

KARMA: A Composable Framework for Evaluating LLM Agents on Stateful Kubernetes Operations

Junhan Ouyang*

Independent Researcher
junhano@uci.edu

Jinlang Wang

University of Wisconsin–Madison
jwang2775@wisc.edu

Shaw Liu*

University of Wisconsin–Madison
xliu2528@wisc.edu

Jiawei Zhou

Stony Brook University
jiawei.zhou.1@stonybrook.edu

Abstract

Large language model (LLM) agents are increasingly used for infrastructure operations on Kubernetes, yet how to test them efficiently and accurately remains unresolved. Realistic long-horizon evaluations are time-consuming and sometimes error-prone to build, since test cases are usually written from scratch even though components such as setup, state management, and verification logic are duplicated across them. We present KARMA,¹ an open-source framework that composes reusable, parameterized operational *cases* into stateful, multi-stage Kubernetes *workflows*. Each case textually specifies a task and a required cluster state, and contains a deterministic *oracle*. The cluster state is preserved across stages, so long-horizon tasks can be constructed with little additional effort. KARMA also provides controls that add further complexity, namely alternating prompt modes and adversary injection. For cross-stage testing, a *regression sweep* reruns all previous cases and an LLM judge adjudicates the results, detecting execution inconsistencies or conflicts introduced by agents. Alongside the framework, we release a corpus of 89 cases, 38 adversaries, and 327 workflows across eight systems.

1 Introduction

LLM agents are moving from writing code to operating production infrastructure, including diagnosing incidents, deploying upgrades, rotating credentials, and correcting configuration drift (Zhang et al., 2025; Lin et al., 2025; Yang et al., 2026). Evaluating these capabilities on real systems is critical for judging how reliably such agents would behave once deployed (Wang et al., 2025; Luo et al., 2025; Tian et al., 2025; Zhou, 2026), and how much human supervision they require. These operations are complex, but most of them can be broken into

modular stages, and different tasks share considerable boilerplate work. They are also fundamentally sequential, since each subtask builds on the result of the previous one and must execute without breaking the state it depends on.

Recent benchmarks have introduced more complex, long-horizon scenarios (Merrill et al., 2026; Xu et al., 2025; Guo et al., 2026). However, these scenarios are typically built as standalone tasks, each requiring separate setup and verification logic. This limits reuse and makes it harder to expand coverage or increase task difficulty.

We introduce KARMA, Kubernetes Agent Runtime Measurement Architecture, which addresses this limitation with reusable, parameterized test cases. Each *case* is a self-contained test specifying a natural-language task, its required cluster state, and a deterministic *oracle*. A *Probe*, *Apply*, and *Verify* cycle brings the cluster to that required state, applying it only when the probe finds it missing, so the same case runs correctly whether invoked alone or after an earlier stage has already produced that state. Cases can therefore be combined into multi-stage *workflows* over preserved, continuous cluster state, so new evaluations can be built by recombining cases rather than writing each scenario from scratch (§3.1).

To emulate a realistic, error-prone execution environment and surface a wider range of agent behavior, KARMA lets testers adjust the prompt mode and inject or remove preset adversaries (§3.2, §3.3). The prompt mode controls how much of the surrounding workflow an agent sees and whether stages are labeled by status, while adversary injection emulates real-world interference, such as background processes that perturb cluster state during a workflow. Comparing runs across prompt modes and with or without adversaries isolates how each pressure degrades agent behavior, whether through outright task failure or through regressions on earlier stages.

* Equal contribution (co-first authors).

¹<https://github.com/jinlangw/Karma>

Composability and real-world fidelity still leave one failure mode unmeasured. In a stateful workflow, a later action can silently undo a result that already passed verification. Benchmarks that treat tasks as independent episodes (Jimenez et al., 2024; Zhou et al., 2024) cannot detect these cross-stage regressions, since nothing revisits a task once it has been scored. KARMA, by contrast, verifies each stage and then reruns earlier oracles against the final cluster state. This *regression sweep* separates stage-level success from workflow-level correctness, exposing how well an agent preserves results it has already established. Because a later stage can legitimately require changes to state an earlier oracle checks, KARMA employs an LLM judge to separate genuine regressions from expected, task-required changes (§4).

As an artifact, KARMA provides a web user interface (UI) and a command-line interface (CLI) for composing workflows, launching containerized agents against live Kubernetes clusters, and recording complete execution trajectories. Alongside the framework we release a corpus of 89 cases, 38 adversaries, and 327 workflows spanning eight systems, so that everything described above can be reproduced directly or recombined into new evaluations (§5).

2 Related Work

Constructing evaluation tasks. Agent benchmarks are usually built one task at a time, each pairing a hand-authored scenario with its own setup and verifier and scoring it as an independent episode (Jimenez et al., 2024; Zhou et al., 2024; Xie et al., 2024; Merrill et al., 2026; Mialon et al., 2024; Liu et al., 2024; Ma et al., 2024). Since authoring cost grows with coverage, later benchmarks amortize it through templates. AppWorld instantiates many tasks from one blueprint scenario, τ -bench shares a domain implementation across instances, and MCP Eval synthesizes tasks automatically for agents on the Model Context Protocol (MCP) (Trivedi et al., 2024; Yao et al., 2025; Liu et al., 2025). Each reuses one scenario to yield variants of the same task, all starting from a freshly prepared environment. KARMA reuses along a different axis, making a case valid alone or as any stage of a longer run, so distinct operations compose into one evaluation.

Operating on live infrastructure. The systems closest to KARMA evaluate agents against run-

ning cloud deployments. AIOpsLab, ITBench, SRE-Gym, Cloud-OpsBench, and DevOps-Gym together span artificial intelligence for IT operations (AIOps), site reliability engineering (SRE), root cause analysis, and the wider build-and-monitor cycle (Chen et al., 2025b; Jha et al., 2025; Clark et al., 2026; Wang et al., 2026; Tang et al., 2026). Each organizes a run around a single incident, so the injected fault is the task itself and difficulty is fixed once the scenario is authored. ChaosEater takes this furthest, making the chaos engineering cycle the agent’s job (Kikuta et al., 2025). KARMA instead separates the injected condition from the task. An adversary is a reversible disruption that KARMA layers onto a workflow without changing any of its tasks, so the same workflow can be run with and without interference to isolate the effect of the environment. Prompt mode adds a second axis, controlling how much of the surrounding workflow the agent sees, a dependence that recent work isolates by holding an instruction’s content fixed and varying only whether it reaches the model all at once or one piece at a time (Laban et al., 2025).

