Don't Train, Just Prompt: Towards a Prompt Engineering Approach for a More Generative Container Orchestration Management

Nane Kratzke[1][a] and André Drews[2][b]
[1]Department of Electrical Engineering and Computer Science, Lübeck University of Applied Sciences, Germany
[2]Expert Group Artificial Intelligence in Applications, Lübeck University of Applied Sciences, Germany

Keywords: Prompt Engineering, Large Language Model, cloud-native, Container, Orchestration, Automation, Intelligent, Service Management, Kubernetes, LLM, GPT-3.5, GPT-4, Llama2, Mistral, DevOps.

Abstract: **Background:** The intricate architecture of container orchestration systems like Kubernetes relies on the critical role of declarative manifest files that serve as the blueprints for orchestration. However, managing these manifest files often presents complex challenges requiring significant DevOps expertise. **Methodology:** This position paper explores using Large Language Models (LLMs) to automate the generation of Kubernetes manifest files through natural language specifications and prompt engineering, aiming to simplify Kubernetes management. The study evaluates these LLMs using Zero-Shot, Few-Shot, and Prompt-Chaining techniques against DevOps requirements and the ability to support fully automated deployment pipelines. **Results** show that LLMs can produce Kubernetes manifests with varying degrees of manual intervention, with GPT-4 and GPT-3.5 showing potential for fully automated deployments. Interestingly, smaller models sometimes outperform larger ones, questioning the assumption that bigger is always better. **Conclusion:** The study emphasizes that prompt engineering is critical to optimizing LLM outputs for Kubernetes. It suggests further research into prompt strategies and LLM comparisons and highlights a promising research direction for integrating LLMs into automatic deployment pipelines.

1 INTRODUCTION

In the evolving landscape of cloud-native computing, Kubernetes has emerged as a central tool that revolutionises how we deploy, scale, and manage containerized applications. However, the intricate architecture of Kubernetes and similar systems relies on the critical role of manifest files, which declaratively describe an intended state of operation. These manifest files serve as the blueprints for orchestrating these containers. However, managing these manifest files presents significant challenges that are often complex and require considerable expertise (Kratzke, 2023). Herein lies an untapped potential for automation and optimization.

Alongside the progress in Kubernetes, the development of Large Language Models (LLMs) has seen significant growth (Chang et al., 2023; Kaddour et al., 2023), marking their ability to process and generate text that mimics human writing (Naveed et al., 2023). These models have expanded in complexity and capabilities, opening up possibilities for programming in high-level languages. An intriguing question arises regarding their capabilities in generating declarative deployment instructions for Kubernetes or similar technologies (Zhao et al., 2023). This could simplify Kubernetes manifests, making them more accessible to DevOps engineers and potentially rendering deployment-specific languages unnecessary (Quint and Kratzke, 2019).

An underexplored or even overseen link between Kubernetes and LLMs could be prompt engineering. This approach leverages LLMs to tackle Kubernetes challenges, especially in managing manifests. Prompt engineering's applications could revolutionize cloud computing and Kubernetes management, leading to a smarter, more efficient system management approach.

This study demonstrates the potential of innovative integration of Large Language Models (LLMs) into Kubernetes management through prompt engineering. The study's **contribution** lies in its novel approach to simplifying Kubernetes operations, potentially transforming how developers and system administrators interact with Kubernetes environments to address the existing challenges in Kubernetes manifest management.

1. **Pioneering Integration:** This study is one of

[a] https://orcid.org/0000-0001-5130-4969
[b] https://orcid.org/0000-0001-9496-2531

Kratzke, N. and Drews, A.
Don't Train, Just Prompt: Towards a Prompt Engineering Approach for a More Generative Container Orchestration Management.
DOI: 10.5220/0012710300003711
Paper published under CC license (CC BY-NC-ND 4.0)
In *Proceedings of the 14th International Conference on Cloud Computing and Services Science (CLOSER 2024)*, pages 248-256
ISBN: 978-989-758-701-6; ISSN: 2184-5042
Proceedings Copyright © 2024 by SCITEPRESS – Science and Technology Publications, Lda.

the first explorations into using state-of-the-art (non-fine-tuned) LLMs for streamlining Kubernetes manifest generation, setting a precedent for future research and practical applications. Recent studies like (Lanciano et al., 2023), (Xu et al., 2023) , and (Komal et al., 2023) focus mainly on fine-tuned LLMs.

2. **Setting a Framework for Future Innovations:** The study lays a framework for how AI and machine learning, especially LLMs, can be leveraged in other areas of cloud computing and IT infrastructure management, opening doors for further innovations and research.

Overall, this study contributes to the technical field of container orchestration but also adds to the growing body of knowledge on the practical applications of LLMs and prompt engineering in technology and cloud-native computing in general.

2 BACKGROUND AND RELATED WORK

In Kubernetes. manifest files, primarily in YAML or JSON, define the desired state of operations, including pods, services, and controllers. These files are crucial for deploying and managing applications within Kubernetes. However, managing these files becomes challenging with scaling, including maintaining configuration consistency, updating features, and ensuring security compliance. The complexity increases with many microservices (Tosatto et al., 2015; Sultan et al., 2019). Integrating AI and machine learning, mainly through large language models, offers potential improvements in managing and generating these manifests by automating tasks and optimizing configurations, promising to simplify management and enhance container orchestration efficiency and reliability.

Large Language Models: Like OpenAI’s GPT series, LLMs have significantly advanced natural language processing with their ability to understand, generate, and manipulate written texts. These models have evolved from simple beginnings to complex systems with remarkable linguistic abilities, shifting from rule-based to neural network architectures that learn from extensive data to produce context-rich text (Petroni et al., 2019). Their role in automation and data processing is expanding, enabling the automation of complex language tasks such as document summarization, code generation, language translation, and content creation (Hou et al., 2023). LLMs analyze text to extract insights and trends in data processing, supporting business and technology strategies. Their precise language processing capabilities offer potential in domains like healthcare, finance, customer service, and system management, like Kubernetes, where they can streamline manifest file generation, error diagnosis, and configuration optimization, reducing manual work and enhancing efficiency.

Prompt Engineering: However, training LLMs for domain-specific purposes involves a computationally intensive pre-training phase for general language understanding, followed by a problem-specific fine-tuning phase. Recently, the focus has shifted towards a ”pre-train, prompt, predict” approach, reducing computational effort and requiring specialized datasets using prompt engineering (Liu et al., 2023; Chen et al., 2023). This prompt engineering technique involves designing strategic inputs (prompts) to guide LLMs in generating desired outcomes. In the context of Kubernetes, prompt engineering could significantly improve LLMs’ ability to handle technical tasks, such as generating or optimizing manifest files, diagnosing deployment issues, and suggesting configuration practices without task-specific fine-tuning. Although not systematically explored, prompt engineering offers a promising method for making Kubernetes management more intuitive and efficient, potentially lowering technical barriers and enhancing system reliability.

Related Work: The current research on integrating LLMs with Kubernetes shows promising but limited approaches. Lanciano et al. propose using specialized LLMs to analyze Kubernetes deployment files and aid non-experts in design and quality assurance (Lanciano et al., 2023). Xu et al. introduce CloudEval-YAML, a benchmark for evaluating LLMs in generating cloud-native application code, focusing on YAML and providing a dataset with unit tests (Xu et al., 2023). Kowal et al. suggest a pipeline utilizing LLMs for anomaly detection and auto-remediation in microservices to enhance system stability. These methods typically rely on training specialized LLMs (Komal et al., 2023). Our research, however, investigates the use of standard LLMs combined with straightforward prompt engineering to automate Kubernetes configurations for security and compliance, distinguishing from the reliance on specialized models.

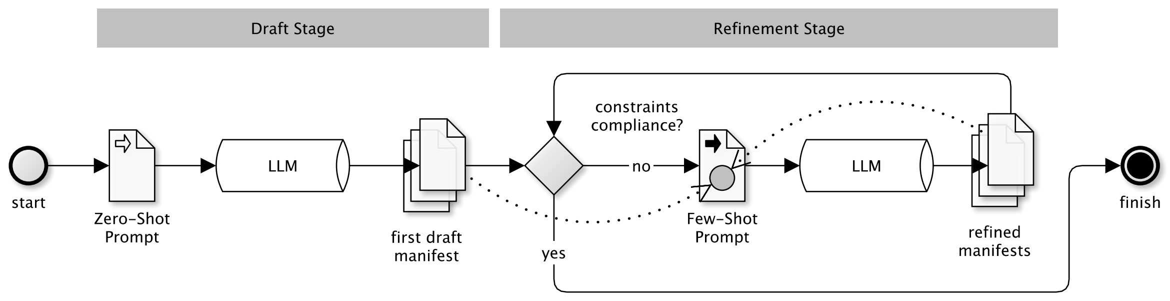

Figure 1: Analyzed prompt chain composed of a drafting Zero-Shot and an iteratively refining Few-Shot stage.

