# CIGE: An Agentic AI Test Case Standard

Vivek Krishna Choppa[1], Anusha Kovi[2]

[1] Amazon.com Services LLC, Washington, USA
choppak@amazon.com
[2] Amazon.com Services LLC, Washington, USA
anukovi@amazon.com

**Abstract.** Test automation is moving from fixed script replay to agent driven execution and judgement. An agent may read a goal, inspect product state, call tools, recover from small changes, and collect evidence before reporting a result. Existing test case formats do not give enough structure for this style of execution. Plain prompts mix setup, purpose, rules, and steps. Traditional scripts are repeatable, but they break when the product flow changes. This paper presents CIGE, a simple structure for agentic test cases with four fields: Context, Intent, Guardrails, and Execution. Context grounds the run. Intent captures the reason the test exists. Guardrails define what the agents must not do. Execution gives the first path to try and can be repaired when the product changes. The paper describes the model, gives a concrete example, compares it with common test formats, how it supports progressive context reveal and controlled self healing, and reports early industry experience.

**Keywords:** Agentic Testing · Test Automation · Self-Healing Tests · Runtime Guardrails · Context Engineering

## 1 Introduction

Most automated tests were written with a simple assumption: the test case describes a path, and the automation framework repeats that path. This works when the product flow is stable and the framework is expected to do exactly what the author wrote.

That assumption is weaker in current systems. User interfaces change often. APIs evolve. Authentication moves between services. Test environments are rebuilt. A test that was correct last week can fail today because a button moved or because the validation signal moved from a page to a service response. These failures are not always product defects. Many of them are test drift or environment drift.

Agentic automation tries to handle this kind of change by giving the runner more room to reason. That flexibility is useful, but it also changes what a test case must say. A test case is no longer only a list of steps. It must also explain the purpose of the test, the context in which the test is valid, the limits that the runner must obey, and the evidence required before a pass can be reported. While traditional automation systems had step by step instructions coded to

execute, with clear path to goal, they were fragile. With agentic automation, the runner operates over a much larger action space. This reduces brittleness to product changes, but shifts more responsibility to the test case to define intent, limits, and acceptable evidence.

CIGE is a test case standard optimized for both developers and agents as customers. The name stands for Context, Intent, Guardrails, and Execution. The main idea is to keep the stable meaning of the test separate from the parts that are likely to change. Intent should be stable. Execution may change, but only when the repair still satisfies the intent and respects the guardrails. This paper contributes a four-field CIGE representation, a runtime workflow for staged context reveal and controlled repair, and early industry evidence from applying the model in a codeless agentic testing platform.

## 2 Background and Related Work

CIGE is not a replacement for existing testing practices. It is a layer for a different execution style. Keyword-driven testing separates test cases from reusable action keywords and test data. ISO/IEC/IEEE 29119-5:2024 defines keyword-driven testing and describes artifacts such as test cases, keywords, data, and results across tools [1]. The Page Object Model gives UI automation a stable abstraction over pages and elements; Leotta et al. studied its maintainability benefit in an industrial Selenium WebDriver setting [2]. Behaviour Driven Development, usually written in Given-When-Then form, helps teams express behavior in a readable way. A systematic mapping study describes the field and notes gaps in evidence and metrics [3]. Other approaches start from models, requirements, or parameter coverage. Model-based testing generates tests from behavioral models [4]. Requirements-based test generation derives tests from requirement artifacts [5]. Constrained combinatorial testing focuses on interaction coverage under constraints [6]. These approaches help create tests, but they do not by themselves define how an autonomous runner should reveal context, enforce limits, or repair a failing path. Recent work on large language model agents makes the runtime safety problem clearer. AgentSpec proposes runtime enforcement for safe and reliable agent behavior through customizable policies [8]. Context engineering work also makes a practical point: agents need the right information at the right time, not one large block of mixed context [9]. CIGE applies these ideas at the test case level.

## 3 The CIGE Test Case Model

A CIGE test case has four top-level fields:

```
{ context, intent, guardrails, execution }
```

The fields can be stored in JSON, YAML or a test management system. The storage format is less important than the separation of concerns.

