High Level Overview

The difficulty in our environments is in the *investigation*, not the fix. A cascading failure might resolve with a one-line code change -- but tracing from the user-visible symptom through 5-8 logical hops to find that line is the multi-hour task. Each hop crosses a different boundary (service, infrastructure, conceptual), requires a different investigative action, and every intermediate system is behaving correctly. The only actual bug is at the far end of the chain.

We design environments where: (a) the root cause is separated from symptoms by multiple hops, (b) evidence is scattered across realistic sources accessed through real tool interfaces, and (c) the environment resists shortcuts -- shallow fixes are punished by the causal structure itself.

Design Process

We start with a real failure pattern. Every environment adapts a documented incident or a pattern from production experience. Example: a payment service connection leak causes reconnect storms that flood a shared Redis instance, evicting order service idempotency keys, causing duplicate orders, double charges, and fraud-triggered account suspensions. The agent sees: "customers are locked out." The root cause is a missing `conn.close()` -- 7 hops away.

We map each hop to evidence sources. For each step in the chain, we define what evidence exists, where it lives, and how the agent accesses it. Evidence difficulty increases as you approach root cause -- the fraud suspension is easy to find via Sentry; the connection leak is hard because the payment service has *no errors*.

We make the environment feel lived-in. Thousands of log lines, ongoing Slack conversations, recent unrelated PRs, wrong hypotheses from teammates ("maybe increase Redis memory"), and correct-looking intermediate systems. Sentry shows loud order service cache-miss errors (wrong trail) and nothing for the payment service (significant silence).

Agents use real tool interfaces, not data files. We build lightweight MCP tool servers -- Slack (`search_messages`, `read_channel`, `read_thread`), Sentry (`list_issues`, `get_event`, `get_stacktrace`), Datadog (`query_metrics`, `list_alerts`), PagerDuty (`list_incidents`, `get_timeline`), Email (`list_emails`, `read_email`) -- each backed by SQLite, seeded per environment, and reusable across all environments. Standard bash tools (grep, psql, redis-cli, git) are always available alongside these. The agent interacts with the environment the way an engineer would.

The task prompt mirrors a real page. It starts from the user-visible symptom and is deliberately open-ended: *"Multiple customers report being locked out. The fraud team says accounts were auto-suspended by the duplicate charge rule. Investigate and resolve."* This points at the fraud system -- 6 hops from root cause. The connection between "locked out" and "connection leak" is not guessable from the prompt.

Grading

Outcome over process. "Are duplicate orders still being created?" is robust. "Did the agent check Redis before reading source code?" is brittle. Each environment has 5-8 weighted subscores in a Taiga `Grade` object: core outcome (~25%), root cause understanding (~50%), no collateral damage (~5%), communication (~10%), ancillary fixes (~10%). Programmatic checks verify state changes; LLM judges (run 5x, >90% agreement required) evaluate understanding and communication quality. Grading state runs as root, inaccessible to the agent.

QA Layer: Each environment is QA'd in three layers:

- The author solves it using only the agent's tools
- A blind peer attempts with no knowledge of the solution
- Sonnet/Opus runs verify pass rate, check for shortcuts, and validate grader consistency across 5+ runs

Where Scenarios Come From

Option A: Acquire datasets from companies with Sentry, Datadog, Slack, and RCA documentation -- work backwards from real incidents. Highest fidelity.

Option B: Adapt failure patterns from public postmortems (Knight Capital, CrowdStrike, Cloudflare, GitLab, Meta, Stripe, k8s.af) into purchased real codebases. Both approaches use real, human-written code -- we don't generate synthetic codebases.

Chains can be extended to calibrate difficulty: the same Redis eviction produces a 4-hop task (start from "cache misses"), a 7-hop task (start from "account lockouts"), or a 9-hop task (chargebacks downgrade the merchant account). We target 5-8 hops for <50% frontier model pass rate.

Deep Dive

The core challenge is constructing a scenario where: (a) the root cause is separated from the symptoms by multiple logical hops, (b) the evidence needed to connect those hops exists but is scattered across realistic sources, and (c) the environment resists shortcuts -- an agent can't pattern-match its way to the answer without doing the actual investigative work a senior engineer would do.

This document walks through that design process using a single environment end to end.

What Makes a Good Environment

The difficulty is in the investigation, not the fix. The code change that resolves a cascading Redis eviction might be adding a single `conn.close()`. But tracing from "customers are locked out" through fraud rules, duplicate charges, missing idempotency keys, Redis eviction, and a payment service event flood to find that one missing line -- that's the multi-hour task.

