IaC Eval Bench: Auditing Static Evaluators for LLM-Generated Kubernetes Configurations

Deavyansh Gautam H Zhang
Independent engineering research

https://github.com/deavgprime/iac-eval-bench

Abstract

Evaluating generated infrastructure configuration requires more than checking that an output parses. A configuration can satisfy a schema while violating a task requirement; conversely, a correct configuration can fail an overly restrictive grader. This report presents IaC Eval Bench, a reproducible static evaluation harness for a constrained Kubernetes configuration-generation interface. The suite contains six original synthetic tasks and 50 deterministic requirement checks, with constructed positive and negative controls. An internal scoring audit identified five grading errors and one resume-provenance defect. The grading errors included both false acceptance of contradictory configurations and false rejection of compatible additions. We describe the corrected scoring contract, evidence-preserving execution protocol and failure-analysis procedure. A recorded CPU-only experiment evaluated Qwen2.5-Coder 1.5B and 0.5B, each once on all six tasks. Neither model passed a complete task; their macro rubric scores were 17.13% and 0%, respectively. The smaller model's six responses were valid JSON but failed the required response envelope. Replaying the saved outputs reproduced every score. These observations support an engineering contribution in evaluator auditing and experiment integrity, not a model ranking or a claim of general infrastructure competence. The artifacts expose the prompts, outputs, rubrics, diagnostic edits and operational measurements needed to inspect that distinction.

Keywords: language model evaluation; Kubernetes; infrastructure as code; scoring correctness; reproducibility; regression testing.

Contents

1. Introduction **3**

2. Related work and positioning **3**

3. Scope, artifacts and task design **3**

4. Evaluation method **4**

4.1 A staged scoring contract 4

4.2 Semantic checks and deliberate restrictions 5

4.3 Controls, mutations and diagnostic edits 5

4.4 Experiment identity and failure retention 5

5. Evaluator audit **6**

6. Recorded model experiment **6**

6.1 Protocol and execution conditions 6

6.2 Results 7

6.3 Failure interpretation 8

7. Reproduction and artifact validation **8**

8. Limitations and next experiments **8**

9. Conclusion **9**

References **9**

1. Introduction

A model evaluation has at least two failure sources: the system being evaluated and the software that decides whether it succeeded. Infrastructure configuration makes this separation especially important. Requirements are distributed across resource objects, defaults, selectors, names and cross-resource relationships. A Deployment can have a valid schema but select no pods. A Service can name a container port whose transport differs from its own. A test that verifies only the presence of expected keys can accept both mistakes.

The reverse problem is equally consequential. Comparing whole objects against a reference rejects harmless additions and compatible alternatives. An extra metadata label need not invalidate the required selector. If a grader treats it as a failure, the resulting score measures an accidental implementation restriction rather than the stated task.

This report studies a deliberately small evaluation environment in which these distinctions can be inspected. It asks three questions. RQ1: can positive and targeted negative fixtures be supplemented with adversarial cases that reveal grading defects? RQ2: can complete-task results, partial scores and stage-specific failures explain different behaviors hidden behind identical pass counts? RQ3: what evidence and identity checks are needed to resume an experiment without silently changing its meaning?

The contribution is an implemented harness and an auditable case study: a bounded six-task contract, a documented five-case scoring audit, evidence-preserving execution and analysis of 12 model attempts. This is not a new model architecture, a large benchmark, or an estimate of deployment readiness. The distinction is important because a narrowly measured engineering result can still be useful without supporting a broad capability claim.

2. Related work and positioning

HELM argues for evaluating several dimensions of model behavior and retaining raw prompts and completions [1]. This report adopts that measurement perspective at a much smaller scale: complete-task success, partial requirements, failure stages, usage and execution conditions are reported separately. No comparison with HELM scores is intended.

HumanEval assesses generated programs through functional tests and studies repeated sampling [2]. IaC Eval Bench instead checks declarative configurations without executing them. Its one-attempt experiment does not estimate pass@k, sampling reliability or functional behavior inside a cluster. Qwen2.5-Coder provides the model family used for the recorded pilot [3]; the present experiment does not reproduce the model authors' benchmarks.

Datasheets for Datasets motivates documenting how examples were made and what they can support [4]. Here, constructed fixtures are explicitly separated from generated model responses. Kubernetes documentation defines the selector and network-policy semantics relevant to several checks [5, 6]. The report contributes a concrete evaluator audit against those semantics, not a claim that existing infrastructure tooling lacks schema validation.

3. Scope, artifacts and task design

The artifact inspected for this report is repository revision `58d376038b5d5ad4b34e3616605c79cf2d20d10f` [7]. The recorded model experiment used clean source revision `3da83dbb58855804b49b0f2c7b1d995c587ddbdd`, harness 0.2.0 and suite `kubernetes-config-v2` version 2.0.0 [8]. The audit's original revision is identified separately in its evidence [9]. These identifiers must not be treated as interchangeable: the current repository includes documentation and saved results produced after inference.

A response must be a single JSON object with a `resources` array. The target is Kubernetes 1.34.0, with pinned, bundled strict schemas. The suite does not evaluate Dockerfiles, Terraform programs, arbitrary YAML, cloud provisioning or interactive tool use. Outputs are treated as data

and are never applied to a cluster.

Table 1. Task scope and failure boundaries.

Task	Required behavior	Important failure boundary
deployment-resources	Replicas, resource bounds, labels, named port and HTTP readiness	A selector can contradict the submitted pod labels
service-routing	Deployment and Service inventory with aligned names and ports	A named port does not establish transport agreement
network-isolation	Required ingress and constrained DNS egress	Namespace and pod restrictions must preserve conjunction
cronjob-bounds	Schedule, concurrency, retries and execution bounds	Correct kind with the wrong required name still fails inventory
workload-hardening	Declared security settings and disruption budget	Selected field checks do not prove runtime security
configmap-wiring	Immutable configuration and workload references	Resource existence alone does not establish correct wiring

The six tasks and 50 checks are original public synthetic material. Repository metadata records AI assistance in task authorship. The fixtures are constructed examples, not measurements of a model, and no private client dataset is used. The same author maintained tasks, fixtures and grader; this simplifies auditability but creates a shared-error risk. An external reviewer and held-out tasks would be needed to reduce that risk.

4. Evaluation method

4.1 A staged scoring contract

The evaluator first enforces strict JSON, rejecting duplicate keys, non-finite numbers, invalid Unicode, excessive nesting and outputs over 1 MiB. It then checks allowlisted resource kinds and schema validity, followed by exact API-version, kind, name and namespace inventory. Extra or duplicate resource objects fail the inventory gate. Only responses that pass these stages receive requirement credit.

For a completed response on task t, let G_t be one if its parse, schema and inventory gates pass, and zero otherwise. Let K_t be the number of rubric checks and b_tk their binary results. The response score is:

```
score_t = G_t * (sum_k b_tk / K_t)
task_pass_t = 1 if G_t = 1 and every b_tk = 1; otherwise 0
```

The macro score averages per-task mean scores, giving tasks equal weight rather than weighting them by their number of checks. Provider failures have no conditional quality score. The operational score instead includes failed provider attempts as zero and reports coverage. This prevents an incomplete run from appearing complete simply because its unsuccessful requests disappeared from the denominator.

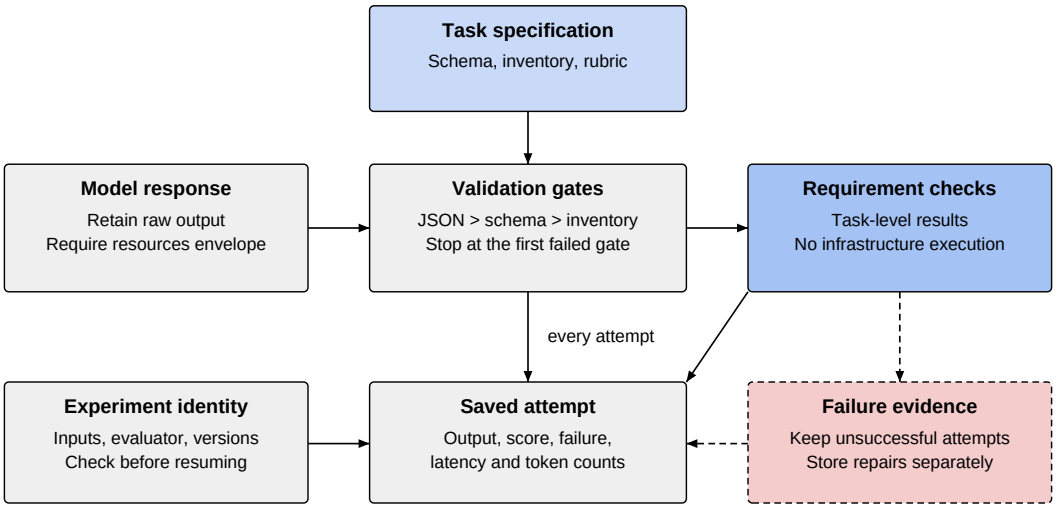


Figure 1. Static evaluation and retained evidence. Validation gates precede task requirements. Experiment identity is checked before resuming; every attempt is retained. No resources are deployed.