**Context.** Context describes the operating ground for the test. It can include the details of application under test as reusable prompts that apply to multiple tests in different combinations. For tools specifically, we enable the tool search tool in the context to discover tools on demand instead of loading all tool definitions. This enables progressive context disclosure with token optimization. We also implemented another enhancement with a skills file information of each agent associated with each tool, with its description, while detailed tool calling contracts are federated to respective agent contexts, further optimized in multi agent architectures. The tool-search mechanism and skills-file approach are complementary rather than contradictory. The skills file acts as a lightweight capability index for routing intent to the appropriate agent or tool domain, while the tool-search mechanism performs on-demand retrieval of the precise tool contract required for execution. To avoid redundancy, skills files should contain only high-level capability metadata, while detailed tool schema and invocation contracts should remain federated and loaded only at execution time.

**Intent.** Intent captures the reason the test exists. It should state the capability under test, the success criteria, the failure criteria, and the evidence required for a pass. In practice, intent is usually more stable than the path. A button can move, or a validation signal can move from the UI to an API, while the reason for validating the feature remains the same. Every test case is uniquely identified with intent. While Context, Execution, and Guardrails refer to the intent for their evolution over time, intent itself rarely evolves. In addition, intent acts as metadata for the entire test case. To give you an analogy, when there is a large video file, it will be stored in blob storage, while the metadata of the file gets stored in DDB. This enables systems to learn about the properties of the file without loading the large file from blob storage every time the system needs that data. Similarly intent acts as metadata and informs what the test case is about in a large pool of tests, making intent crucial for agents for filtering, grouping, authoring new tests, etc., with optimal context engineering for tests.

**Guardrails (Runtime)** define what agents must not do. They can block destructive actions, prevent the exposure of secrets, require approval before changing a test, or require specific evidence before recording a pass. Guardrails also reduce false positives by stopping shortcuts that make the final state look correct without really validating the feature. While individual agents may define local guardrails, runtime guardrails enforced through CIGE provide a shared control layer for multi-agent architectures while remaining runtime-agnostic.

**Execution.** Execution describes the first path to try and limits the agent's search space. It may include UI steps, API calls, checks, and expected observations. In traditional automation, the steps often become the test. In CIGE, the steps are guidance. They can be repaired when the product changes, as long as the repair remains the same intent.

Table 1 shows a concrete CIGE test case for a workspace streaming flow.

This example shows the main design choice. The intent is not "click these exact elements." The intent is to prove that a user can start a streaming session under a defined context and with defined evidence. The execution path can

**Table 1.** Example CIGE test case for a workspace streaming flow.

| Field | Example content |
|---|---|
| Context | System: workspace streaming service. Environment: staging. Actor: standard enterprise user. Tools: browser, screenshot capture, and network-log reader. Context reveal policy: load login notes first, load diagnostics only after connection failure. |
| Intent | Verify that the user can start a streaming session. A pass requires a connected state, first frame within the expected latency window, no authentication error, no policy error, and evidence from a screenshot plus connection logs. |
| Guardrails | Do not delete resources. Do not change production configuration. Do not expose tokens or credentials in logs. Do not mark pass from UI text alone. Do not update intent without human approval. |
| Execution | Open the client. Sign in with the test account. Start streaming. Wait for connected state. Capture screenshot and connection log. Validate latency and error fields. |

change if the product changes, but the evidence bar and safety rules remain clear.

## 4 Runtime Workflow

CIGE has been useful especially when the agent treats the four fields differently. Figure 1 shows the workflow used in our implementation. When test failures and false positives are classified before repair, the test result analyzer and self healing agents work together to create product defects when there is a defect report, Environment drift may update context, Test drift may update execution, A false positive may add a stricter evidence rule or guardrail. Candidate repairs are replayed in an isolated environment and reviewed with human in the loop mechanisms for all test case update auto generated suggestions.

## 5 Comparison with Existing Formats

Table 2 compares CIGE with common testing formats. The main shift is from step fidelity to goal fidelity. The fidelity of the step asks whether the automation followed the expected path. The fidelity of the goal asks whether the intended behavior was validated safely with enough evidence, even if the path changed. Agentic testing still needs reviewable steps, but these steps should not become more important than the purpose of the test.

## 6 Industry Experience

CIGE was applied in a codeless agentic testing platform used to create, execute, and maintain tests for dynamic enterprise workflows. The test set included UI flows, service dependencies, environment setup, and business-rule validations. The platform stored the four CIGE fields together with execution logs, evidence artifacts, failure categories, repair proposals, and reviewer approvals.