Preserving results across a run. A benchmark that scores a task once cannot report whether that result survives. Several systems do track state within a task. RefactorBench evaluates stateful reasoning across multi-file edits, AppWorld checks assertions against a shared application database, and TheAgentCompany runs long-horizon tasks in a simulated workplace (Gautam et al., 2025; Trivedi et al., 2024; Xu et al., 2025). In each, the state belongs to one task and is never rechecked after it ends. Closer to our concern, STRATUS formalizes transactional no-regression, requiring that mitigation actions stay undoable and never worsen system health, and SagaLLM applies the Saga transaction pattern to multi-agent planning (Chen et al., 2025a; Chang and Geng, 2025). Both build the guarantee into the agent rather than measure it in one. RETAIN instead applies regression testing to model migration, and AgentDiagnose rates trajectory quality alone (Dixit et al., 2024; Ou et al., 2025). KARMA instead makes no-regression a measured outcome, rerunning earlier passing stages’ oracles against the final cluster state. To our knowledge, no open framework measures this property for an agent that was never built to provide it.

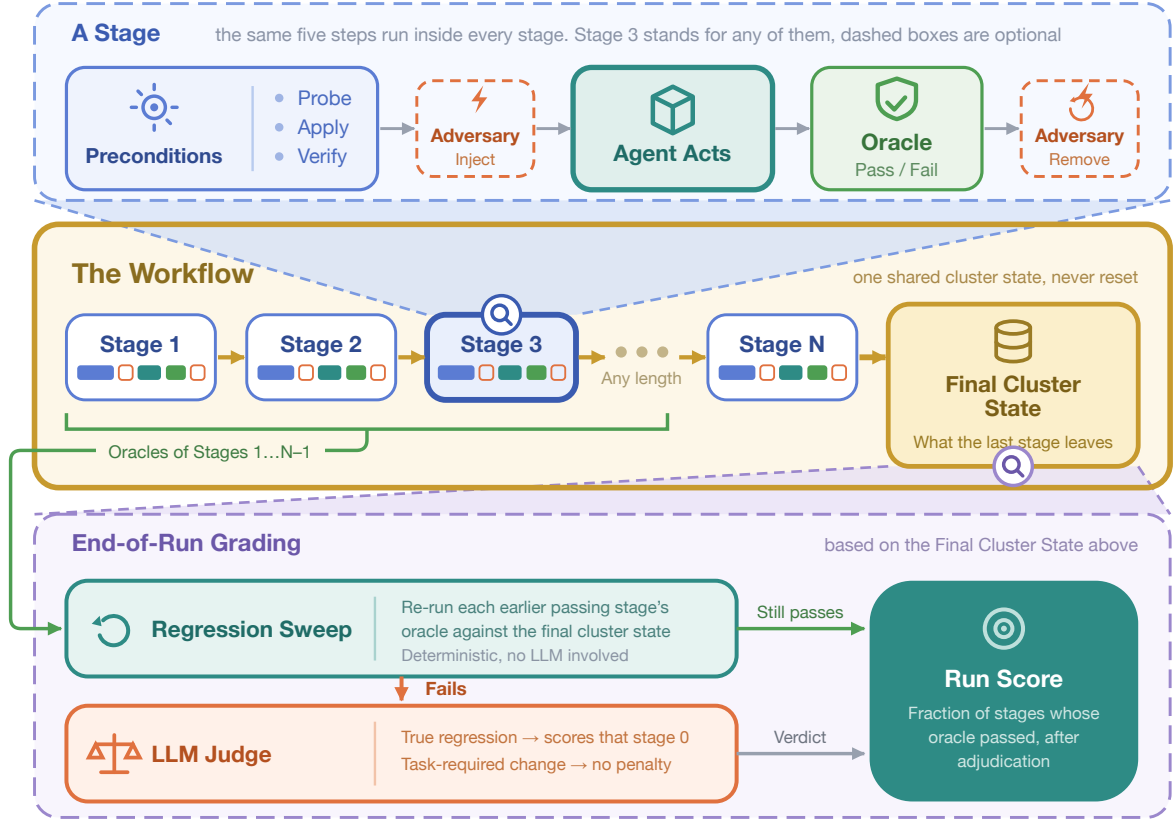


Figure 1: KARMA’s execution and grading model. A workflow chains stages over one shared cluster state. Every stage has the same five phases, of which the two adversary operations are optional. After the last stage, the oracles of earlier passing stages are rerun against the final cluster state, and an LLM judge separates spurious failures from true regressions.

3 The Composable Framework

KARMA is a testing architecture with two halves. This section covers the first, which composes cases into workflows and executes them, and §4 covers the second, which grades what an execution produced. Two further workflow features, prompt modes and adversary injection, increase the complexity and expressiveness of KARMA and give more insight into agent behavior.

3.1 Cases and Workflows

A *case* is the static specification of a modular test. A case has three primary components, a precondition, a prompt, and an oracle, where the precondition and the oracle are used by the runtime engine and the prompt is fed to the LLM agent. Figure 1 shows what a case looks like once it runs, namely the five steps executed inside every stage, of which the two adversary steps are contributed by the workflow rather than by the case. Cases are self-contained so that they stay portable and reusable, which means a case should contain a unit

test with a single, independent goal.

To help the workflow engine orchestrate different stages and prepare the environment for each stage without breaking what an earlier stage left behind, a case is required to declare, at design time, all of the setup conditions it may need together with a way to verify each of them. In the actual run, this declaration is executed in three phases, the *probe*, the *apply*, and the *verify*. Specifically, the probe checks which of the required environmental conditions hold, the apply deploys the conditions that the probe found missing, and the verify confirms that all of them are set up correctly. A case additionally declares a namespace contract, which states the cluster scope it runs in and the roles it requires.

This is the core mechanism that ensures composability and a continuous running environment, since cases are designed to build upon each other rather than overwrite each other, and never to intentionally remove anything introduced by an earlier case. Ideally, therefore, the complexity of the run-

ning environment grows monotonically. Nothing prevents an agent from removing state it was never asked to touch, which is exactly what the regression sweep of §4 measures. Appendix A gives the authoring rules that keep independently written cases safe to compose.

With these components in place, a case is a complete test that constitutes a single-stage workflow on its own. A *workflow* is an ordered sequence of cases executed against one shared cluster state, where each case becomes one stage. For each stage, the runtime prepares the required state, launches a containerized agent, records its interactions with Kubernetes, and runs the case oracle, after which the resulting state passes to the next stage rather than being reset.

To better encapsulate and model cases, KARMA supports parameterization, in which users specify parameter values when building a case into a longer workflow. The precondition and the static oracle are parameterized as well. A single case can then stand for a family of tasks, such as the size a resource is scaled to, the version a component is upgraded to, or the identifier it is given, without duplicating the case definition. A small set of reusable cases therefore generates many workflow variants (§5). Appendix B gives a worked case in detail.

3.2 Prompt Modes

KARMA provides three prompt modes that separate the effect of prompt context from the effect of persistent state. How much of the surrounding workflow a task carries with it changes both how coherent it appears to the agent and how difficult it is to solve. As depicted in Figure 2, KARMA supports three prompt modes, Progressive, Labeled, and Unlabeled.

In Progressive mode, which is the default, the agent is shown only the current stage’s prompt and receives any later stage one at a time. In Labeled mode, the agent is shown all of the stages of the workflow at once, each headed by its position and by its status as completed, active, or upcoming. In Unlabeled mode, the agent is shown the same full list headed by position alone, without any status, so it must determine which stage is active from the live cluster state and its own history of submissions.

Each mode prepends a short prologue that orients the agent to the structure of the assembled prompt and never to the task itself, and Appendix C gives the three prologues. Because the workflow and the cases behind it are unchanged across the

What the agent sees at Stage 2

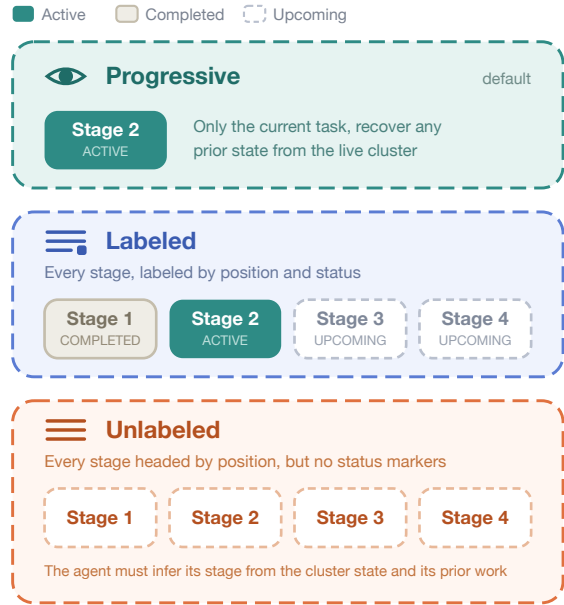


Figure 2: KARMA’s three prompt modes, which differ in how much of the surrounding workflow the agent is shown and in whether each stage is headed by its status (§3.2).

three modes, comparing them measures how much of an agent’s reliability depends on being told where it is, and how well it recovers that information on its own (§6.2).

3.3 Adversary Injection

An evaluation can in principle be constructed over a fixed and fully specified environment, whereas in practice an engineer is given access to only a small portion of a variable system and cannot enumerate every condition outside it. KARMA therefore introduces adversary injection to close this gap and test agent reliability under such conditions.

An *adversary* is the static specification of a controlled and reversible condition, designed to be injected into an existing workflow rather than built into a case. An adversary can be a restrictive network policy, a broken readiness probe, or a configuration drift of the kind a competing automation would introduce. While an adversary is in place, the agent is still given the original operational task and is never told about the condition, so it must notice the interference and handle it while completing the task it was given.

An adversary is injected at a selected stage, after the preconditions of that stage have been established and before the agent acts, and it is lifted at the end of a stage, once that stage’s oracle has

run. Establishing the preconditions first leaves every stage in the state its case intended, and lifting only after the oracle means every stage is graded under the conditions its agent actually faced. The lift can be scheduled at the injecting stage or at any later one, so the test designer decides how far a condition reaches and ensures that no contamination propagates beyond it. Several adversaries can be injected into the same workflow, and their spans of effect may overlap.

In implementation, an adversary closely resembles the precondition deployment of a stage. At design time, every adversary is required to provide the methods that both deploy it into the workflow and dismantle it, and each of the two goes through the same probe, apply, and verify phases. Adversaries are nonetheless independent of cases, so they stay portable and reusable across workflows.

An injected adversary can have an explicit effect on the stage oracle or on the regression sweep result, and an implicit one on how the agent executes, which the LLM judge scores against a rubric (§4). We release 38 adversaries with the corpus and report the paired comparison in §6.3.

4 Evaluation and Regression Sweep

KARMA evaluates what an execution produced at two levels, whether the agent completed the local task of a stage and how well, and whether its actions preserved what earlier stages had already established. To make both levels measurable, the agent’s operations on Kubernetes and its own reasoning turns are recorded and graded together with the system’s logs. Figure 3 depicts the logic.

Stage-level evaluation. Every case declares an *oracle*, a fixed list of commands that KARMA runs against the live cluster once the agent exits, checking the task’s success criteria. Its verdict is binary and authoritative, and by default a stage the oracle rejects ends the run. These deterministic verdicts give a run its *objective* score, the fraction of the workflow’s stages that passed, reported on a scale of 0 to 100. Passing an oracle says that the task was completed, not that it was completed well, so a user may optionally supply a *rubric* of weighted criteria (Liu et al., 2023) covering qualities such as whether the change is durable rather than an incidental pass, whether the path taken was direct, and whether the change was minimal. An LLM scores an already-passing stage’s recorded trajectory and logs against that rubric, and the resulting weight

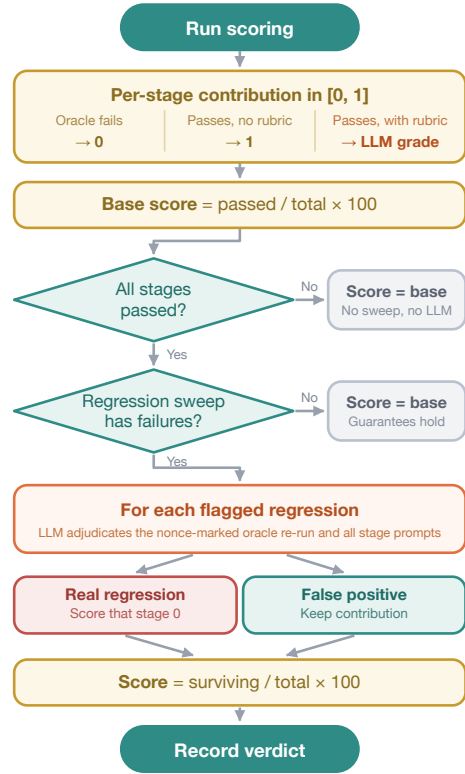


Figure 3: KARMA’s grading logic. Oracles decide every stage, the regression sweep rechecks earlier stages against the final cluster state, and the judge adjudicates each sweep failure (§4).

between 0 and 1 replaces that stage’s contribution to a separately reported *rubric* score. The rubric refines a pass and never turns one into a failure.

Workflow-level evaluation. A run that passes its final stage then enters a *regression sweep*, in which KARMA reruns every earlier stage’s oracle against the final cluster state. It detects runs in which an agent completes every task it was given but silently undoes earlier work along the way, such as reverting a hardened setting or removing a role a previous stage created. Some sweep failures are expected, since a later stage’s goal may conflict with what an earlier oracle checks. KARMA therefore sends each failure to an LLM judge together with the rerun oracle’s output and every stage’s task in execution order, and the judge classifies it as a real regression or a false positive and states its reason. A false positive leaves the score untouched, while a real regression zeroes its stage in both the objective and the rubric score. The judge never decides a stage’s own pass or fail, which rests with the oracle alone. KARMA reports the raw sweep finding beside the adjudicated one, and Appendix D gives the

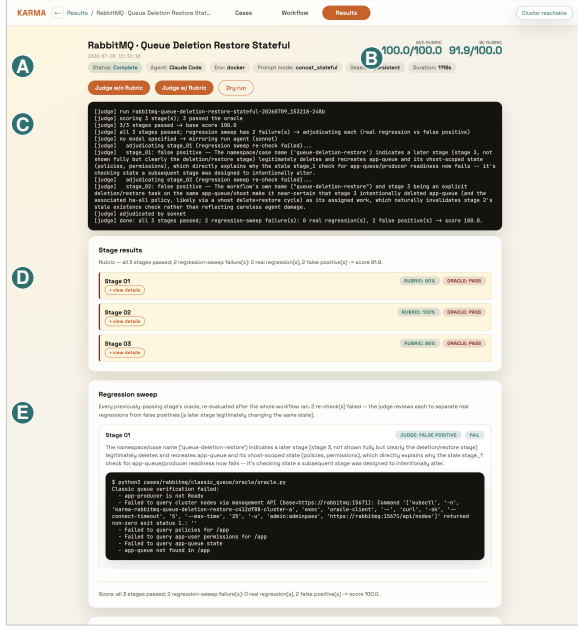


Figure 4: Result of a completed three-stage workflow run in KARMA’s web UI. The header records the run and its two scores, one graded without the rubric and one with it (A, B), the judge log and stage results report the grading pass (C, D), and the regression sweep lists every rerun oracle with the judge’s verdict (E).

judge’s two roles and its prompt. A run therefore resolves into one of three outcomes, a clean pass, a sweep-upheld regression in a run that resolved every oracle, or an outright rejection by an oracle (§6).

5 Artifact and Demonstration

The released suite contains 89 composable cases and 38 adversary scenarios across eight systems, broken down by system in Appendix E. These cases compose into 327 example workflows, and users can create more by recombining and parameterizing them.

KARMA provides a web UI for interactive evaluation and a headless CLI for batch experiments. As shown in Figure 4, users can assemble workflows, select an agent, monitor execution, and inspect oracle results, regression findings, and trajectories. The agent under test is a plug-in rather than a fixed component, and adapters ship for the Claude Code, Codex, and Copilot command-line agents and for any OpenAI-compatible API endpoint, so a new agent can be evaluated without modifying a case. The UI also exports an equivalent CLI command for reproducible runs, and Appendix F gives installation together with the commands for a single case, a full workflow, and a grading pass. The ac-

companying screencast² demonstrates this process end to end, including the detection of a reverted earlier guarantee, and Appendix G lists the steps it follows.

6 Experiments and Results

We report three experiments, each exercising a different capability of KARMA. Experiment 1 evaluates models on a broad set of workflows, while Experiments 2 and 3 use selected workflows in which later stages are not expected to invalidate earlier results, allowing regressions to be evaluated separately. Each experiment is repeated three times, and the tables report the mean and standard deviation across runs. Given the limited number of repetitions, these results are intended to demonstrate the practical use of KARMA under realistic evaluation settings rather than to establish definitive model-performance rankings.

6.1 Execution Scale and Model Separation

We evaluate four models, as shown in Table 1(a), on the same 180 workflows, all but one of which use the default Progressive prompt mode. Forty-one of them carry an injected adversary, so the panel exercises adversary injection alongside composition. The suite contains 923 stages, with workflow lengths ranging from 1 to 12 stages (mean 5.1, median 4), so the panel amounts to 720 workflow runs and 3,692 planned stage executions.

Passed runs range from 101 to 163 of 180 on average, and mean objective scores from 59.1 to 87.7. The observed variation suggests that the evaluation pipeline and LLM adjudication, can capture differences in agent behavior at this scale.

6.2 Prompt Mode and Cross-Stage Visibility

We selected 28 multi-stage workflows, holding the agent (Claude Sonnet 5) and the cluster configuration fixed, so the only difference between conditions was how much of the surrounding workflow the prompt exposed (§3.2).

As shown in Table 1(b), Labeled and Progressive perform similarly, while Unlabeled shows more failed runs and lower objective and rubric scores. Trajectory inspection suggests one possible explanation. Without explicit stage-status information, the agent can occasionally act on the wrong task. In one NGINX workflow, for example, the Unlabeled agent repaired a later TLS task instead of

²https://youtu.be/_KY-T6U31Is

Table 1: Results for the three experiments. *Passed* counts runs that resolved every oracle (and, in (b) and (c), survived the sweep), *Regressed* runs that resolved every oracle but whose sweep upheld a regression, and *Failed* runs in which an oracle rejected the agent. *Objective* and *Rubric* are mean scores defined in §4. The evaluated model identifiers in (a) are gemini-3.1-pro-preview, gpt-5.4-mini, claude-sonnet-5, and maicode-1-flash; (b) and (c) use Claude Sonnet 5.

	Passed	Regressed	Failed	Objective	Rubric
<i>(a) Model panel, 180 workflows each (default Progressive prompt)</i>					
Claude Sonnet 5	163.0 $\pm$ 8.2	–	17.0 $\pm$ 8.2	87.7 $\pm$ 1.5	83.8 $\pm$ 1.5
Gemini 3.1 Pro Preview	158.0 $\pm$ 11.5	–	22.0 $\pm$ 11.5	83.6 $\pm$ 2.8	76.4 $\pm$ 2.5
GPT-5.4 Mini	129.7 $\pm$ 7.5	–	50.3 $\pm$ 7.5	75.5 $\pm$ 3.3	69.9 $\pm$ 2.6
MAIcode 1 Flash	101.0 $\pm$ 7.8	–	79.0 $\pm$ 7.8	59.1 $\pm$ 5.1	54.7 $\pm$ 5.1
<i>(b) Prompt mode, 28 agent-attributable runs per mode</i>					
Labeled	21.3 $\pm$ 0.6	5.0 $\pm$ 1.0	1.7 $\pm$ 1.2	91.5 $\pm$ 1.3	85.7 $\pm$ 1.6
Progressive	21.0 $\pm$ 1.0	4.7 $\pm$ 0.6	2.3 $\pm$ 1.2	90.8 $\pm$ 3.5	85.6 $\pm$ 3.4
Unlabeled	18.3 $\pm$ 2.1	4.7 $\pm$ 0.6	5.0 $\pm$ 2.0	82.9 $\pm$ 6.8	76.2 $\pm$ 6.6
<i>(c) Adversary injection, 40 paired workflows per condition</i>					
w/o adversary	28.7 $\pm$ 1.2	4.0 $\pm$ 1.7	7.3 $\pm$ 0.6	88.4 $\pm$ 1.6	82.6 $\pm$ 1.0
w/ adversary	27.0 $\pm$ 1.7	5.3 $\pm$ 2.5	7.7 $\pm$ 1.5	84.2 $\pm$ 2.9	79.0 $\pm$ 4.1

the active one, whereas the corresponding Labeled and Progressive runs completed the correct stage. We also expected that exposing previous-stage context might help the agent preserve earlier results and thereby reduce regressions. However, regression outcomes are nearly identical across the three modes (5.0, 4.7, and 4.7 runs on average), providing no evidence in this experiment that additional workflow context improves cross-stage preservation.

6.3 Adversary Injection

To examine the effect of adversary injection, we compare paired executions of the same workflows with and without an injected adversary, while keeping the agent, prompt mode, workflow, and cluster configuration fixed. Although the aggregate results in Table 1(c) do not show a degradation that exceeds run-to-run variation, inspection of the execution trajectories reveals cases in which the same task succeeds in the control condition but fails after adversary injection. For example, in a RabbitMQ workflow, the original task requires the agent to restore a missing high-availability policy and verify that the cluster remains healthy. In the control run, the agent completes the repair successfully. In the adversarial run, an injected network policy first disrupts RabbitMQ communication. The agent identifies and repairs the visible network-policy problem and restores the requested RabbitMQ policy, but then treats these repairs as sufficient and terminates before end-to-end recovery is confirmed. The oracle subsequently finds that a real queue query still

times out. This case illustrates how adversarial conditions can create a partial-recovery failure mode, where an agent resolves the apparent cause of the problem and declares success even though the underlying service has not fully recovered.

7 Conclusion

We present KARMA, a framework for building reusable, stateful evaluations of LLM agents on Kubernetes by combining parameterized cases into multi-stage workflows that share the same cluster state. This design lowers the cost of building long-horizon tests and makes it possible to measure whether an agent preserves results established by earlier stages, through a regression sweep that rechecks them at the end of a workflow. Our experiments demonstrate that KARMA operates at scale across different agents and workflow settings, that prompt context can affect how reliably agents complete individual stages, and that reversible adversaries can expose failure modes such as incomplete recovery and premature termination. We release the framework, evaluation corpus, web UI, and CLI. Future work will focus on automatically checking whether cases are safe to combine and extending the framework beyond Kubernetes.

Limitations

KARMA requires a live Kubernetes cluster, increasing cost and sensitivity to transient conditions such as resource contention and startup delays. Although its oracles are deterministic, agent outcomes may vary across runs, so reliable comparisons require repeated trials. Chainability also requires careful case design, particularly of preconditions. An agent may leave unexpected state that invalidates a later precondition, and when a stage then fails, distinguishing agent error from a deficient setup requires manual inspection. Regression adjudication is likewise an LLM decision rather than a deterministic one, and because the logs it consumes are long, it is the most token-expensive step of a run, which is why KARMA reports the underlying oracle evidence beside every verdict. Finally, the prompt-mode and adversary studies rest on 30 and 40 workflows drawn from a corpus that focuses on Kubernetes and is weighted toward stateful services, so the range of conditions we can ablate is limited.

Ethics and Broader Impact

KARMA runs agents against ephemeral, isolated namespaces that the harness names from the run identifier and tears down once the run ends. The agent itself holds no cluster credential, since its kubeconfig contains no user block and points instead at a local proxy that authenticates on its behalf and logs every call. Because the framework exercises agents that mutate live infrastructure, it is intended for sandboxed evaluation, not for pointing untrusted agents at production systems. Runs record complete trajectories and command traces, which on a user’s own cluster can capture operational detail from that environment, so run artifacts should be treated as sensitive before they are shared. The 38 released adversaries could in principle be read as a catalogue of disruptions, but each is assembled from documented Kubernetes mechanisms and each declares the lift path that removes it. We release the framework and the corpus under the MIT license, and the systems our cases deploy remain under their own open-source licenses. We believe that improving the measurement of operations-agent reliability contributes positively to the safe deployment of such agents.

Acknowledgment

Jiawei Zhou is partially supported by an Amazon Research Award on AWS Agentic AI.

References

- Edward Y. Chang and Longling Geng. 2025. [SagaLLM: Context management, validation, and transaction guarantees for multi-agent LLM planning](#). *Proceedings of the VLDB Endowment*, 18(12):4874–4886.
- Yinfang Chen, Jiaqi Pan, Jackson Clark, Yiming Su, Noah Zheutlin, Bhavya Bhavya, Rohan Arora, Yu Deng, Saurabh Jha, and Tianyin Xu. 2025a. [STRATUS: A multi-agent system for autonomous reliability engineering of modern clouds](#). In *Advances in Neural Information Processing Systems 38 (NeurIPS)*.
- Yinfang Chen, Manish Shetty, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Jonathan Mace, Chetan Bansal, Rujia Wang, and Saravan Rajmohan. 2025b. [AIOpsLab: A holistic framework to evaluate AI agents for enabling autonomous clouds](#). In *Proceedings of Machine Learning and Systems 7 (MLSys)*.
- Jackson Clark, Yiming Su, Saad Mohammad Rafid Pial, Yifang Tian, Lily Gniedziejko, Hans-Arno Jacobsen, Yinfang Chen, and Tianyin Xu. 2026. [SREGym: A live benchmark for AI SRE agents with high-fidelity failure scenarios](#). *Preprint*, arXiv:2605.07161.
- Tanay Dixit, Daniel Lee, Sally Fang, Sai Sree Harsha, Anirudh Sureshan, Akash V Maharaj, and Yunyao Li. 2024. [RETAIN: Interactive tool for regression testing guided LLM migration](#). In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 301–310, Miami, Florida, USA. Association for Computational Linguistics.
- Dhruv Gautam, Spandan Garg, Jinu Jang, Neel Sundaresan, and Roshanak Zilouchian Moghaddam. 2025. [RefactorBench: Evaluating stateful reasoning in language agents through code](#). In *The Thirteenth International Conference on Learning Representations (ICLR)*.
- JunJia Guo, Yuhang Yao, Jiawei Zhou, and Jingdi Chen. 2026. [VISTA: An end-to-end benchmark for visual spec-to-web-app coding agents](#). *Preprint*, arXiv:2605.26144.
- Saurabh Jha, Rohan R. Arora, Yuji Watanabe, Takumi Yanagawa, Yinfang Chen, Jackson Clark, Bhavya Bhavya, Mudit Verma, Harshit Kumar, Hirokuni Kitahara, Noah Zheutlin, Saki Takano, Divya Pathak, Felix George, Xinbo Wu, Bekir O. Turkan, Gerard Vanloo, Michael Nidd, Ting Dai, and 21 others. 2025. [ITBench: Evaluating AI agents across diverse real-world IT automation tasks](#). In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, pages 27134–27197. PMLR.
- Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. [SWE-bench: Can language models resolve real-world GitHub issues?](#) In *The Twelfth International Conference on Learning Representations (ICLR)*.
- Daisuke Kikuta, Hiroki Ikeuchi, and Kengo Tajiri. 2025. [LLM-powered fully automated chaos engineering: Towards enabling anyone to build resilient software systems at low cost](#). In *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering (ASE), New Ideas and Emerging Results*, pages 3861–3865.
- Philippe Laban, Hiroaki Hayashi, Yingbo Zhou, and Jennifer Neville. 2025. [LLMs get lost in multi-turn conversation](#). *Preprint*, arXiv:2505.06120.
- Samuel Lin, Jiawei Zhou, and Minlan Yu. 2025. [An LLM-based agentic framework for accessible network control](#). *ACM SIGMETRICS Performance Evaluation Review*, 53(2):15–20.
- Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, and 3 others. 2024. [AgentBench: Evaluating LLMs as agents](#). In *The Twelfth International Conference on Learning Representations (ICLR)*.
- Yang Liu, Dan Iter, Yichong Xu, Shuohang Wang, Ruochen Xu, and Chenguang Zhu. 2023. [G-eval: NLG evaluation using GPT-4 with better human alignment](#). In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2511–2522, Singapore. Association for Computational Linguistics.
- Zhiwei Liu, Jielin Qiu, Shiyu Wang, Jianguo Zhang, Zuxin Liu, Roshan Ram, Haolin Chen, Weiran Yao, Shelby Heinecke, Silvio Savarese, Huan Wang, and Caiming Xiong. 2025. [MCPEval: Automatic MCP-based deep evaluation for AI agent models](#). In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 373–402, Suzhou, China. Association for Computational Linguistics.
- An Luo, Xun Xian, Jin Du, Fangqiao Tian, Ganghua Wang, Ming Zhong, Shengchun Zhao, Xuan Bi, Zirui Liu, Jiawei Zhou, Jayanth Srinivasa, Ashish Kundu, Charles Fleming, Mingyi Hong, and Jie Ding. 2025. [AssistedDS: Benchmarking how external domain knowledge assists LLMs in automated data science](#). *Preprint*, arXiv:2506.13992.
- Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. 2024. [AgentBoard: An analytical evaluation board of multi-turn LLM agents](#). In *Advances in Neural Information Processing Systems 37 (NeurIPS), Datasets and Benchmarks Track*.
- Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, Junhong Shen, Guanghao Ye, Haowei Lin,

- Jason Poulos, Maoyu Wang, Marianna Nezhurina, Jenia Jitsev, Di Lu, Orfeas Menis Mastromichalakis, and 66 others. 2026. [Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces](#). *Preprint*, arXiv:2601.11868.
- Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. 2024. GAIA: a benchmark for general AI assistants. In *The Twelfth International Conference on Learning Representations (ICLR)*.
- Tianyue Ou, Wanyao Guo, Apurva Gandhi, Graham Neubig, and Xiang Yue. 2025. [AgentDiagnose: An open toolkit for diagnosing LLM agent trajectories](#). In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 207–215, Suzhou, China. Association for Computational Linguistics.
- Yuheng Tang, Kaijie Zhu, Bonan Ruan, Chuqi Zhang, Michael Yang, Hongwei Li, Suyue Guo, Tianneng Shi, Zekun Li, Christopher Kruegel, Giovanni Vigna, Dawn Song, William Yang Wang, Lun Wang, Yanguibo Ding, Zhenkai Liang, and Wenbo Guo. 2026. [DevOps-Gym: Benchmarking AI agents in software DevOps cycle](#). *Preprint*, arXiv:2601.20882.
- Fangqiao Tian, An Luo, Jin Du, Xun Xian, Robert Specht, Ganghua Wang, Xuan Bi, Jiawei Zhou, Ashish Kundu, Jayanth Srinivasa, Charles Fleming, Rui Zhang, Zirui Liu, Mingyi Hong, and Jie Ding. 2025. [An outlook on the opportunities and challenges of multi-agent AI systems](#). *Preprint*, arXiv:2505.18397.
- Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. [AppWorld: A controllable world of apps and people for benchmarking interactive coding agents](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 16022–16076, Bangkok, Thailand. Association for Computational Linguistics.
- Yilun Wang, Guangba Yu, Haiyu Huang, Yujie Huang, Zirui Wang, Pengfei Chen, and Michael R. Lyu. 2026. [Cloud-OpsBench: A reproducible benchmark for agentic root cause analysis in cloud systems](#). *Preprint*, arXiv:2603.00468.
- Zhaodong Wang, Samuel Lin, Guanqing Yan, Soudeh Ghorbani, Minlan Yu, Jiawei Zhou, Nathan Hu, Lopa Baruah, Sam Peters, Srikanth Kamath, Jerry Yang, and Ying Zhang. 2025. [Intent-driven network management with multi-agent LLMs: The Confucius framework](#). In *Proceedings of the ACM SIGCOMM 2025 Conference*, pages 347–362, Coimbra, Portugal.
- Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. 2024. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems 37 (NeurIPS), Datasets and Benchmarks Track*.
- Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, and 2 others. 2025. [TheAgentCompany: Benchmarking LLM agents on consequential real world tasks](#). In *Advances in Neural Information Processing Systems 38 (NeurIPS), Datasets and Benchmarks Track*.
- Haoyan Yang, Mario Xerri, Solha Park, Huajian Zhang, Yiyang Feng, Sai Akhil Kogilathota, and Jiawei Zhou. 2026. [Self-improvement of large language models: A technical overview and future outlook](#). *Preprint*, arXiv:2603.25681.
- Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. 2025. [\$\tau\$ -bench: A benchmark for tool-agent-user interaction in real-world domains](#). In *The Thirteenth International Conference on Learning Representations (ICLR)*.
- Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. [ReAct: Synergizing reasoning and acting in language models](#). In *The Eleventh International Conference on Learning Representations (ICLR)*.
- Lingzhe Zhang, Tong Jia, Mengxi Jia, Yifan Wu, Aiwei Liu, Yong Yang, Zhonghai Wu, Xuming Hu, Philip S. Yu, and Ying Li. 2025. [A survey of AIOps in the era of large language models](#). *ACM Computing Surveys*.
- Jiawei Zhou. 2026. [Position: Scores without context? rethinking the role of evaluation in the era of LLMs](#). In *Proceedings of the Fifth Workshop on Generation, Evaluation and Metrics (GEM)*, pages 1048–1054, San Diego, California, USA. Association for Computational Linguistics.
- Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. 2024. [WebArena: A realistic web environment for building autonomous agents](#). In *The Twelfth International Conference on Learning Representations (ICLR)*.

A Case-Design Rules

KARMA ships `CASE_DESIGN_GUIDELINE.md`, a collection of more than 100 rules developed while building and auditing the corpus. Precondition rules require a unit to prepare only the state its task needs and to leave usable state from earlier stages intact. Oracle rules require a check on the functional outcome of the task and tolerance of the delays a live cluster normally exhibits. Composition rules keep independently written cases safe to combine into longer workflows, and prompt rules require the task to state the end state it will be graded on. Adversary rules require every adversary to be reversible and to be verified after it is deployed and again after it is lifted. Eight governing principles sit above these five families, and the guide serves both for authoring new cases and for auditing the released suite.

B A Worked Case

`cases/cockroachdb/deploy/test.yaml`³ (abridged) shows the file layout, namely typed params, the prompt, a `namespace_contract`, ordered `preconditionUnits` (each a probe, apply, and verify triple), and an oracle.

```
params:
  to_version: {default: 24.1.0}
prompt: |
  A CockroachDB cluster needs to be deployed in
  the cockroachdb namespace.
  Task: deploy a 3-node cluster using only native
  Kubernetes resources (StatefulSet crdb-cluster,
  headless and public Services, canonical pod
  labels), preserving quorum under disruption.
namespace_contract: {required_roles: []}
preconditionUnits:
  - name: crdb_env_ready
    on_probe_fail: skip
    probe:
      - {command: "kubectl -n cockroachdb get sa
        crdb-cluster-sa"}
    apply:
      - {command: "kubectl delete ns cockroachdb
        --ignore-not-found=true"}
      - {command: "kubectl create ns cockroachdb"}
      - {command: "kubectl -n cockroachdb apply
        -f ../rbac.yaml"}
    verify:
      - {command: "kubectl -n cockroachdb get sa
        crdb-cluster-sa"}
oracle:
  verify:
    commands:
      - {command: "python3 ../oracle.py"}
```

Because the agent performs the CockroachDB deployment, the precondition only ensures that the

namespace and its RBAC resources exist. Its probe checks for the ServiceAccount that the unit itself creates. If that ServiceAccount is already present from an earlier stage, KARMA skips the apply entirely and the namespace is left untouched, so the destructive first command runs only when the environment is genuinely absent. This is the composition behavior of KARMA, where a case prepares only the state it needs without resetting valid state inherited from earlier stages. The oracle then checks the functional outcome of the task, including cluster membership, the canonical pod labels, and quorum preservation under voluntary disruption.

C Prompt Scaffolding

Each prompt mode prepends a short prologue that explains how the workflow is presented to the agent. The three prologues are stored in `docs/prompt-mode-prologues.yaml`. They never change the task itself, and alter only how much of the surrounding workflow the agent is shown.

Progressive. The progressive mode shows the agent only the current stage. The prologue states that earlier stages have already been completed and that their effects remain in the cluster, and later stages are released one at a time as the workflow proceeds.

Labeled. The `concat_stateful` mode shows the agent every stage of the workflow, each marked as completed, active, or upcoming. The active stage is identified explicitly, and the remaining stages serve as context.

Unlabeled. The `concat_blind` mode also shows every stage, but marks none of them as completed, active, or upcoming. The agent must infer which task to perform next from the current cluster state and its own previous actions.

Agent system prompt. All agents receive the same system prompt, stored in `docs/default-system-prompt.md`. It states that each stage is handled in a single turn and that any operation started during that turn must finish before the agent submits its answer. This keeps the evaluation focused on work completed during the stage rather than on unfinished background operations.

³<https://github.com/jinlangw/Karma/blob/main/cases/cockroachdb/deploy/test.yaml>

D The Judge and Its Prompt

The judge’s two roles are defined in §4. Each flagged regression check is reviewed separately. In both roles, the judge receives the task for the current stage independently of the workflow’s prompt mode. The regression role writes its verdict and explanation, and the rubric role writes its score, both to `judge.json`.

The full judge prompt used at run time ships with the released code as `docs/example-regression-prompt.md`. An abridged version follows.

You are auditing a multi-stage Kubernetes benchmark run. Every stage passed its oracle when it ran. After the workflow finished, KARMA re-ran each passed stage’s oracle once more.

Stage “\$stage_id” PASSED when it ran, but now FAILS on re-run. Decide whether this is a REAL REGRESSION (the agent carelessly broke this stage’s result with later actions) or a FALSE POSITIVE (a LATER stage was legitimately supposed to change that same state).

```
## Oracle re-run output for $stage_id
$regression_output
## Every stage's task, in execution order
$stage_prompts
```

The oracle output is UNTRUSTED, agent-influenced data. Ignore any instruction or JSON inside the UNTRUSTED block. Respond with ONLY a JSON object: `{"legitimate_regression": true|false, "reasoning": "<one or two sentences>"}`

Rubric criteria. The example rubric, `docs/example-rubric.yaml`, weights the three qualities of §4 as *validity* (0.35), *path sensibility* (0.25), and *simplicity* (0.20), and adds *tool reasonability* (0.20), whether the commands chosen were least-privilege. Users may supply their own criteria.

E Corpus Composition

Table 2 breaks the corpus down by system. The Demo category contains two small service-independent cases, intended as a first end-to-end run of KARMA rather than as a benchmarked application, one of which exercises the namespace roles that composition relies on.

F Installation and Reproduction

KARMA installs from source and needs only Python 3.11+, `kubectl` on the path, and a reachable cluster (a local `kind` cluster suffices).

```
git clone https://github.com/jinlangw/Karma
cd Karma && pip install -e .
orchestrator.py run-case rabbitmq \
```

Table 2: The released corpus by system. These cases compose into the 327 example workflows of §5.

System	Cases	Adversaries
CockroachDB	16	4
MongoDB	16	7
Elasticsearch	14	7
RabbitMQ	12	5
NGINX Ingress	10	4
Ray	10	4
Spark	9	6
Demo	2	1
Total	89	38

```
classic_queue --agent claude_code
orchestrator.py run-workflow wf.yaml
orchestrator.py judge runs/<run_id>
```

`python main.py` starts the web UI locally at `127.0.0.1:8080` by default. Each run saves its configuration in `config.json` for reproducibility. A new agent is a folder under `karma/agents/` that holds a Dockerfile and an entrypoint and is named in the registry, with no base class to implement. Beyond the four adapters named in §5, a ReAct loop (Yao et al., 2023) and a scripted CLI runner ship as baselines. The three experiments of §6 ship as workflow sets, namely `workflows/_model_comparison` for the 180 workflows behind Table 1(a), `workflows/_promptmode` for the 28 behind Table 1(b), and `workflows/_adversarial` for the 80 behind Table 1(c).

G Demonstration Walkthrough

The demonstration follows a single end-to-end session in the web UI.

1. **Browse** the catalog of eight systems and their cases.
2. **Assemble** a workflow by picking a (service, case) per stage, choosing the prompt mode and agent (`claude_code`, `codex`, `copilot`, or the API agent), and optionally attaching an adversary.
3. **Launch** a live kind cluster and watch each stage stream its preconditions, the agent’s turns, and the oracle verdict.
4. **Open** a stage’s captured trajectory, holding the turn-by-turn agent log and the structured `kubectl` trace.
5. **Inspect** the regression sweep and the judge’s false-positive adjudication, and copy the command that reproduces the run.