Metadata Is Not Enough: Characterizing Pre-Execution Issues in Mixtral-Generated Kubernetes Validation Commands

Bento Rafael Siqueira
Federal University of Lavras
Lavras, Minas Gerais, Brazil
bento.siqueira@ufla.br

Samira Santos da Silva
Federal University of Lavras
São Sebastião do Paraíso, Minas Gerais, Brazil
samirasilva@ufla.br

Johnatan Alves Oliveira
Federal University of Lavras
Lavras, Minas Gerais, Brazil
johnatan.oliveira@ufla.br

Juka Francis Pereira Gonçalves
Federal University of Lavras
São Sebastião do Paraíso, Minas Gerais, Brazil
juka.goncalves@estudante.ufla.br

Maria Júlia Pedroso Pimenta
Federal University of Lavras
São Sebastião do Paraíso, Minas Gerais, Brazil
maria.pimenta3@estudante.ufla.br

Fabiano Cutigi Ferrari
Federal University of São Carlos
São Carlos, São Paulo, Brazil
fcferrari@ufscar.br

Abstract

Cloud-native deployments evolve through manifests and operational checks that must remain synchronized with declared resources and runtime assumptions. We report an exploratory pre-execution audit of 34 validation files and 178 step-level CLI command units produced with Mixtral-8x7B-Instruct-v0.1 for 10 evolution comparisons of a Kubernetes-based financial application. The corpus is descriptive, single-model evidence and does not include a quota-balanced generic-prompt baseline. Among all command units, 138 (77.5%) referenced at least one declared Kubernetes resource; the complementary group includes application-level commands for which such a reference is not necessarily expected. Moreover, 136 commands (76.4%) used `kubectl`. Eight commands (4.5%) retained unresolved placeholders, and 11 (6.2%) referred to undeclared resources, with 9 of those 11 occurrences concentrated in one comparison. A further 143 commands (80.3%) explicitly encoded `-n default`; because this namespace is correct for the studied manifests, we treat it as a portability-related binding rather than an error. These selected textual properties show that manifest-focused commands can coexist with unresolved or undeclared references. Because the commands were not executed, the study does not assess executability, validation effectiveness, or fault detection. The findings therefore frame metadata as a focusing signal—not as evidence that selected pre-execution issues are absent—and motivate runtime-aware grounding, structured command schemas, and post-generation checks for CI/CD and GitOps workflows.

Keywords

Large language models, Prompt engineering, Kubernetes, Cloud-native systems, Metadata grounding, Infrastructure-aware testing, Software maintenance, Software evolution

1 Introduction

Cloud-native systems are maintained through both source-code changes and the continuing evolution of deployment manifests, service definitions, secrets, resource constraints, and operational policies. In Kubernetes, those artifacts determine how a system is deployed, inspected, debugged, and validated [4, 9, 10]. Large Language Models (LLMs) can assist code, test, Infrastructure as Code (IaC), and cloud-configuration tasks [3, 5, 7, 14, 16]; however, a plausible validation command may still contain an outdated resource name, an invalid selector, a missing secret, or unresolved input. Deployment-aware validation consequently depends on both manifest semantics and runtime context, making synchronization between evolving manifests and generated commands a maintenance concern.

This paper investigates metadata-grounded generation of Kubernetes validation commands from a software maintenance and evolution perspective. Rather than asking whether deployment metadata makes commands more specific, we characterize selected pre-execution issues that remain after metadata is provided. Prior LLM-based software engineering work shows that prompt context influences generated outputs [5, 15]; our narrower question is which properties can be checked against manifests before any cluster is available, and which assumptions remain dependent on runtime context.

Accordingly, the contribution is neither a claim that metadata-grounded prompting outperforms generic prompting nor a measurement of command executability. It is a command-level audit of manifest-checkable properties in one Mixtral-8x7B-Instruct-v0.1 corpus.

The study is guided by the following research questions:

- **RQ1.** To what extent do metadata-grounded LLM-generated validation commands refer to Kubernetes resources declared in the inspected deployment manifests?
- **RQ2.** Which selected pre-execution issues and environment bindings appear in the generated Kubernetes validation commands?
- **RQ3.** How do these grounding indicators vary across deployment evolution comparisons?