Table 3 reports aggregate operational changes observed from execution logs, repair records, and reviewer-approved updates after CIGE adoption.

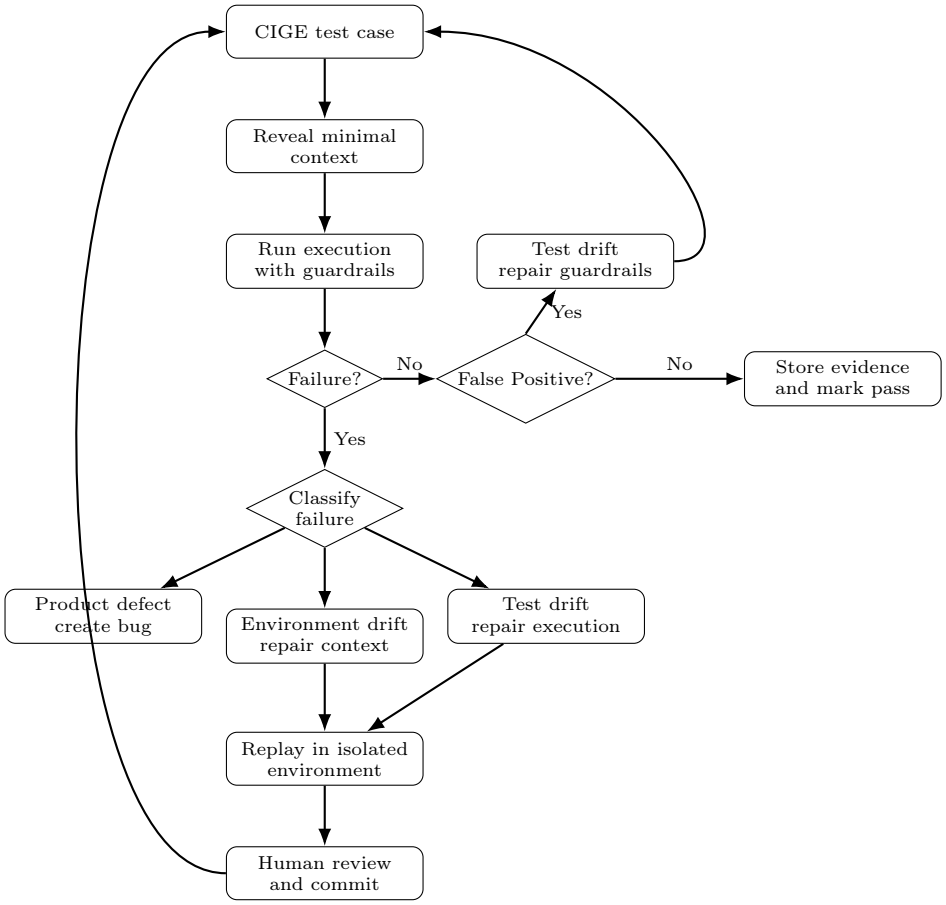


**Fig. 1.** CIGE runtime flow with staged context, guardrail checks, failure classification, and controlled repair.

CIGE is a domain-agnostic architecture designed to address challenges common to software test automation across industries. The following examples illustrate how the framework could be applied in representative domains.

Retail / E-commerce Platforms: Checkout, pricing, promotions, and search evolve continuously, while seasonal release cycles amplify the cost of brittle test suites. Intent (for example, "the user completes checkout with a valid discount and correct tax") remains stable as Execution is repaired to accommodate UI changes. Guardrails naturally prevent unintended business impact—for example, by prohibiting real orders, blocking payment processing, and requiring backend order-service validation rather than relying solely on UI confirmation.

Banking and Financial Services: The controlled-repair model is particularly valuable in regulated environments. Guardrails encode compliance constraints (such as protecting customer data and preserving audit records), while human

**Table 2.** Relationship between CIGE and existing testing formats.

| Format | What it does well | What CIGE adds |
|---|---|---|
| Keyword-driven testing | Reuses actions and data through keywords. | Adds intent, guardrails, evidence rules, and staged context around keyword execution. |
| BDD / Gherkin | Gives stakeholders a readable behavior description. | Separates behavior intent from a repairable execution path. |
| Page Object Model | Hides UI locator changes behind page abstractions. | Can support vision-assisted execution that reduces dependence on brittle locators while preventing repair from changing the test purpose. |
| Model-based testing | Generates tests from a model of behavior. | Adds a runtime structure for agent execution, safety limits, and evidence. |
| Requirements-based generation | Derives tests from requirement artifacts. | Keeps generated tests tied to stable intent and approval boundaries, leverages intent for analyzing existing coverage. |
| Combinatorial testing | Covers parameter combinations under constraints. | Can produce CIGE variants while preserving context, guardrails, and evidence rules. |