3 METHODOLOGY

We aim to leverage standard LLMs like GPT-3.5/4 and Llama2 across various sizes (7B, 13B, 70B) to automate Kubernetes manifest file generation and management without specific fine-tuning. We aim to use the inherent knowledge base of these LLMs (Petroni et al., 2019) to create accurate Kubernetes configurations. We explored different LLMs and prompt engineering techniques to assess their suitability for this task. Advanced prompt engineering can guide LLMs to understand Kubernetes manifest intricacies better, ensuring best practices in container security and operations. This research intends to connect LLMs' advanced language capabilities with the technical requirements of Kubernetes management, aiming for improved DevOps efficiency and security in Kubernetes operations.

3.1 Research Questions

Although prompt engineering is still very young and dynamic, several distinct approaches exist to different prompting techniques that can be derived from existing prompt engineering overviews (Liu et al., 2023). The following methods seem very promising from the current state of knowledge and were used to derive our research questions.

Zero-Shot: Large LLMs are tuned to follow instructions and are pre-trained on large amounts of data to perform some tasks out of the box (zero-shot). For example, an LLM can generate text with a single prompt without any required specifications as input. This works astonishingly well for simple tasks like categorization (Wei et al., 2021).
RQ1: *We want to determine how well LLMs can generate Kubernetes manifests out-of-the-box.*

Few-Shot: Although LLMs demonstrate remarkable zero-shot capabilities, they fall short on more complex tasks when using the zero-shot setting. In these cases, prompting can enable in-context learning where we provide a guess of expected output text within a prompt, so-called demonstrations (e.g., Kubernetes manifest files) to steer the model to better performance. The demonstrations serve as conditioning for subsequent examples where we would like the model to generate a response. According to (Touvron et al., 2023a), few shot prompting needs models with sufficient size (Kaplan et al., 2020).
RQ2: *We are therefore interested in seeing whether larger LLMs produce better results in Few-Shot settings.*

Prompt-Chaining: To enhance the performance and reliability of LLMs, an essential prompt engineering technique involves breaking down complex tasks into smaller, manageable subtasks. This approach starts by prompting the LLM with one subtask at a time. The response generated from each subtask becomes the sequence's input for the following prompt. This method of sequentially linking prompts allows the LLM to tackle complex tasks that might be challenging to address in a single, comprehensive prompt. Prompt chaining not only improves the LLM's ability to handle intricate tasks but also increases the transparency and controllability of LLM applications. This approach makes debugging and analysing the model's responses at each stage easier, facilitating targeted improvements where needed. A frequently used framework in this context is LangChain (Topsakal and Akinci, 2023).
RQ3: *We want to determine whether Kubernetes manifests can be gradually refined with prompt chaining in order to add capabilities that LLMs do not "retrieve from their memory" by default in Zero-Shot settings.*

Further Prompting Techniques: The techniques mentioned above seem the most promising for an initial explorative analysis based on the current state of knowledge. Nevertheless, techniques such as Chain-of-Thought (Wei et al., 2022; Kojima et al., 2022), Self-Consistency (Wang et al., 2022), Generated Knowledge Prompting (Liu et al., 2021), Tree of Thoughts (Yao et al., 2023; Long, 2023; Hulbert, 2023), Automatic Reasoning and Tool-use (Paranjape et al., 2023), Program-Aided Language Models (Gao et al., 2022), ReACT Prompting (Yao et al., 2022) , and Retrieval Augmented Generation (Lewis et al., 2020) should also be investigated in a systematic screening in the future.

3.2 Analyzed Use Case (NoSQL DB)

We examine basic prompt engineering methods like Zero-/Few-Shot and Prompt-Chaining to assess if non-fine-tuned LLMs (e.g., GPT-3.5, GPT-4, Llama2, Mistral) can efficiently generate Kubernetes manifest files. Our goal is to determine the effectiveness of these LLMs and identify which prompt engineering techniques, as discussed in Sec. 3.1, are most effective for designing and optimizing manifest generation. Our exploratory approach centred on deploying and operating a NoSQL database (MongoDB or similar systems) within Kubernetes, adhering to typical real-world constraints, including those from the "Kubernetes Security Hardening Guide".

- The database or application containers should not be executed with privileges (`securityContext.priviledged: false`)
- The database/application should only be accessible from within its namespace (so a correct `NetworkPolicy` should be generated).
- The database/application containers should not be able to monopolise resources (so memory and CPU resource limits should be generated)

Furthermore, we expect the LLM to derive the required manifests, even if these are not explicitly requested in the prompt. An experienced DevOps engineer would have developed manifests for the above-mentioned setting. We use this DevOps experience as a frame of reference for our expectations of LLMs.