We analyze 11 prepared Kubernetes deployment versions of FinSim, covering 10 evolution comparisons. A deterministic extractor converted each manifest pair into structured records that were inserted into a fixed prompt template for Mixtral-8x7B-Instruct-v0.1. We then inspected every generated command unit for placeholder persistence, explicit namespace binding, declared and undeclared resource references, and command type. All quantitative claims are descriptive and scoped to this unbalanced, metadata-grounded corpus.

The study makes the following contributions as a lightweight pre-execution audit of LLM-generated Kubernetes validation artifacts:

- an exploratory empirical characterization of 34 generated Kubernetes validation files and 178 step-level CLI commands across 10 deployment evolution comparisons;
- a set of lightweight operational indicators for auditing metadata-grounded validation artifacts, including resource grounding, namespace binding, placeholder persistence, and undeclared resource references;
- a characterization of selected pre-execution issues, showing that manifest-focused commands can still contain unresolved placeholders and undeclared references, while concrete namespace bindings should be interpreted separately as portability indicators; and
- a discussion of research opportunities for runtime-aware grounding, structured command generation, and post-generation validation in CI/CD and GitOps-oriented maintenance workflows.

2 Background and Related Work

Large Language Models (LLMs) have become increasingly relevant in software engineering, supporting code generation, test generation, development assistance, and maintenance tasks [3, 5]. However, prior work also reports reliability, security, hallucination, and prompt-sensitivity risks in generated software artifacts [6, 11, 14, 15].

Prompt structure and contextual information strongly influence LLM outputs [13, 15, 17]. This makes metadata grounding attractive for software engineering tasks, but it also raises a question that is especially important in operational domains: which properties context can ground in static artifacts and which properties still require runtime evidence.

Cloud-native and Infrastructure as Code environments amplify this distinction. Kubernetes deployments involve resource relationships, namespaces, selectors, configuration dependencies, security-sensitive resources, and runtime constraints [1, 4, 9, 10]. Continuous delivery and DevOps studies further emphasize that deployment validation must evolve with rapidly changing infrastructure artifacts [2, 8, 12].

Existing work still provides limited evidence on how metadata-grounded LLM generation behaves for Kubernetes validation commands.

IaC-Eval benchmarks the generation of complete cloud IaC programs and evaluates program-level correctness [7], whereas CloudEval-YAML evaluates generated cloud-configuration files with syntax- and configuration-oriented criteria [16]. Our study neither proposes another configuration-generation benchmark nor executes generated infrastructure. It instead uses the individual validation command as the unit of analysis and checks whether command text agrees with resources declared across evolving Kubernetes manifests. The distinguishing contribution is therefore a reusable pre-execution audit of declared references, unresolved placeholders, undeclared references, namespace bindings, and command mix after metadata has already been supplied.

This scope complements program-level IaC and YAML benchmarks: it separates manifest-checkable grounding from runtime-dependent properties and explicitly avoids treating the absence of the selected textual issues as proof that a command will execute or validate the intended behavior.

3 Study Design

Our exploratory investigation examines command text produced after structured deployment metadata is supplied for evolving Kubernetes manifests. The study treats generation as the beginning of an auditable workflow, not as evidence that commands are ready for cluster execution. The workflow comprises four activities: (i) deterministic metadata extraction from paired manifests; (ii) scripted prompt-template filling; (iii) CLI-oriented command generation with Mixtral-8x7B-Instruct-v0.1; and (iv) a rule-based pre-execution audit plus qualitative interpretation.

Figure 1 connects deployment metadata extraction, prompt construction, command generation, and rule-based inspection of selected textual properties before command execution. The generic-prompt branch in the figure is included to clarify the type of under-specified prompt used as a contrastive example; it is not used as a balanced quantitative baseline. Representative examples of prompts and generated artifacts associated with this workflow are presented in Listings 1, 2, and 3.

4 Study Setup

This section describes the target application, deployment evolution scenarios, metadata extraction process, prompt construction strategy, LLM configuration, artifact corpus, and operational indicators used to inspect the generated validation artifacts.

4.1 Target Application

We investigated FinSim, a financial simulation application for which Kubernetes deployment manifests were prepared for this exploratory study. The application source was taken from the `master` snapshot at commit `4d7b10838a982d72e4cec74db7ddc5da98681350`.[1] FinSim models a financial ecosystem with merchants, credit-card processors, and banks and comprises the following microservices:

- transaction-processing component: Flask-based Python service;
- web-interface component: Django-based frontend service;
- backend API component: Django REST Framework service; and
- MySQL database service based on MySQL 8.0.

The application is representative of microservice-oriented systems deployed on Kubernetes, with inter-service communication, database dependencies, API endpoints, and configuration/security adaptations across versions.

4.2 Deployment Evolution Scenarios

The exploratory investigation used 11 Kubernetes deployment versions: one baseline version (`v1`) and ten evolved versions (`v2_1` to `v2_10`). The resulting corpus covers 10 pairwise evolution comparisons, from `v1_vs_v2_1` to `v1_vs_v2_10`. These scenarios include security-related configuration updates, secret handling adaptations,

[1] The cmu-sei/finsim repository is maintained by the Software Engineering Institute at Carnegie Mellon University: https://github.com/cmu-sei/finsim/tree/4d7b10838a982d72e4cec74db7ddc5da98681350.

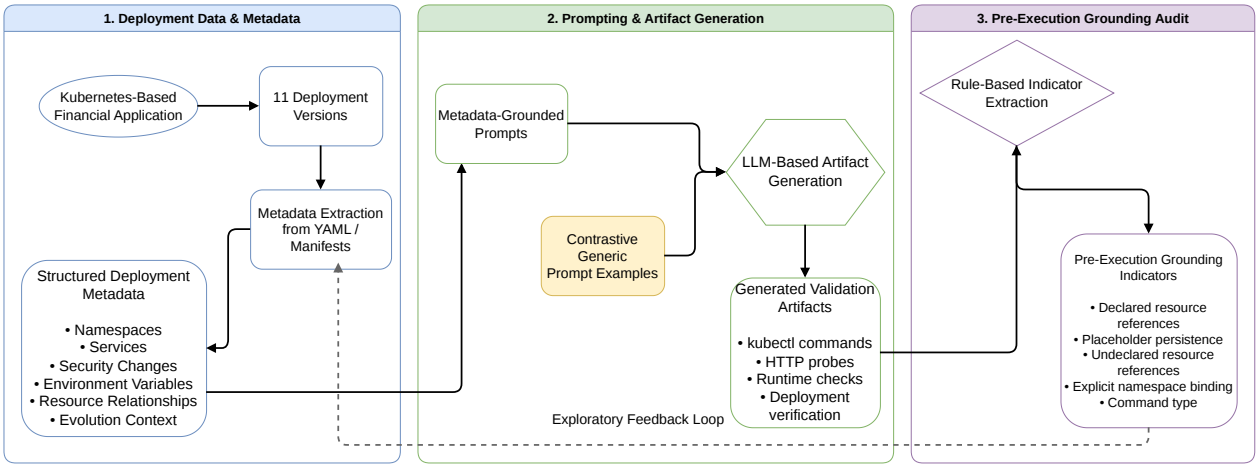

Figure 1: Workflow for metadata-grounded generation and pre-execution audit of Kubernetes validation commands.

Table 1: Validation-command corpus inspected in the exploratory study

Corpus element	Value
Deployment versions	11
Evolution comparisons	10
Validation files	34
CLI command units	178
`kubectl` command tags	136 (76.4%)
`curl`/`echo` command tags	44 (24.7%)
Declared-resource refs.	138 (77.5%)
Placeholder-bearing commands	8 (4.5%)
Undeclared-resource refs.	11 (6.2%)
Explicit `-n default` bindings	143 (80.3%)

token/session handling changes, service and deployment adaptations, and resource or configuration changes.

The comparisons are study units, not equal-complexity treatments. The package links every output to the manifests and metadata used to construct its prompt. Accordingly, the audit characterizes maintenance grounding, not deployment correctness or test effectiveness.

In this study, a generated validation file denotes one LLM output retained for an evolution comparison, whereas a command unit denotes one step-level CLI command extracted from a possibly multi-command procedure. The exploratory generation run was not quota-balanced across comparisons: we retained every available output rather than downsampling to an equal number of files. Consequently, the 2–5 files per comparison are unequal sample sizes, not controlled repetitions. We expose both denominators and use per-comparison rates descriptively; we do not perform inferential comparisons between scenarios.

4.3 Metadata Extraction and Prompt Construction

For each evolution comparison, a deterministic Python extractor loaded the baseline and evolved multi-document YAML manifests and traversed each Kubernetes object. It recorded `apiVersion`, `kind`, `metadata.name`, namespace, labels and selectors, replicas, container images, ports, environment variables, and references to Secrets and ConfigMaps. Missing namespaces were normalized to `default`. The extractor then compared the normalized baseline and evolved records to produce a structured change summary containing added, removed, and modified resources and fields. Extraction and differencing were automated; two researchers audited the resulting records against their source manifests.

A script serialized each change summary into a fixed prompt template with sections for the deployment comparison, namespace, declared resources, modified fields, and the requested Kubernetes/HTTP validation task. Thus, metadata was injected by template filling as textual context, not by manual paraphrasing or a separate retrieval call. The package retains the manifests, extracted records, filled prompts, and scripts so that each prompt field can be traced to its source object. Generic prompts omitted these deployment-specific sections and serve only as qualitative examples of underspecification, not as a quantitative baseline.

Listings 1 and 2 illustrate the contrast between a generic prompt and a metadata-grounded prompt.

```
Generate Kubernetes commands
to validate deployment health.
```

Listing 1: Generic prompt without deployment contextualization

```
Namespace: default
Modified component: finsim-web-v2-4
Change focus: HttpOnly token storage
Deployment comparison: v1_vs_v2_4

Generate Kubernetes and HTTP commands to
inspect deployment consistency and
authentication behavior after the change.
```

Listing 2: Representative metadata-grounded prompt

4.4 LLM Configuration

We used openly accessible Mixtral-8x7B-Instruct-v0.1 through the Hugging Face inference API. This older, single open-weight model supports reproducible or self-hosted instruction following, but is

a case study rather than a proxy for LLMs; newer multi-model replications are needed.

The recorded generation settings are Mixtral-8x7B-Instruct-v0.1, temperature 0.7, a maximum output length of 2,000 tokens, and seeds 1, 2, and 3. The 2,000-token ceiling was a pragmatic budget intended to accommodate multi-step command procedures while bounding response length and inference cost; it was fixed before the audit rather than tuned against indicator outcomes. Because a length ceiling can shape output completeness, we treat it as part of the model configuration and do not generalize beyond it. The package also provides a deterministic mock mode for local smoke checking; this auxiliary facility is not an evaluation of model quality.

4.5 Operational Indicators and Analysis Procedure

To strengthen the auditability of the qualitative analysis, we inspected each generated step-level command using lightweight operational indicators derived from the replication package. The audit is intentionally pre-execution: commands are assessed before manual repair, cluster execution, or post-processing. This choice preserves the exact failure evidence that a maintainer or validation assistant would need to identify before integrating generated commands into a deployment workflow.

RQ1 uses declared-resource references and command-type tags. RQ2 uses placeholder persistence, undeclared-resource references, and explicit namespace binding. RQ3 compares their command-level rates across the ten scenarios in Table 2. This compact mapping replaces a separate table because the indicators are defined immediately below.

The command unit is the level of analysis because generated validation artifacts often mix shell pipelines, `kubectl` inspection steps, and application-level HTTP probes. Treating each step as an inspectable unit makes it possible to identify whether a specific command is grounded in declared resources, still contains unresolved input values, or introduces a resource name that is plausible but absent from the paired manifests. We preserved the original command text before counting; no repair or execution was performed.

The following definitions capture the main pre-execution properties relevant to Kubernetes maintenance and evolution workflows:

- **Declared resource reference**: whether the command references at least one FinSim resource, service, deployment, label, or configuration name declared in the corresponding `v1` and evolved Kubernetes manifests. The denominator is all 178 command units, including application-level probes such as `curl`; therefore, absence of a declared Kubernetes name is not itself classified as an error.
- **Undeclared resource reference**: whether the command references at least one FinSim-like resource name that is not declared in the corresponding manifests, after excluding shell variables and YAML file names.
- **Placeholder persistence**: whether the command contains unresolved placeholders or generic implementation markers, such as `<valid-user>`, `<form_data>`, `your_app`, or `<command_to_execute>`.
- **Explicit default-namespace binding**: whether the command explicitly includes `-n default`. This indicator is not treated as an error by itself because the inspected manifests use the default namespace; instead, it captures how often generated commands encode the observed namespace as a concrete execution binding rather than leaving namespace selection as an environment parameter.
- **Command type**: whether the command is a Kubernetes-oriented command, such as `kubectl`, or an application-level command, such as `curl`. These tags overlap: a shell pipeline may contain both command types. Two mixed units are counted in both categories, which is why the command-type totals in Table 1 sum to 180 rather than 178.

A deterministic indicator-extraction script split the retained files into command units, applied the rules above, and emitted a per-command evidence table with matched manifest names or trigger strings. The numerical counts in Tables 1 and 2 were aggregated from that table. The indicators are not mutually exclusive: a command may refer to a declared deployment while also containing an undeclared selector or unresolved placeholder. Counts are therefore indicator occurrences, not a partition of 178 commands. Two researchers audited the rules and every automatically flagged case, then selected the examples in Table 3 and Listings 3–4. The package includes the script, its inputs, and the per-command evidence table.

No Kubernetes cluster or `kubectl` binary was invoked in this pre-execution study, so Kubernetes and `kubectl` runtime versions are not experimental factors. This is also why the audit cannot establish syntax acceptance, permissions, cluster-state compatibility, or validation effectiveness.

Generic-prompt materials are qualitative underspecification examples only; all quantitative claims concern the metadata-grounded corpus and do not establish comparative superiority.

5 Empirical Observations

The exploratory analysis identified both manifest-focused commands and a small, unevenly distributed set of selected pre-execution issues. This section reports descriptive indicators for 34 generated validation files and 178 step-level CLI command units, organized around the three research questions.

5.1 Observation 1: Declared Resource References and Command Mix

Across all 178 command units, 138 (77.5%) referenced at least one FinSim resource, label, service, deployment, or configuration name declared in the paired `v1` and evolved manifests. The remaining 40 commands are not automatically failures: the denominator includes application-level HTTP probes that may legitimately name an endpoint rather than a Kubernetes resource. In addition, 136 commands (76.4%) used `kubectl`; command type tags overlap when a pipeline combines Kubernetes and application-level operations.

Regarding RQ1, the result means that 138 command units contained a verifiable lexical link to a resource declared in the corresponding manifest pair. It does not show that the other commands are wrong, that the referenced resources are used semantically correctly, or that metadata improved the rate relative to generic prompting. Without a balanced baseline, 77.5% is a corpus description rather than a good/poor threshold or an attributable treatment effect.

5.2 Observation 2: Concrete Namespace Binding

For RQ2, 143 commands (80.3%) explicitly included `-n default`. This binding is correct for the inspected manifests, and some prompts explicitly supplied the namespace; it is therefore not a failure count. The indicator records how often a concrete environment binding appears in command text so that maintainers can distinguish a manifest-consistent command from one that is portable without editing.

This distinction matters for Kubernetes maintenance: entity focus can be checked against manifests, whereas portability depends on the target environment. A concrete namespace may be intentional and correct; whether it becomes problematic can be decided only for a deployment context.

5.3 Observation 3: Selected Issues Are Sparse and Unevenly Distributed

Regarding RQ2, eight commands (4.5%) contained unresolved placeholders or generic implementation markers, including placeholder form payloads, credentials, `your_app`, and `<command_to_execute>`. These units require value resolution to become concrete command text. The indicator does not establish that command units without these markers are executable.

We also identified 11 commands (6.2%) containing FinSim-like resource references not declared in the corresponding paired manifests. For example, commands in the `v1_vs_v2_4` comparison referred to `finsim-web-v2.4`, while the manifest declares `finsim-web-v2-4`.

Other cases referenced Secrets or NetworkPolicies absent from the inspected manifests. For RQ3, the distribution is highly concentrated: `v1_vs_v2_4` accounts for 9 of the 11 undeclared-reference occurrences in Table 2, and most other comparisons have none. We therefore interpret the dotted-versus-hyphenated suffix as a localized naming issue in this corpus, not as evidence of a recurrent rate across evolution scenarios.

5.4 Observation 4: Implications for Post-Generation Validation

Regarding RQ2 and RQ3, manifest grounding can coexist with unresolved operational issues, so generated commands remain candidates for post-generation validation. Checks can verify referenced resources, target namespaces, remaining placeholders, and conformance to structured Kubernetes command schemas. Listings 3 and 4 contrast a grounded pattern with unresolved input and an undeclared resource spelling.

```
kubectl get deployment finsim-web-v2-1 -n default
kubectl get pods -l app=finsim-web-v2-1 \
  -n default -o jsonpath='{.items[0].metadata.name}'
```

Listing 3: Representative deployment-grounded artifact

```
echo "<form_data>" | curl -X POST http://localhost:8080/
    signup/
kubectl get pods -l app=finsim-web-v2.4 -n default
```

Listing 4: Representative unresolved and undeclared references

6 Discussion

Manifest references and selected textual issues coexist in Mixtral-generated validation commands. Because no command was executed, the indicators do not establish executability, fault detection, or behavioral validity. Static review can compare names, namespaces, selectors, and placeholders with manifests; permissions, cluster state, endpoints, syntax, and semantic intent require additional evidence.

Most commands named declared FinSim resources, while a few retained placeholders or plausible undeclared names, which is consistent with supplied metadata being combined with generalized Kubernetes patterns. Explicit `-n default` is different: it is correct for this corpus and exposes environment coupling rather than an error. Likewise, absence of the measured indicators would establish only that those selected issues were not observed.

Three testable follow-up directions arise:

(1) evaluate whether schema-constrained generation reduces unresolved fields without altering the intended command sequence;
(2) compare manifest-only grounding with live-cluster retrieval for namespaces, resources, permissions, and endpoints; and
(3) measure whether validators and grounding-status displays improve human review and downstream command success.

For CI/CD and GitOps workflows, validation assistants could visualize which commands are manifest-grounded, runtime-dependent, or affected by unresolved or undeclared references.

A practical workflow would reject unresolved placeholders, compare names and namespaces with versioned manifests, and flag permission-, endpoint-, or state-dependent commands for staging validation. Retaining each matched object or triggering rule would keep the gate auditable and distinguish deliberate external dependencies from generation errors.

Manifest-checkable indicators can report their supporting object or field at pull-request time; runtime-dependent checks can be deferred to a controlled cluster without mistaking static evidence for operational validation.

These interpretations are configuration-specific: the corpus uses one older model, unequal comparison sizes, and no controlled generic-prompt baseline. The study offers hypotheses and audit rules, not an effect estimate; generated commands remain candidates whose readiness requires structural validation, manifest checks, and, where safe, controlled execution.

7 Limitations

This exploratory study characterizes metadata grounding under the following threats to descriptive validity.

Construct validity. These textual pre-execution indicators do not measure executability, validation effectiveness, fault detection, coverage, or live-cluster behavior. Commands may still fail because of syntax, selectors, permissions, endpoints, state, output assumptions, or semantic mismatch. Conclusions are limited to the defined measures and denominators.

An indicator-free command may still be invalid or require unavailable permissions and state; the analysis supports pre-execution triage, not certification.

Table 2: Per-comparison indicator counts and command-level rates

Comparison	Files	Cmds.	Declared n (%)	Placeholder n (%)	Undeclared n (%)	`-n default` n (%)
v1_vs_v2_1	3	18	12 (66.7)	2 (11.1)	0 (0.0)	12 (66.7)
v1_vs_v2_2	5	26	19 (73.1)	0 (0.0)	0 (0.0)	20 (76.9)
v1_vs_v2_3	2	14	12 (85.7)	4 (28.6)	0 (0.0)	12 (85.7)
v1_vs_v2_4	3	16	16 (100.0)	0 (0.0)	9 (56.3)	16 (100.0)
v1_vs_v2_5	3	15	12 (80.0)	1 (6.7)	0 (0.0)	12 (80.0)
v1_vs_v2_6	3	12	8 (66.7)	0 (0.0)	0 (0.0)	8 (66.7)
v1_vs_v2_7	3	14	11 (78.6)	0 (0.0)	0 (0.0)	11 (78.6)
v1_vs_v2_8	5	26	20 (76.9)	0 (0.0)	1 (3.8)	21 (80.8)
v1_vs_v2_9	3	17	16 (94.1)	1 (5.9)	0 (0.0)	15 (88.2)
v1_vs_v2_10	4	20	12 (60.0)	0 (0.0)	1 (5.0)	16 (80.0)
Total	**34**	**178**	**138 (77.5)**	**8 (4.5)**	**11 (6.2)**	**143 (80.3)**

Table 3: Representative textual properties observed in generated commands

Indicator	Representative generated command	Interpretation
Declared resource grounding	`kubectl get deployment finsim-web-v2-1 -n default`	Command refers to a deployment declared in the inspected manifests.
Placeholder persistence	`curl ... "<valid-user>"`	Command sketches an interaction but still requires value resolution.
Undeclared resource reference	`kubectl get pods -l app=finsim-web-v2.4 -n default`	Command uses a plausible but undeclared component identifier.
Explicit default-namespace binding	`kubectl get pods ... -n default`	Command encodes the observed namespace as a concrete execution binding.
Application-level validation	`curl -X GET http://localhost:8080/login`	Artifact mixes infrastructure inspection with runtime HTTP checks.

Internal validity. Two researchers audited rules, counts, and examples after extraction. Inter-rater agreement is not reported because indicators follow explicit rules rather than ordinal coding.

External validity. One application, scenario family, and older Mixtral-8x7B-Instruct-v0.1 configuration may not represent other applications, platforms, prompts, output limits, or models [3, 5, 15]. Illustrative, quota-imbalanced generic prompts preclude attributing rates to metadata grounding or generalizing them to LLMs.

Reliability and reproducibility. The package contains manifests, metadata, prompts, outputs, settings, deterministic extraction scripts, and per-command evidence; Section 4 pins the source snapshot. Kubernetes and `kubectl` versions are omitted because neither ran. Multi-model replications should report execution environments and compare generic, grounded, retrieval-, schema-, and runtime-aware prompting.

8 Conclusion

Across ten FinSim comparisons, 34 Mixtral-8x7B-Instruct-v0.1 validation files yielded 178 command units. Few unresolved or undeclared references appeared, and explicit `-n default` bindings were correct. Together, these observations show that metadata can improve lexical grounding while still leaving operational validity unresolved. Without execution or balanced baselines, the findings do not establish executability, validation effectiveness, causal benefit, or general LLM behavior.

The contribution is an auditable distinction between manifest-checkable and runtime-dependent evidence, not a claim of executable generation. By pairing explicit indicators with per-command evidence, the study turns this distinction into a repeatable pre-execution review that can show where manifests suffice and where sandbox or live-cluster validation remains necessary.

Future work should compare newer models under balanced prompts, combine schema constraints with live-cluster retrieval, and replicate across applications and clusters. Evaluations should measure reviewer effort, false positives, portability, failure localization, and command success to identify dependable combinations of generation and validation.

Overall, metadata grounding exposes assumptions and narrows the search space; provenance, structural validation, and controlled execution turn plausible commands into reviewable maintenance artifacts.

Artifact Availability

Available from the corresponding author upon request, the replication package contains manifests, metadata, prompts, outputs, settings, deterministic scripts, and per-command evidence[2].

Acknowledgments

The authors gratefully acknowledge the financial support provided by the Fundação de Amparo à Pesquisa do Estado de Minas Gerais (FAPEMIG), Brazil, through projects CIN II APQ-06513-25 and CEX-BIS-00192-25.

[2] https://github.com/californi/MetadataIsNotEnough/

References

[1] David Bernstein. 2014. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing* 1, 3 (2014), 81–84. doi:10.1109/MCC.2014.51

[2] Lianping Chen. 2015. Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software* 32, 2 (2015), 50–54. doi:10.1109/MS.2015.27

[3] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M. Zhang. 2023. Large Language Models for Software Engineering: Survey and Open Problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, Melbourne, Australia, 31–53. doi:10.1109/ICSE-FoSE59343.2023.00008

[4] Michele Guerriero, Martin Garriga, Damian A. Tamburri, and Fabio Palomba. 2019. Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Cleveland, OH, USA, 580–589. doi:10.1109/ICSME.2019.00092

[5] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. *ACM Transactions on Software Engineering and Methodology* 33, 8, Article 220 (2024), 79 pages. doi:10.1145/3695988

[6] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiannan Yu, Dan Su, Yan Xu, Christy Dougherty, Justin S. Y. Fu, and Maosong Sun. 2023. Survey of Hallucination in Natural Language Generation. *Comput. Surveys* 55, 12 (2023), 1–38. doi:10.1145/3571730

[7] Patrick Tser Jern Kon, Jiachen Liu, Yiming Qiu, Weijun Fan, Ting He, Lei Lin, Haoran Zhang, Owen M. Park, George Sajan Elengikal, Yuxin Kang, Ang Chen, Mosharaf Chowdhury, Myungjin Lee, and Xinyu Wang. 2024. IaC-Eval: A Code Generation Benchmark for Cloud Infrastructure-as-Code Programs. In *Advances in Neural Information Processing Systems 37: Annual Conference on Neural Information Processing Systems 2024, Datasets and Benchmarks Track*. Curran Associates, Inc., Vancouver, Canada, 134488–134506. https://openreview.net/forum?id=7TCK0aBL1C

[8] Marko Leppanen, Simo Makinen, Max Pagels, Vesa-Pekka Eloranta, Juha Itkonen, Mika Mantyla, and Tomi Mannisto. 2015. The Highways and Country Roads to Continuous Deployment. *IEEE Software* 32, 2 (2015), 64–72. doi:10.1109/MS.2015.50

[9] Kief Morris. 2016. *Infrastructure as Code: Managing Servers in the Cloud*. O'Reilly Media, Inc., Sebastopol, CA. https://www.oreilly.com/library/view/infrastructure-as-code/9781491924334/

[10] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. 2017. Cloud Container Technologies: A State-of-the-Art Review. *IEEE Transactions on Cloud Computing* 7, 3 (2017), 677–692. doi:10.1109/TCC.2017.2702586

[11] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2022. Asleep at the Keyboard? Assessing the Security of GitHub Copilot's Code Contributions. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, San Francisco, CA, 754–768. doi:10.1109/SP46214.2022.9833571

[12] Akond Rahman and Laurie Williams. 2018. Characterizing Defective Configuration Scripts Used for Continuous Deployment. In *Proceedings of the 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Västerås, Sweden, 34–45. doi:10.1109/ICST.2018.00014

[13] Laria Reynolds and Kyle McDonell. 2021. Prompt Programming for Large Language Models: Beyond the Few-Shot Paradigm. arXiv:2102.07350 [cs.CL] https://arxiv.org/abs/2102.07350

[14] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software Testing with Large Language Models: Survey, Landscape, and Vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936. doi:10.1109/TSE.2024.3368208

[15] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashis Ghosh, Andrew C. Cates, Sydney Heaton, Eric Gombolay, Jeff Jenkins, and Douglas C. Schmidt. 2023. A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT. arXiv:2302.11382 [cs.SE] https://arxiv.org/abs/2302.11382

[16] Yifei Xu, Yuning Chen, Xumiao Zhang, Xianshang Lin, Pan Hu, Yunfei Ma, Songwu Lu, Wan Du, Zhuoqing Mao, Ennan Zhai, and Dennis Cai. 2024. CloudEval-YAML: A Practical Benchmark for Cloud Configuration Generation. arXiv:2401.06786 [cs.SE] https://arxiv.org/abs/2401.06786

[17] J. D. Zamfirescu-Pereira, Richmond Wong, Benedetta Piantella, and Qian Yang. 2023. Why Johnny Can't Prompt: How Non-AI Experts Try (and Fail) to Design LLM Prompts. *Proceedings of the ACM on Human-Computer Interaction* 7, CSCW1, Article 123 (2023), 32 pages. doi:10.1145/3544548.3581388