4.2 Semantic checks and deliberate restrictions

Selectors are evaluated against submitted pod labels, including all declared `matchExpressions`. Matching requires the required app label and satisfaction of every selector constraint. Network-policy array order is ignored where the contract allows it, but element multiplicity remains significant; duplicates cannot replace missing required peers. TCP defaults are applied where implemented. A readiness probe must use the requested HTTP action without a conflicting second handler.

This is a restricted contract, not a Kubernetes equivalence solver. Exact resource names and some quantity strings are intentional. Schema acceptance is not API-server admission, and a valid alternative outside the prompt's constrained representation may fail. That boundary is documented so that an operator can distinguish an intentional task restriction from an accidental grader defect.

4.3 Controls, mutations and diagnostic edits

Each task has a constructed passing reference and a schema-valid negative fixture targeting a named requirement. The regression suite also mutates required or forbidden fields across all 50 checks. Broader adversarial cases test compatibility, contradictions, defaults, duplicate elements and transport behavior.

A diagnostic repair is a reviewer intervention on a saved response, not a second model attempt. Its purpose is to determine whether the claimed defect explains the failure. Diagnostic repairs are saved separately and never replace raw outputs or improve the reported model score. This preserves the distinction between what the model produced and what an engineer can repair afterward.

4.4 Experiment identity and failure retention

A run records prompts, inputs, schemas, evaluator content, configuration, validation dependencies and model metadata. Fingerprints bind resume behavior to this identity. Resume validates the manifest fields as well as the fingerprint, preventing an edited display field from silently changing a report's model attribution. Attempts are written atomically, checksummed and protected by a directory lock.

Checksums detect changes; they do not authenticate the author or prevent a privileged user from rewriting both evidence and hashes. Similarly, a supplied model revision is recorded rather than independently verified by the runner. Reproduction requires checking the actual model digest, not trusting a mutable tag. Provider failures remain part of the run, and retries require a separate

experiment rather than hidden replacement.

5. Evaluator audit

The audit found three false acceptances and two false rejections in the original scoring behavior [9]. Both the original six positive fixtures and six targeted negative fixtures had their expected outcomes despite these defects. Passing a small fixture suite was therefore insufficient evidence that the evaluator implemented its intended semantics.

Table 2. Historical scoring defects and corrected verdicts.

Audit case	Earlier verdict	Corrected verdict	Mechanism
Add app NotIn [api] to a selector for pods labeled app=api	Pass	Fail	Extra selector constraints were ignored
Add unrelated team=platform to pod labels	Fail	Pass	Whole-object equality rejected a harmless addition
Add compatible app Exists to the PDB selector	Fail	Pass	Compatible selector expressions were rejected
Add a TCP handler beside HTTP readiness	Pass	Fail	Required keys were checked, conflicting handlers were not
Use UDP for the workload port and TCP for the Service	Pass	Fail	Named-port checks omitted protocol agreement

The resume audit found a sixth, different issue: recorded model configuration could be changed without updating the fingerprint, allowing misleading attribution on resume. The corrected implementation checks the stored provenance fields against current inputs. This is an integrity defect rather than a sixth scoring defect.

Not every later improvement is counted as a discovered bug. Order invariance in supported NetworkPolicy arrays was a clarified contract improvement. Treating a desirable expansion as an original correctness failure would overstate the audit. Regression evidence and documentation retain that distinction.

An illustrative selector example makes the issue concrete. If a pod carries `app=api`, then requiring both `app=api` and `app NotIn [api]` selects no such pod. A grader checking only the first conjunct accepts the configuration incorrectly. In contrast, adding `team=platform` to the pod does not invalidate `app=api`. The evaluator must reason about relevant constraints rather than treating every addition as either harmless or forbidden.

6. Recorded model experiment

6.1 Protocol and execution conditions

The experiment on 18 October 2026 ran Qwen2.5-Coder 1.5B before 0.5B, both quantized as Q4_K_M, with one attempt for each task and no retries. Both used JSON mode, seed 42, temperature 0, context 4096, output cap 1024, two model threads and a 600-second request timeout. All 12 requests ended normally with `stop`; none was reported as output-cap truncation.

Ollama 0.40.6 ran on an x86_64 host with four logical CPUs, approximately 31.3 GiB RAM, no GPU and no swap. The workload used a 6 GiB memory maximum, 4 GiB high-memory threshold and 150% CPU quota, with one model and one request active at a time. A watchdog monitored available memory, disk and host load. The earlier single-task pilot is retained separately and excluded from this comparison.

The fixed model order, cold-loading effects and brief background dependency installation are potential confounders. Background work overlapped two 0.5B attempts before being frozen. The measurements are observations of these executions, not an isolated performance benchmark.

6.2 Results

Table 3. Recorded model totals.

Model	Task passes	Macro rubric score	Input tokens	Output tokens	Mean request time
Qwen2.5-Coder 1.5B	0/6	17.13%	1,536	1,661	43.57 s
Qwen2.5-Coder 0.5B	0/6	0.00%	1,536	1,472	19.47 s

The 1.5B model received 77.78% on deployment-resources and 25.00% on network-isolation. The remaining four attempts failed schema or inventory gates. All six 0.5B outputs were syntactically valid JSON but lacked the required response envelope and therefore failed the evaluator's parse/interface stage. A zero score here must not be paraphrased as an inability to generate JSON.

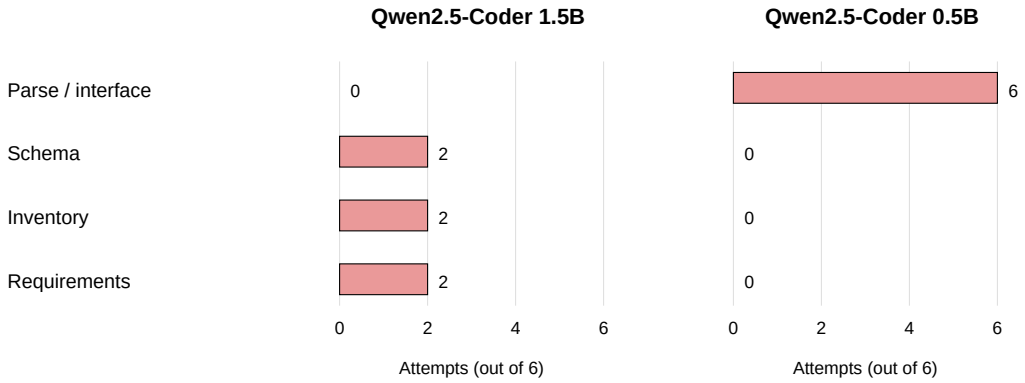


Figure 2. First failed stage in each saved attempt. All six 0.5B responses are valid JSON but violate the required response envelope. Both models have zero complete-task passes. Counts are not estimates of failure probability.

Table 4. Per-task failures, scores and request latency.

Task	1.5B stage; score	0.5B stage; score	1.5B / 0.5B request seconds
configmap-wiring	Schema; 0%	Parse/interface; 0%	44.08 / 16.67
cronjob-bounds	Inventory; 0%	Parse/interface; 0%	33.03 / 15.44
deployment-resources	Requirements; 77.78%	Parse/interface; 0%	46.22 / 22.48
network-isolation	Requirements; 25.00%	Parse/interface; 0%	41.92 / 28.47
service-routing	Inventory; 0%	Parse/interface; 0%	44.36 / 15.27
workload-hardening	Schema; 0%	Parse/interface; 0%	51.82 / 18.48

Peak sampled cgroup memory was approximately 2.52 GiB for the larger model's attempts and 1.89 GiB for the smaller model's attempts. These figures include workload page cache and runner overhead, not only model resident memory. CPU usage was sampled every two seconds within attempt windows and can undercount interval edges. Token counts came from Ollama; monetary cost and electricity usage were not measured.

6.3 Failure interpretation

For the 1.5B deployment task, the output supplied a liveness probe instead of the requested readiness probe and omitted the required named port. For the network task, it omitted `podSelector` and confused namespace and pod restrictions. Other failures included incorrect resource names and misplaced security fields. These are inspectable output defects against the recorded prompt, rather than an inference from the final score alone.

Targeted reviewer repairs made all six 1.5B responses pass. Wrapping the six 0.5B responses in the required envelope did not make any pass: they still contained missing resources, wrong names or incorrect structures. Thus format compliance contributed to the smaller model's zero scores, but repairing format alone did not remove its substantive errors. No repaired output counts as model success.

The manual review identified no blocking ambiguity or false rejection in these 12 attempts. That finding is local to these cases. It does not certify all prompts, all possible correct outputs or the complete grader.

7. Reproduction and artifact validation

For this report, the current source snapshot was installed in an isolated Python environment. All 51 unittest methods passed. Fixture replay produced six passing references and detected all six targeted negative controls. Offline review reproduced the 12 saved model scores; the six larger-model diagnostic repairs passed and all six smaller-model wrapper-only repairs still failed. These checks used saved outputs, not fresh model inference. The test log also records a schema-dialect fallback warning: the bundled schemas declare `http://json-schema.org/schema#`, which the installed validator does not recognize as a registered metaschema. The present run falls back to its latest supported draft. Passing tests do not remove this compatibility risk; a future revision should explicitly resolve and test the intended dialect. A socket resource warning also appears in a provider-error test. Both warnings are retained in the validation log.

The reproducible entry points are `python -m unittest discover -s tests -v`, `python scripts/verify_baselines.py --output runs/audit-reproduction`, and `python results/2026-10-18-qwen-comparison/review.py`. The last command writes a derived failure-review file, so it should run in a disposable checkout or after preserving the original. The accompanying report package records the environment and validation logs. A fresh model experiment additionally needs the saved model digests, Ollama version, protocol and resource limits [8].

Source changes, dependency changes and model-tag changes can alter results. A reproduced score on a stored output is a deterministic evaluator check; a fresh generation is a new experiment. Both are useful, but they answer different questions.

8. Limitations and next experiments

The suite has six public synthetic tasks, one model family and one attempt per task. There is no hidden test set, independent task authoring or estimate of within-model variability. The experiment supports neither a general model ranking nor a stable tail-latency estimate. Public fixtures can also appear in training or retrieval material.

The static contract omits API admission, runtime scheduling, image execution, network enforcement and application health. It is not a security scanner. Some valid representations are intentionally outside scope. A wider suite must distinguish these exclusions from faults in the grader and should publish counterexamples for accepted and rejected equivalents.

The next empirical study should independently review a held-out task set; repeat attempts under randomized model order; isolate warm and cold timings; compare multiple model families; and

add disposable-cluster admission checks as a separately labeled stage. An ablation should compare schema-only scoring, the original semantic checks and the corrected checks on the same mutation corpus. This would quantify the marginal value of each layer instead of relying only on five selected examples. These experiments are proposed, not completed.

9. Conclusion

The central finding is that correct fixture outcomes can coexist with an incorrect evaluator. The audit exposed both false acceptance and false rejection, while the model comparison showed how identical complete-task pass counts can conceal distinct failure mechanisms. IaC Eval Bench provides a compact artifact for studying those problems with explicit contracts, retained attempts and reproducible diagnostic checks. Its contribution is the inspectability of the measurement process, within a clearly bounded static Kubernetes task suite.

References

[1] Liang, P., et al. Holistic Evaluation of Language Models. 2022/2023. https://arxiv.org/abs/2211.09110

[2] Chen, M., et al. Evaluating Large Language Models Trained on Code. 2021. https://arxiv.org/abs/2107.03374

[3] Hui, B., et al. Qwen2.5-Coder Technical Report. 2024. https://arxiv.org/abs/2409.12186

[4] Gebru, T., et al. Datasheets for Datasets. 2018, revised 2021. https://arxiv.org/abs/1803.09010

[5] Kubernetes Authors. Labels and Selectors. https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/

[6] Kubernetes Authors. Network Policies. https://kubernetes.io/docs/concepts/services-networking/network-policies/

[7] Gautam, D. IaC Eval Bench, source snapshot. https://github.com/deavgprime/iac-eval-bench/tree/58d376038b5d5ad4b34e3616605c79cf2d20d10f

[8] Gautam, D. Qwen2.5-Coder: six-task CPU comparison, protocol and raw artifacts. https://github.com/deavgprime/iac-eval-bench/tree/58d376038b5d5ad4b34e3616605c79cf2d20d10f/results/2026-10-18-qwen-comparison

[9] Gautam, D. Scoring audit: suite v2, observations and regression evidence. https://github.com/deavgprime/iac-eval-bench/blob/58d376038b5d5ad4b34e3616605c79cf2d20d10f/docs/scoring-audit.md