**Table 3.** Observed outcomes from the CIGE experience report.

| Metric | Observed change | Practical meaning |
|---|---|---|
| Maintenance effort | About 90% lower | Many changes were limited to execution or context. The original test purpose did not need to be rewritten. |
| False positives | About 38% lower | Evidence rules and guardrails reduced invalid pass paths. |
| Average execution time | About 62% lower | Staged context reduced repeated prompt loading and unnecessary exploration. |
| Self-healed updates | Validated before approval | Candidate repairs were replayed in isolated environments before being accepted. |

approval for Intent changes establishes a clear governance boundary. The resulting evidence and approval history also provide a useful audit trail.

Healthcare: Similar to financial services, Guardrails enforce protections around PHI and prevent unintended modification of patient records. Evidence-based validation ensures that clinical and claims workflows are verified against authoritative backend services rather than UI behavior alone.

## 7 Limitations and Future Work

Although CIGE proposes an early test case structure for the agentic automation era, There are four useful next steps to follow to improve adoption and standardization further. First, migration layers should map BDD scenarios, keyword-driven tests, Page Object based automation, and structured test management records into CIGE. Second, the schema should support multi-agent execution, including planner, executor, validator, and guardrail-monitor roles. Third, CIGE should be evaluated under common sources of drift: selector changes, flaky environments, missing context, API contract changes, unavailable tools, prompt injection, and changing business rules. Fourth, enterprise platforms can use CIGE as a portable artifact between customer workflows, test infrastructure, agent runtimes, and validation tools.

## 8 Conclusion

Agentic automation needs test cases that are more structured than prompts and less brittle than scripts. CIGE is a critical step in that direction charting a smooth transition path from deterministic validations to agentic era verifications leveraging strengths of agents while also mitigating limitations with agentic validations. The experience reported here suggests that this separation of concerns in CIGE can reduce maintenance effort, false positives, and execution time while still giving humans and agents a clear artifact to review and iterate.

**Acknowledgments.** The author thanks the QRS 2026 Industry Track reviewers for feedback that improved the example, comparison, and evaluation discussion.

**Disclosure of Interests.** The author has no competing interests to declare that are relevant to the content of this article.

## References

1. IEEE: ISO/IEC/IEEE International Standard - Software and Systems Engineering - Software Testing - Part 5: Keyword-Driven Testing. IEEE/ISO/IEC 29119-5-2024 (2024). https://doi.org/10.1109/IEEESTD.2024.10806546
2. Leotta, M., Clerissi, D., Ricca, F., Spadaro, C.: Improving test suites maintainability with the Page Object pattern: An industrial case study. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, pp. 108-113. IEEE (2013). https://doi.org/10.1109/ICSTW.2013.19
3. Binamungu, L.P., Maro, S.: Behaviour driven development: A systematic mapping study. Journal of Systems and Software 203, 111749 (2023). https://doi.org/10.1016/j.jss.2023.111749
4. Schieferdecker, I.: Model-based testing. IEEE Software 29(1), 14-18 (2012). https://doi.org/10.1109/MS.2012.13
5. Yang, Z., Huang, R., Cui, C., Niu, N., Towey, D.: Requirements-Based Test Generation: A Comprehensive Survey. ACM Computing Surveys (2025). https://doi.org/10.1145/3771727
6. Wu, H., Nie, C., Petke, J., Jia, Y., Harman, M.: A survey of constrained combinatorial testing. arXiv:1908.02480 (2019)
7. Rathnayake, A., Shahin, M., Abaei, G.: Behaviour Driven Development Scenario Generation with Large Language Models. arXiv:2603.04729 (2026)
8. Wang, H., Poskitt, C.M., Sun, J.: AgentSpec: Customizable Runtime Enforcement for Safe and Reliable LLM Agents. arXiv:2503.18666 (2025)
9. Anthropic: Effective context engineering for AI agents. Anthropic Engineering Blog (2025). `https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents`