RAM.

Manifests validation. To evaluate the correctness of our synthetic manifests, we started analyzing the validity of the generated K8s manifests. This helps demonstrate how effectively our generation process aligns with the YAML and K8s structure and identifies any errors or inconsistencies that could impact the LLM response. For each generated manifest's intent, we asked each adopted LLM to generate the corresponding K8s manifest using few-shot examples in the range $[0, 10]$, using the Cartesian product of all possible combinations of LLM and few-shot examples. After each manifest is generated and before it is saved, it is analyzed using the `safe_load` function provided by the YAML Python library.This function parses the YAML structure, ensuring that the generated manifest adheres to the YAML specification and contains no syntax errors or invalid formatting before being saved. For the scope of this analysis, those YAML-valid will be saved with the *".yaml"* extension and those not compliant with the YAML structure, returning an error in the `safe_load` function, will be saved with the *"_error.yaml"* extension. This helped us count the valid ones and better investigate the invalid ones. After all the manifests were generated, we verified if the manifests were Kubernetes valid or not. To do so, we run the `kubectl`, the K8s command line tool, in conjunction with the GNU Parallel tool to speed up the process. We report the results obtained in this analysis in Fig. 2. In the figure, it is visible that all the adopted LLM (*i.e.,* `Mixtral`, `Llama3-8B`, and `Llama3-70B`) achieved



(a) (b) (c)

Fig. 2: Percentage of valid K8s manifests and YAML files generated by (a) Mixtral, (b) Llama3-8B, and (c) Llama3-70B at increasing *n*-shot examples.



(a) (b) (c)

Fig. 3: Average similarity scores for manifests generated by (a) Mixtral, (b) Llama3-8B, and (c) Llama3-70B at varying *n*-shot example numbers.

TABLE I: Average similarity scores achieved across different epochs when fine-tuning and comparison with the baseline.

Metric	**Baseline**	**Fine-tuned**			
		Epoch 1	**Epoch 2**	**Epoch 3**	**Epoch 4**
Levenshtein	0.27 ± 0.02	0.61 ± 0.015	0.63 ± 0.015	0.62 ± 0.015	0.62 ± 0.015
Cosine	0.45 ± 0.03	0.92 ± 0.010	0.93 ± 0.010	0.92 ± 0.010	0.92 ± 0.010
BLEU	0.29 ± 0.02	0.58 ± 0.012	0.59 ± 0.012	0.58 ± 0.012	0.58 ± 0.012
METEOR	0.20 ± 0.01	0.40 ± 0.010	0.40 ± 0.010	0.39 ± 0.010	0.38 ± 0.010

an exceptional percentage of correctly generated manifests. However, `Mixtral-8x7B` increases its percentage of valid manifests proportionally to the number of examples provided, a sign of how more examples can better guide the model towards an optimal response, as shown in Fig. 2a. Opposite considerations can be reached with the `Llama3` family. These models, respectively composed of `8B` and `70B`, have shown significantly higher percentages of valid manifests with fewer provided examples, indicating that more examples could somehow reduce the model's accuracy or clarity (Fig. 2b and Fig. 2c). It can also be observed (for example, in Fig. 2a) that some files were valid Kubernetes manifests but not YAML-valid. This could be caused by the fact that Kubernetes allows for a less strict syntactic validation.

Similarity to the original manifests. After this first validation step, we compared the generated manifests with the original manifests used to produce the intents. While evaluating LLMs is still considered challenging, and there are no common ways to evaluate models' responses, as mentioned earlier in BRAT we adopted the Levenshtein score, cosine similarity, BLEU score, and Meteor score. It is important to note that we reported the similarity score as a value between 0 and 1, where 0 corresponds to completely different manifests and 1 to identical ones. Each LLM received the previously generated manifest's intents with the request of generating the manifest back with an increasing number of few-shot examples (from 0 to 10). We report the results in Fig. 3. It is visible in Fig. 3a that `Mixtral` provides more accurate scores with more provided examples, coherent with the number of valid generated manifests shown in Fig. 2a. The same coherency is shown in the `Llama3` family. Both LLMs, `Llama3-8B` and `Llama3-70B`, achieve a higher similarity score with only a few examples, ranging from 0 to 3, according to the observed metric.

B. Fine-tuning of the Generator

When fine-tuning an LLM, it is essential to carefully choose the training-related hyperparameter. For `Granite-3B`, we tested epochs ranging from 1 to 4, to find the optimal that could give us the best performance, while also balancing training efficiency and model generalization, ensuring minimal overfitting while achieving the lowest validation

loss. To perform this evaluation, we compared the fine-tuned model with the validation set, appropriately created before the training. We report the result of this comparison in Table I. The figure reports the average similarity score between the generated manifest from the fine-tuned model and the expected one, using the aforementioned similarity metrics: Levenshtein, Cosine, BLEU, and METEOR score. As visible from the figure, with only 2 epochs, `Granite-3B` is able to learn from the training set, converging at its lowest loss, without leading to overfitting. When comparing the fine-tuned *Generator* with only two epochs to a baseline `Granite-3B`, it outperforms the baseline across all similarity scores, with average improvements ranging from 103% to 131%. These improvements show Granite's ability to positively react to fine-tuning, specializing in domain-specific tasks with only 2 epochs and a relatively small dataset of $10,000$ samples. This also demonstrates Granite's efficiency and adaptability, making it the right choice for resource-constrained scenarios or when large datasets are not available.

VI. Conclusion

In this paper, we presented BRAT, a novel solution that enables even less experienced users to generate customized Kubernetes manifests by introducing natural-language description intents. To do so, BRAT provides a methodology for creating an appropriate and diverse intent-K8s dataset useful for fine-tuning a chosen lightweight LLM (*i.e.,* `Granite-3B`) and providing domain-specific intelligence. We also utilized an advanced LLM to create such a dataset, adapting few-shot and template-based prompting techniques. The paper shows that smaller LLMs can outperform larger ones in some cases when only a few *n*-shot examples are provided, reducing the number of input tokens and overall costs. Furthermore, the `Granite-3B` model fine-tuned on the dataset achieves higher accuracy than the baseline, demonstrating the effectiveness of the training mechanism.

References

[1] N. Kratzke and A. Drews, "Don't train, just prompt: Towards a prompt engineering approach for a more generative container orchestration management." in *CLOSER*, 2024, pp. 248–256.

[2] Y. Xu, Y. Chen, X. Zhang, X. Lin, P. Hu, Y. Ma, S. Lu, W. Du, Z. Mao, E. Zhai *et al.*, "Cloudeval-yaml: A practical benchmark for cloud configuration generation," *Proceedings of Machine Learning and Systems*, vol. 6, pp. 173–195, 2024.

[3] Y. Ge, W. Hua, K. Mei, J. Tan, S. Xu, Z. Li, Y. Zhang *et al.*, "Openagi: When llm meets domain experts," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[4] C. Ling, X. Zhao, J. Lu, C. Deng, C. Zheng, J. Wang, T. Chowdhury, Y. Li, H. Cui, X. Zhang *et al.*, "Domain specialization as the key to make large language models disruptive: A comprehensive survey," *arXiv preprint arXiv:2305.18703*, 2023.

[5] J. Zou, M. Zhou, T. Li, S. Han, and D. Zhang, "Promptintern: Saving inference costs by internalizing recurrent prompt during large language model fine-tuning," *arXiv preprint arXiv:2407.02211*, 2024.

[6] K. VM, H. Warrier, Y. Gupta *et al.*, "Fine tuning llm for enterprise: Practical guidelines and recommendations," *arXiv preprint arXiv:2404.10779*, 2024.

[7] J. Kaddour and Q. Liu, "Text data augmentation in low-resource settings via fine-tuning of large language models," *arXiv preprint arXiv:2310.01119*, 2023.

[8] B. Zhang, Z. Liu, C. Cherry, and O. Firat, "When scaling meets llm finetuning: The effect of data, model and finetuning method," *arXiv preprint arXiv:2402.17193*, 2024.

[9] Mixtral-8x7b. Accessed: 13-01-2025. [Online]. Available: https://huggingface.co/mistralai/Mixtral-8x7B-v0.1

[10] Llama3-8b. Accessed: 13-01-2025. [Online]. Available: https://ollama.com/library/llama3:8b

[11] Llama3-70b. Accessed: 13-01-2025. [Online]. Available: https://ollama.com/library/llama3:70b

[12] M. Mishra, M. Stallone, G. Zhang, Y. Shen, A. Prasad, A. M. Soria, M. Merler, P. Selvam, S. Surendran, S. Singh *et al.*, "Granite code models: A family of open foundation models for code intelligence," *arXiv preprint arXiv:2405.04324*, 2024.

[13] A. Angi, A. Sacco, and G. Marchetto, "LLNeT: An Intent-Driven Approach to Instructing Softwarized Network Devices Using a Small Language Model," *IEEE Transactions on Network and Service Management*, vol. 22, no. 4, pp. 3403–3418, 2025.

[14] A. Fuad, A. H. Ahmed, M. A. Riegler, and T. Čičić, "An intent-based networks framework based on large language models," in *2024 IEEE 10th International Conference on Network Softwarization (NetSoft)*. IEEE, 2024, pp. 7–12.

[15] B. Xiao, B. Kantarci, J. Kang, D. Niyato, and M. Guizani, "Efficient prompting for llm-based generative internet of things," *arXiv preprint arXiv:2406.10382*, 2024.

[16] Y. Zhang, S. Qian, B. Peng, S. Liu, and J. Jia, "Prompt highlighter: Interactive control for multi-modal llms," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2024, pp. 13 215–13 224.

[17] Y. Huang, H. Du, X. Zhang, D. Niyato, J. Kang, Z. Xiong, S. Wang, and T. Huang, "Large language models for networking: Applications, enabling techniques, and challenges," *IEEE Network*, 2024.

[18] J. Lin, K. Dzeparoska, A. Tizghadam, and A. Leon-Garcia, "Appleseed: Intent-based multi-domain infrastructure management via few-shot learning," in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*. IEEE, 2023, pp. 539–544.

[19] S. Pujar, L. Buratti, X. Guo, N. Dupuis, B. Lewis, S. Suneja, A. Sood, G. Nalawade, M. Jones, A. Morari *et al.*, "Automated code generation for information technology tasks in yaml through large language models," in *2023 60th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2023, pp. 1–4.

[20] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "Wizardcoder: Empowering code large language models with evol-instruct," *arXiv preprint arXiv:2306.08568*, 2023.

[21] A. Mekrache, A. Ksentini, and C. Verikoukis, "Intent-based management of next-generation networks: an llm-centric approach," *IEEE Network*, 2024.

[22] A. Zerouali, R. Opdebeeck, and C. De Roover, "Helm charts for kubernetes applications: Evolution, outdatedness and security risks," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*. IEEE, 2023, pp. 523–533.

[23] Z. Zhou, C. Li, X. Chen, S. Wang, Y. Chao, Z. Li, H. Wang, R. An, Q. Shi, Z. Tan *et al.*, "Llm x mapreduce: Simplified long-sequence processing using large language models," *arXiv preprint arXiv:2410.09342*, 2024.

[24] Z. Ji, T. Yu, Y. Xu, N. Lee, E. Ishii, and P. Fung, "Towards mitigating llm hallucination via self reflection," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023, pp. 1827–1843.

[25] M. Renze, "The effect of sampling temperature on problem solving in large language models," in *Findings of the association for computational linguistics: EMNLP 2024*, 2024, pp. 7346–7356.

[26] G. Pu, A. Jain, J. Yin, and R. Kaplan, "Empirical analysis of the strengths and weaknesses of peft techniques for llms," *arXiv preprint arXiv:2304.14999*, 2023.

[27] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," *arXiv preprint arXiv:2106.09685*, 2021.

[28] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[29] D. Banerjee, P. Singh, A. Avadhanam, and S. Srivastava, "Benchmarking llm powered chatbots: methods and metrics," *arXiv preprint arXiv:2308.04624*, 2023.