Information is discovered, not given. A senior engineer getting paged at 2am receives "payments are failing for some users." They don't receive a bullet list of which logs to check. The task prompt is sparse; the agent must figure out where to look.

Shallow fixes are punished by the causal structure. Fixing the symptom doesn't fix the problem. Increasing Redis memory treats the symptom -- the payment service leak will fill it again. The grading naturally rewards root cause analysis because surface-level fixes don't produce lasting outcomes.

Multiple SDLC phases emerge naturally. The scenario is designed so that a complete response naturally spans Debugging, Maintenance, and Communication -- not as separate tasks bolted on, but as what a complete incident response looks like.

Designing an Environment, Step by Step

We walk through each step using a single example: a Redis eviction cascade that starts with a connection leak and ends with customer account lockouts.

Step 1: Start with a real failure pattern

Every environment begins with something that actually happened. We adapt real incidents, changing the domain and specifics while preserving the causal structure.

Our example: A payment service leaks connections on its error path. Reconnect storms flood Redis with event backlog. Redis evicts order service keys. Without idempotency protection, retries create duplicate orders. Customers get charged twice. Fraud detection auto-suspends their accounts. The agent sees: "customers are locked out."

Step 2: Map the causal chain

The causal chain is the sequence of hops connecting root cause to visible symptoms. Each hop is where one system's failure becomes the input to the next. This is what separates a training environment from a code puzzle.

```
None
1. Payment service leaks connections on error path (missing
conn.close())
2. Reconnect storms flush event backlog -> Redis flooded with
payment-event keys
3. Redis hits maxmemory, allkeys-lru eviction drops order service keys
4. Order service loses idempotency keys for in-flight orders
5. Retry logic creates duplicate orders (idempotency check fails on
cache miss)
6. Payment gateway charges customer twice
7. Fraud detection rule auto-suspends accounts with duplicate charges
   -> Customer reports: "I'm locked out of my account"
```

Every system along the chain is behaving correctly -- the fraud service, the retry logic, Redis eviction. The only actual bug is a missing `conn.close()` 7 hops away.

Design principles for chains:

- **Each hop is a real consequence, not a trick.** If Redis evicts keys, the order service *will* lose cached data. Difficulty comes from chain length and boundary crossings, not deception.
- **Hops should cross different boundaries** -- service boundaries (root cause in service A, symptoms in service B), infrastructure boundaries (app -> shared resource -> different app), conceptual boundaries (code bug -> business logic -> user-facing effect), temporal boundaries (change from months ago interacts with recent deployment).
- **Each hop should require a different investigative action.** If the agent can find everything by grepping logs, the task collapses. Our chain requires: querying a database, reading application logs, running `redis-cli INFO`, scanning key patterns, reading source code.
- **Symptoms naturally point away from root cause.** "Customers locked out" points at the fraud system. "Duplicate charges" points at the payment gateway. The root cause (connection leak in payment service) is the component with the *least* visible symptoms.

- **Chains can be extended to calibrate difficulty.** The same Redis eviction can produce a 4-hop task (start from "cache misses") or a 7-hop task (start from "account lockouts") or a 9-hop task (duplicate charges trigger chargebacks that downgrade the merchant account). Same root cause, longer chain, harder task.
- **Target 5-8 hops for <50% pass rate.** Below 4, frontier models trace the chain too easily. Above 9, the task risks being unsolvable within context limits.

Step 3: Map evidence to sources

For each hop, decide what evidence exists, where it lives, and how the agent accesses it:

Hop	Evidence	Agent accesses via	Difficulty
7	Account suspended: "fraud_rule: duplicate_charge"	`sentry.list_issues()` or `pagerduty.get_incident()`	Easy
6	Duplicate charge records	`bash`: `psql` query on charges table	Medium
5	Duplicate orders, different transaction IDs	`bash`: `psql` query on orders table	Medium
4	"idempotency key not found, proceeding"	`bash`: `grep` in order service logs	Medium
3	`evicted_keys` spiking, memory at limit	`bash`: `redis-cli INFO` or `datadog.query_metrics()`	Medium
2	Flood of `payment-event:*` keys	`bash`: `redis-cli --scan`	Medium
1	Missing `conn.close()` in except block	`bash`: read source code	Hard -- no errors for payment service

This table is the blueprint. Every evidence source becomes something that needs to be built or seeded. It's ordered by how the agent encounters the evidence (symptoms first), not causal order.

Step 4: Make the environment feel lived-in

A bare causal chain in an empty environment is a puzzle, not a realistic incident. The environment needs:

- **Log volume.** Thousands of lines of normal traffic around the relevant signals. The agent must grep and filter.
- **Normal activity.** Ongoing Slack conversations, recent unrelated PRs, other open tickets.
- **Wrong hypotheses from teammates.** In Slack: "Maybe we need to increase Redis memory." "Could be the load balancer." These aren't traps -- they're what people actually say mid-incident.
- **History at multiple timescales.** Changes from yesterday, last week, last month. Most are irrelevant.
- **Correct-looking components.** In a cascading failure, most systems are behaving correctly. An agent that "fixes" the fraud service or the retry logic has been pulled off track.

For our example: Sentry shows lots of order service cache-miss errors (loud, wrong trail). Sentry shows *nothing* for the payment service (quiet, significant). A recent PR changed retry count from 3 to 5 (unrelated, but plausible culprit). Datadog shows a healthy-looking 94% Redis hit rate; evictions are only visible if the agent queries `redis.evicted_keys` specifically.

Step 5: Build mock tool servers

The agent should access evidence the same way an engineer would -- through tool interfaces, not by reading data files. We build lightweight MCP tool servers that mock real engineering tools:

Tool	MCP interface	Backed by
Slack	`search_messages`, `read_channel`, `read_thread`, `post_message`	SQLite
Sentry	`list_issues`, `get_issue`, `get_event`, `get_stacktrace`	SQLite
Datadog	`query_metrics`, `list_alerts`, `get_dashboard`, `list_monitors`	JSON/SQLite
PagerDuty	`list_incidents`, `get_incident`, `get_timeline`, `acknowledge`	SQLite
Email	`list_emails`, `read_email`	SQLite

Each is a lightweight Python MCP server running inside the container, seeded during `setup_problem`. They're **reusable across environments** -- each environment just seeds different data. Standard terminal tools (bash for logs, code, databases, services) are always available alongside these.

Step 6: Design the task prompt

The prompt reads like what a senior engineer receives on a page. It starts from the user-visible symptom, not an intermediate cause:

```None
Multiple customers have reported being locked out of their accounts over the
past 4 hours. The #support channel shows 8 reports, all from customers who
recently placed orders. The fraud team says the accounts were auto-suspended by
the duplicate charge detection rule. Investigate why legitimate customers are
triggering fraud rules and resolve the issue.
```

This points at the fraud system -- 6 hops from root cause. "Investigate why" is open-ended. The connection between "locked out" and "connection leak in payment service" is not guessable from the prompt.

Step 7: Walk the golden path

Before building, walk through the scenario as an engineer to validate all evidence exists and is accessible:

1. `slack.read_channel("#support")` -- customer reports of lockouts after placing orders
2. `pagerduty.list_incidents()` -- fraud suspension alert
3. `sentry.list_issues(project="fraud-service")` or query DB -- accounts suspended for duplicate charges
4. `psql` charges table -- verify charges are actually duplicated
5. `psql` orders table -- duplicate orders, different transaction IDs
6. `grep "idempotency" /var/log/order-service/app.log` -- "key not found, proceeding"
7. `redis-cli INFO stats` -- `evicted_keys` spiking, memory at limit
8. `redis-cli --scan --pattern '*'` -- flood of `payment-event:*` keys
9. `sentry.list_issues(project="payment-service")` -- nothing. Suspicious.
10. Read `payment-service/handlers/checkout.py` -- missing `conn.close()` in error handler
11. Fix the leak, restart, verify recovery. Unsuspend affected accounts. Write incident report.

The golden path is not the only valid path. Grading must account for multiple valid investigation orders.

Step 8: Define grading

Grading maps from the golden path to Taiga's `Grade` object with weighted subscores.

Core principle: outcome over process. "Are duplicate orders still being created?" is robust. "Did the agent check Redis before reading the payment service code?" is brittle. Process checks are reserved for negative checks (did the agent do something destructive?) and understanding checks via LLM judge (does the incident report demonstrate the agent grasped the full chain?).

Subscore	Weight	Method
Traced duplicate charges to duplicate orders	0.10	LLM judge on transcript
Identified idempotency failure	0.10	LLM judge on transcript
Connected to Redis eviction	0.15	LLM judge on transcript
Found payment service as source of memory pressure	0.15	LLM judge on transcript
Found and fixed the connection leak	0.15	Programmatic file check + LLM judge
System recovers (no duplicate orders on test run)	0.10	Programmatic integration test
Unsuspended affected accounts	0.10	Programmatic DB check
Didn't weaken fraud rules	0.05	Programmatic file check
Incident report quality	0.10	LLM judge on file content

Weight distribution across categories: Core outcome (fix + recovery): ~25%. Root cause understanding (tracing the chain): ~50%. No collateral damage: ~5%. Communication: ~10%. Ancillary fixes: ~10%.

LLM judge design: Each LLM-judged subscore includes 2-3 positive examples, 2-3 negative examples, and explicit partial credit criteria. Judges run 5 times per evaluation; >90% agreement required for reliability.

Anti-hacking: Grading state runs as root (agent is uid 1000). MCP server code is not readable by the agent. Programmatic checks verify actual state changes. LLM judges evaluate transcripts for genuine investigation vs. lucky guesses.

QA before shipping

1. **Author solves it** using only the agent's tools. Catches missing evidence and inaccessible clues.
2. **Blind peer test.** A second engineer attempts with no knowledge of the intended solution.
3. **Model test.** Run Sonnet and Opus against it. Check pass rate is in target range, no shortcuts bypass the investigation, grader is self-consistent across 5+ runs.

Where Scenarios Come From

Two approaches to sourcing

Option A -- Acquired company data. Companies with Sentry, Datadog, and Slack data provide pre-built causal chains with real evidence trails. Highest fidelity, but requires acquisition timelines.

Option B -- Injected failure patterns. Adapt documented failure patterns from public postmortems into purchased real codebases. The pattern provides the causal chain; the codebase provides realistic context. Faster to produce, gives control over difficulty.

Both use real, human-written codebases. We don't generate synthetic code.

Source material

Public postmortems provide real causal chains we adapt:

Incident	Root Cause Pattern	SDLC Phase	Hops
Knight Capital 2012	Silent deployment failure, dead code reactivation via flag reuse	Deploy + Debug	6
CrowdStrike 2024	OOB memory read in kernel driver pushed via update pipeline	Deploy + Verify	4
Cloudflare 2025	Config propagation race -> Rust panic from hard-coded limit	Debug	5
Cloudflare 2019	Catastrophic regex backtracking in WAF, no load test	Deploy + Debug	4
GitLab 2017	Accidental `rm -rf` on prod DB, all 5 backup methods broken	Maintenance	5
Meta 2021	BGP withdrawal cascade, tools depend on broken infra	Debug + Maint	6
Stripe 2019	DB migration locks writes 4 hours, rollback also locks	Maint + Deploy	4
k8s.af collection	Liveness probe misconfig, node OOM, resource limits	Deploy + Maint	3-5

Multi-hop patterns (each adaptable to multiple codebases):

- Memory leak -> Redis fills -> eviction drops session keys -> auth failures cascade (4 hops)
- Schema change -> validation failure -> retry-and-continue masks error -> partial export (4 hops)
- CI passes locally -> stale Docker layer -> breaking dependency change -> misleading error (4 hops)
- Terraform state mismatch -> next apply destroys resource -> service loses DB (3 hops)
- Cert rotation misses 1/12 hosts -> intermittent TLS failures via load balancer (5 hops)

Preventing a priori recognition

If Claude recognizes the scenario from training data, it can shortcut the investigation. We mitigate by changing the domain, changing specifics, adding context that doesn't exist in the public postmortem, and using codebases with no connection to the original incident.

Open Questions

1. **Task composability.** Taiga gives each problem a fresh container. Chaining requires either multi-phase problems with internal checkpoints, or later problems that start from pre-built state. We need to clarify the mechanism with Anthropic.
2. **Codebase scale.** 2K-10K lines is our default range. Larger codebases increase realism but also increase the chance the agent gets lost.
3. **Computer Use vs Tool Use.** Most environments are Tool Use (bash + str_replace_editor). Computer Use (Grafana dashboards, visual debugging) adds realism but requires the `6vcpu+32gib` tier. We propose 80/20 split.
4. **Acquired company data timeline.** Option A produces highest-fidelity environments but depends on acquisition speed. For the pilot, we likely use Option B and develop Option A in parallel.