- Deployment (or `StatefulSet` including a `PersistentVolumeClaimTemplate`)
- Correct Volume mounts in `Deployment`/`StatefulSets`
- `PersistentVolumeClaim` (unless the LLM decides in favour of a `StatefulSet`)
- `Service`

3.3 Generation and Evaluation Strategy

Would the following prompt

```
Create required manifests to deploy a MongoDB
database in Kubernetes.
```

be passed to an LLM, the following or a similar Kubernetes manifest file would be generated.

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: mongo
    labels:
        app: mongo
spec:
    replicas: 1
    selector:
        matchLabels:
            app: mongo
    template:
        metadata:
            labels:
                app: mongo
        spec:
            containers:
            - name: mongo
              image: mongo
              ports:
              - containerPort: 27017
```

This example, produced by GPT-4 and slightly modified for clarity, demonstrates the model's ability to generate a complete deployment manifest using a basic zero-shot approach, including a `PersistentVolumeClaim` and a `Service` manifest. This indicates that certain LLMs can create valid Kubernetes manifest files independently without requiring specialized tuning. Our evaluation employed a prompt chain (as illustrated in Fig. 1) that begins with a zero-shot prompt to create initial manifests. This is followed by a second phase of iterative refinement to ensure operation constraints are met, using a specific check and refinement prompt template.

```
Please refine the following Kubernetes manifests.
{{CHECK PROMPT}}
Manifests (YAML):
{{MANIFESTS}}
Refined manifests (YAML):
```

The following check and refinement prompts[1] (`CHECK PROMPT`) were applied in the refinement stage in the following order:

1. `Check whether a Deployment manifest has been generated for the database.`
2. `Check whether a PersistentVolumeClaim manifest has been generated for the database.`

[1]Slightly shortened for presentation.

3. `Check whether the PersistentVolumeClaim manifest has been mounted within the database container.`
4. `Assure that a container has a securityContext set to privileged false.`
5. `Assure that containers have sensible resource/limit settings.`
6. `Check whether a service manifest that addresses the database port has been generated.`
7. `Check whether a NetworkPolicy exists that makes the database port only accessible from the namespace.`

The resulting manifests in the draft stage and the refinement stage were analysed by Kubernetes experts and tools (`kubectl apply --dry-run`) to determine whether the generated manifests **a)** adequately describe the situation and **b)** are valid and Kubernetes deployable (`kubectl apply`).

If a manifest adequately fits an orchestration responsibility by criterion **a)** (e.g., it includes a required deployment manifest), one point was awarded in each case; otherwise, 0 points were awarded. If criterion b) was violated and a manual correction was required, the number of points was halved for each required correction (i.e., $1/2$ point for one error, $1/4$ point for two errors, etc.).

This assessment was both manual and automated. The Kubernetes command-line tool `kubectl` automatically determined whether the generated manifests were syntactically correct and deployable. A DevOps expert corrected and counted errors detected by the tool. Corrections had to be made so that a minimum number of changes led to a deployable result. In our further research, we strive to automate these manual analysis steps, on the one hand, to increase the objectivity of the assessment and, on the other hand, to be able to evaluate larger deployments and data sets. Nevertheless, this semi-automated approach was sufficient for our initial analysis to derive a research position and direction.

We have made this evaluation for the following manifest generation strategies, which are based on Fig. 1.

1. **Zero-Shot:** The prompt did not explicitly specify the constraints to be met. Therefore, the refinement stage in Fig. 1 was not run.
2. **Zero-Shot+Constraints:** The constraints to be met were explicitly specified in the prompt (see CHECK PROMPTS above). However, no incremental refinement was carried out constraint by constraint. This means the refinement stage of Fig. 1 was not run through here either.
3. **Few-Shot+Refinement:** The prompt did not specify the constraints to be met. However, the draft stage results were explicitly refined iteratively for each constraint in the refinement stage of Fig. 1.

The main difference between **Zero-Shot+Constraints** and **Few-Shot+Refinement** is that in the first case, an LLM must take all constraints into account at once, and in the second case, it can process and improve constraint by constraint.

4 RESULTS

The question is which strategy best fulfils all the constraints to be complied with and whether there are differences between the LLMs. The results can be seen in Fig. 2 and in more detail in the Appendix in Tables 1, 2, and 3. The following models were selected for an explorative evaluation based on their current popularity (OpenAI) or reported performance in the case of self-hostability (Llama2, Mistral).

- **GPT-4** (Achiam et al., 2023), OpenAI managed service (operation details unknown)
- **GPT-3.5-turbo** (Ye et al., 2023), OpenAI managed service (operation details unknown)
- **Llama2 13B** (Touvron et al., 2023b), Quantization: AWQ, self-hosted, Memory: 46.8Gi of an Nvidia A6000 with 48Gi
- **Llama2 7B** (Touvron et al., 2023b), Quantization: AWQ, self-hosted, Memory: 14.7Gi of an Nvidia A4000 or A2 with 16Gi
- **Mistral 7B** (Jiang et al., 2023), Quantization: AWQ, self-hosted, Memory: 10.8Gi of an Nvidia A4000 or A2 with 16Gi (fine-tuned for instruct and coding assistance functionalities)

All self-hosted machine learning models were run via HuggingFace's Text Generation Inference Interface, enabling AWQ quantisation (Lin et al., 2023). We worked with the non-fine-tuned basic models from HuggingFace, except for the Mistral model. In the case of Mistral, we explicitly worked with a model tuned for coding assistance to better assess possible fine-tuning effects. The models were used programmatically using the LangChain library[2] and OpenAI[3]

[2]https://pypi.org/project/langchain

[3]https://pypi.org/project/langchain-openai

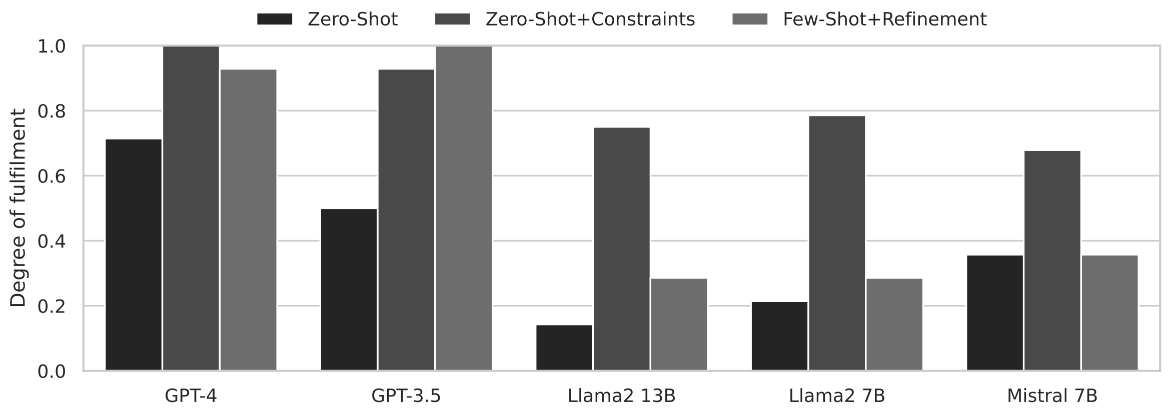

Figure 2: Which degrees of fulfilment were achieved with which LLM and prompt strategies? Experiments were not repeated because the temperature parameter was set to 0.0. Details are in the Appendix.

or the Text Generation Inference Interface from HuggingFace[4]. We used the LangChain default values and set the temperature parameter to 0.

4.1 Explanation of Results

Fig. 2 shows the degree of fulfilment achieved. All LLMs were able to create a working deployment. However, compliance with the desired operation constraints varied.

Fulfilment was measured on a scale from 0.0 (no requirements met) to 1.0 (all requirements met), with values of 1.0 indicating the possibility of fully automatic, error-free deployment in Kubernetes – values below 1.0 required manual corrections, detailed in the appendix.

GPT-4 and GPT-3.5 achieved the highest fulfilment scores, indicating their capability for fully automatic deployment. The free Llama2 and Mistral models had lower fulfilment levels, with more straightforward Zero-Shot approaches outperforming iterative refinement strategies. Notably, the smaller 7B Llama2 model performed comparably or slightly better than its 13B counterpart, and the most miniature Mistral model outperformed Llama2 in Zero-Shot tasks but not when operation constraints were integrated into the prompt.

4.2 Discussion of Results

What conclusions can be drawn?

RQ1: ***How Well Can LLMs Generate Kubernetes Manifests out-of-the-box?*** All LLMs generated Kubernetes manifest files correctly, accurately considering semantic relationships between concepts like Deployment, PersistentVolumeClaim, Service, and NetworkPolicy. However, most cases required some manual adjustments. Notably, the two commercial GPT series models demonstrated the potential for fully automated database deployment without user intervention. Coding-optimized models like Mistral performed better on simple Zero-Shot prompts than Llama2 but did not surpass GPT-series models. However, further investigation is needed to assess the generalizability of these findings.

RQ2: ***Generate larger LLMs better results in Zero-/Few-Shot settings?*** Commercial LLMs like GPT-4 and GPT-3.5 outperform free models such as Llama2 and Mistral, though larger model sizes (13B) do not necessarily yield better results than their smaller counterparts (7B). Our results indicate that result quality depends on model size and training data; for instance, Mistral, optimized for coding tasks, performs better in Zero-Shot tasks than Llama2. Similarly, GPT-3.5 and GPT-4’s superior performance likely stems from more extensive training data. Prompt engineering emerges as a crucial factor, with proper techniques allowing free models to nearly match GPT-4’s performance, suggesting that further research should explore prompt engineering’s role in enhancing LLM outcomes.

RQ3: ***Is it worthwhile to gradually refine Kubernetes manifests with prompt chaining?*** Our study initially posited that iterative refinement would enhance Kubernetes manifest quality across various LLMs by addressing specific optimization aspects. Contrary to expectations, the outcomes diverged significantly. Incremental refinement showed minimal benefits for commercial models like GPT-4 and GPT-3.5, which already performed well with basic prompt engineering. Conversely, this approach negatively impacted the performance of free models like Llama2 and Mistral, possibly due to the overwriting of ear-

[4]https://pypi.org/project/text-generation

lier optimizations over seven iterations. This outcome suggests that the predefined structure of manifest files limits the effectiveness of iterative refinements due to their low complexity. Interestingly, the final refinement stage focusing on SecurityPolicy yielded effective policies, raising questions about the iterative strategy's optimality and the organization of result integration. This discrepancy highlights a potential area for future research, particularly the impact of increasing complexity on the effectiveness of refinement prompts. These findings can be used to optimize the refinement stage. The design and sequence of the individual steps appear to influence the results significantly, and care must be taken to ensure that later refinement steps do not ”overwrite” previous ones.

4.3 Limitations to Consider

This study investigates prompt engineering in Kubernetes, specifically assessing Large Language Models' (LLMs) capabilities in expressing Kubernetes operational states via YAML. Our findings, which focus on single-component and typical database deployments, are preliminary and context-specific, cautioning against their generalization for broader LLM performance assessments. Acknowledging the exploratory nature of our work, we emphasize its role in laying foundational knowledge for future, more complex studies. Although our initial research aligns with existing literature, highlighting the utility of LLMs in DevOps, it deliberately avoids the challenges of multi-service, interconnected deployments to ensure a solid baseline for subsequent investigation. Our phased research approach has been designed to enhance our methodical understanding of LLMs and Kubernetes deployments, setting the stage for a comprehensive exploration of these technologies' interplay in future studies.

5 CONCLUSION AND OUTLOOK

LLMs demonstrated the capability to generate Kubernetes manifests from natural language, with GPT-4 and GPT-3.5 showing promise for fully automated deployments. Interestingly, model size was not a definitive indicator of performance, as smaller models like Llama2 and Mistral 7B occasionally surpassed their larger counterparts, highlighting the importance of training data and optimization. Future work will explore more complex deployments, focusing on security and API integrations, to further understand LLMs' potential. Nevertheless, the study's reliance on partly manual YAML evaluation as a primary assessment method can be seen as a limitation for large-scale analysis. Therefore, intensified automated validation techniques are planned to improve the research's robustness and objectivity. Prompt engineering played a crucial role in enhancing the performance of models like Llama2, though its effectiveness varied. This study underscores the potential of LLMs in automating Kubernetes deployments and suggests a focus on prompt optimization and LLM comparison for integration into deployment pipelines. It might even question the ongoing necessity of traditional DevOps roles as LLM capabilities advance.

ACKNOWLEDGEMENTS

We thank Kirill Fuks for his inspiring seminar talk on prompt engineering and Niklas Beuter, Ralph Hänsel, and Christian Többermann for providing GPU resources via JupyterHub. We also thank Jonas Flodin for managing JupyterHub on our Kubernetes infrastructure. Your support and expertise made this study possible.

REFERENCES

Achiam, O. J., Adler, S., Agarwal, S., ..., and Zoph, B. (2023). Gpt-4 technical report.

Chang, Y., Wang, X., Wang, J., Wu, Y., Zhu, K., Chen, H., Yang, L., Yi, X., Wang, C., Wang, Y., et al. (2023). A survey on evaluation of large language models. *arXiv preprint arXiv:2307.03109*.

Chen, B., Zhang, Z., Langrené, N., and Zhu, S. (2023). Unleashing the potential of prompt engineering in large language models: a comprehensive review. *arXiv preprint arXiv:2310.14735*.

Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. (2022). Pal: Program-aided language models. *ArXiv*, abs/2211.10435.

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., Luo, X., Lo, D., Grundy, J., and Wang, H. (2023). Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*.

Hulbert, D. (2023). Using tree-of-thought prompting to boost chatgpt's reasoning.

Jiang, A. Q., Sablayrolles, A., Mensch, A., ..., and Sayed, W. E. (2023). Mistral 7b. *ArXiv*, abs/2310.06825.

Kaddour, J., Harris, J., Mozes, M., Bradley, H., Raileanu, R., and McHardy, R. (2023). Challenges and applications of large language models. *arXiv preprint arXiv:2307.10169*.

Kaplan, J., McCandlish, S., Henighan, T. J., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. (2020). Scaling laws for neural language models. *ArXiv*, abs/2001.08361.

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. (2022). Large language models are zero-shot reasoners. *ArXiv*, abs/2205.11916.

Komal, S., Zakeya, N., Raphael, R., Harit, A., Mohammadreza, R., Marin, L., Larisa, S., and Ian, W. (2023). Adarma auto-detection and auto-remediation of microservice anomalies by leveraging large language models. In *Proceedings of the 33rd Annual International Conference on Computer Science and Software Engineering*, CASCON '23, page 200–205, USA. IBM Corp.

Kratzke, N. (2023). *Cloud-native Computing: Software Engineering von Diensten und Applikationen für die Cloud*. Carl Hanser Verlag GmbH Co KG.

Lanciano, G., Stein, M., Hilt, V., Cucinotta, T., et al. (2023). Analyzing declarative deployment code with large language models. *CLOSER*, 2023:289–296.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., and Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, NIPS'20, Red Hook, NY, USA. Curran Associates Inc.

Lin, J., Tang, J., Tang, H., Yang, S., Dang, X., and Han, S. (2023). Awq: Activation-aware weight quantization for llm compression and acceleration. *ArXiv*, abs/2306.00978.

Liu, J., Liu, A., Lu, X., Welleck, S., West, P., Bras, R. L., Choi, Y., and Hajishirzi, H. (2021). Generated knowledge prompting for commonsense reasoning. In *Annual Meeting of the Association for Computational Linguistics*.

Liu, P., Yuan, W., Fu, J., Jiang, Z., Hayashi, H., and Neubig, G. (2023). Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Journals, Vol 55, Issue 9*.

Long, J. (2023). Large language model guided tree-of-thought. *ArXiv*, abs/2305.08291.

Naveed, H., Khan, A. U., Qiu, S., Saqib, M., Anwar, S., Usman, M., Barnes, N., and Mian, A. (2023). A comprehensive overview of large language models. *arXiv preprint arXiv:2307.06435*.

Paranjape, B., Lundberg, S. M., Singh, S., Hajishirzi, H., Zettlemoyer, L., and Ribeiro, M. T. (2023). Art: Automatic multi-step reasoning and tool-use for large language models. *ArXiv*, abs/2303.09014.

Petroni, F., Rocktäschel, T., Lewis, P., Bakhtin, A., Wu, Y., Miller, A. H., and Riedel, S. (2019). Language models as knowledge bases? *arXiv preprint arXiv:1909.01066*.

Quint, P.-C. and Kratzke, N. (2019). Towards a lightweight multi-cloud dsl for elastic and transferable cloud-native applications.

Sultan, S., Ahmad, I., and Dimitriou, T. (2019). Container security: Issues, challenges, and the road ahead. *IEEE access*, 7:52976–52996.

Topsakal, O. and Akinci, T. C. (2023). Creating large language model applications utilizing langchain: A primer on developing llm apps fast. *International Conference on Applied Engineering and Natural Sciences*.

Tosatto, A., Ruiu, P., and Attanasio, A. (2015). Container-based orchestration in cloud: state of the art and challenges. In *2015 Ninth international conference on complex, intelligent, and software intensive systems*, pages 70–75. IEEE.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. (2023a). Llama: Open and efficient foundation language models. *ArXiv*, abs/2302.13971.

Touvron, H., Martin, L., ..., and Scialom, T. (2023b). Llama 2: Open foundation and fine-tuned chat models. *ArXiv*, abs/2307.09288.

Wang, X., Wei, J., Schuurmans, D., Le, Q., hsin Chi, E. H., and Zhou, D. (2022). Self-consistency improves chain of thought reasoning in language models. *ArXiv*, abs/2203.11171.

Wei, J., Bosma, M., Zhao, V., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., and Le, Q. V. (2021). Fine-tuned language models are zero-shot learners. *ArXiv*, abs/2109.01652.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., hsin Chi, E. H., Xia, F., Le, Q., and Zhou, D. (2022). Chain of thought prompting elicits reasoning in large language models. *ArXiv*, abs/2201.11903.

Xu, Y., Chen, Y., Zhang, X., Lin, X., Hu, P., Ma, Y., Lu, S., Du, W., Mao, Z. M., Zhai, E., et al. (2023). Cloudeval-yaml: A realistic and scalable benchmark for cloud configuration generation.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. (2023). Tree of thoughts: Deliberate problem solving with large language models. *ArXiv*, abs/2305.10601.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. (2022). React: Synergizing reasoning and acting in language models. *ArXiv*, abs/2210.03629.

Ye, J., Chen, X., Xu, N., ..., and Huang, X. (2023). A comprehensive capability analysis of gpt-3 and gpt-3.5 series models. *ArXiv*, abs/2303.10420.

Zhao, X., Lu, J., Deng, C., Zheng, C., Wang, J., Chowdhury, T., Yun, L., Cui, H., Xuchao, Z., Zhao, T., et al. (2023). Domain specialization as the key to make large language models disruptive: A comprehensive survey. *arXiv preprint arXiv:2305.18703*.

APPENDIX

Table 1: Analyzed results of Zero-Shot draft manifest generation for different LLMs without explicitly mentioned constraints.

	GPT-4	GPT-3.5	Llama2 (13B)	Llama2 (7B)	Mistral (7B)
1. Deployment	OK (1P)	OK (1P)	Missing Image (1 fix, 0.5P)	OK (1P)	OK (1P)
2. Volume Claim	OK (1P)	-	-	-	-
3. Volume Mount	OK (1P)	EmptyDir (0.5P)	EmptyDir (0.5P)	Missing claim (1 fix, 0.5P)	Missing claim (1 fix, 0.5P)
4. Non priviledged sec. context	-	-	-	-	-
5. Requests/Limits	-	-	-	-	-
6. Service	OK (1P)	OK (1P)	-	-	OK (1P)
7. Network Policies	-	-	-	-	-
Coverage of Criteria	5/7	3.5/7	1/7	1.5/7	2.5/7

Table 2: Analyzed results of Zero-Shot draft manifest generation for different LLMs with explicitly mentioned constraints.

	GPT-4	GPT-3.5	Llama2 (13B)	Llama2 (7B)	Mistral (7B)
1. Deployment	OK (1P)	OK (1P)	OK (1P)	Missing port (1 fix, 0.5P)	OK (1P)
2. Volume Claim	OK (1P)	OK (1P)	Not generated (0P)	Wrong class (1 fix, 0.5P)	Wrong class (2 fixes, 0.25P)
3. Volume Mount	OK (1P)	OK (1P)	OK (1P)	OK (1P)	Syntax (1 fix, 0.5P)
4. Non priviledged sec. context	OK (1P)	OK (1P)	OK (1P)	OK (1P)	Syntax (1 fix, 0.5P)
5. Requests/Limits	OK (1P)	OK (1P)	OK (1P)	OK (1P)	OK (1P)
6. Service	OK (1P)	OK (1P)	OK (1P)	Wrong port (1 fix, 0.5P)	OK (1P)
7. Network Policies	Wrong selector (1 fix, 0.5P)	OK (1P)	Wrong definition (2 fixes, 0.25P)	Wrong selector (1 fix, 0.5P)	Wrong selector (1 fix, 0.5P)
Coverage of Criteria	6.5/7	7/7	5.25/7	5.5/7	4.75/7

Table 3: Analyzed results of Few-Shot constraint refinements for different LLMs.

	GPT-4	GPT-3.5	Llama2 (13B)	Llama2 (7B)	Mistral (7B)
1. Deployment	OK (1P)	OK (1P)	OK (1P)	OK (1P)	OK (1P)
2. Volume Claim	OK (1P)	OK (1P)	Not generated (0P)	Not generated (0P)	Not generated (0P)
3. Volume Mount	OK (1P)	OK (1P)	OK (1P)	OK (1P)	EmptyDir (0.5P)
4. Non priviledged sec. context	OK (1P)	OK (1P)	Not generated (0P)	Not generated (0P)	Not generated (0P)
5. Requests/Limits	OK (1P)	OK (1P)	Not generated (0P)	Not generated (0P)	Not generated (0P)
6. Service	OK (1P)	OK (1P)	Not generated (0P)	Not generated (0P)	OK (1P)
7. Network Policies	OK (1P)	Wrong selector (1 fix, 0.5P)	Not generated (0P)	Not generated (0P)	Not generated (0P)
Coverage of Criteria	7/7	6.5/7	2/7	2/7	2.5/7