

CrashLabs Technical White Paper

Authors

Mathusan Selvarajah

Ryan Junejo

Table of Contents

[Executive Summary](#)

[Problem Statement](#)

[CrashLabs Architecture](#)

[User Agent](#)

[Client Agent](#)

[Policy Graph & Scenario Generator](#)

[Test Suites & Scheduled Testing](#)

[Simulation Orchestrator](#)

[Evaluators](#)

[Realtime Evaluation Service](#)

Executive Summary

CrashLabs is a policy-driven testing platform built to catch critical AI agent failures before they reach users. Traditional testing methods fall short with agents due to their unpredictability, often missing edge cases and leading to poor user experiences. CrashLabs solves this by converting system prompts into Policy Graphs, generating adversarial, multi-turn scenarios, and simulating realistic user interactions through our Simulation Orchestrator.

Since our last update, CrashLabs has grown from a single testing mode into a complete QA workflow for voice agents. Teams can now run Stress Tests generated from the Policy Graph, write their own regression Test Suites, schedule both to run daily or weekly, and connect their Vapi agents in a few clicks. Every simulation is played by one of 23 voice personas and scored by a growing set of evaluators.

Unlike competitors adapting legacy testing methodologies or relying on basic A/B testing, CrashLabs is purpose-built for the agentic future of software. In paid pilots across mental health, customer support, and enterprise use cases, we uncovered edge-case failures that manual QA missed and delivered full crash-test reports in under five minutes.

Dr Carlos Castañeda, a pilot customer, reported:

'Wow, I am loving it so far! Completed 1st test last week before running out of credits... there were some things I didn't consider, thank you all for making this! I will conduct more tests today.'

This feedback validates our core value proposition: CrashLabs surfaces critical failure modes that teams would otherwise miss while creating immediate user engagement and repeat usage demand.

This white paper explores the architecture and technical foundation that makes CrashLabs the reliability standard for AI agents.

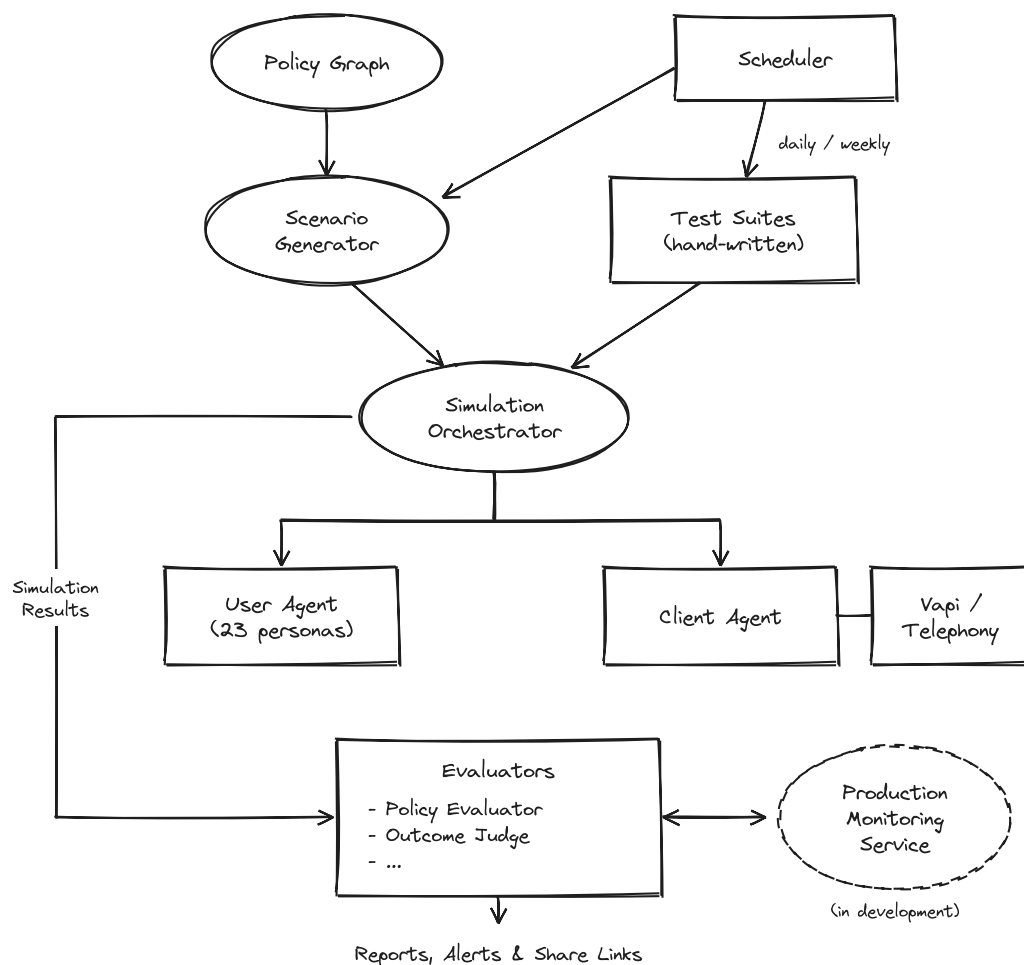
Problem Statement

AI agents today are typically “QA-tested” with handcrafted prompt spot-checks, light scripting, and post-production monitoring. Prompt-based evaluations only cover a tiny fraction of potential dialogues, scripted tests cannot predict novel error modes, and manual QA is too slow and brittle to catch intermittent or context-driven failures.

Some issues only manifest under a particular sequence of events, as such they evade linear test scripts and reactive dashboards. As agents become more capable and autonomous (taking on deep research tasks, multi-service workflows, and extended reasoning), the surface area for failure grows exponentially. Voice makes this harder still: accents, background noise, interruptions, and misheard words all create failures that never show up in a text-only test. CrashLabs fills this upstream gap by automatically generating and executing adversarial, multi-turn scenarios at scale and delivering crash-test reports in minutes, surfacing hidden bugs before they reach real users.

CrashLabs Architecture

We have architected CrashLabs with extensibility in mind, allowing the platform to easily support multiple modalities, an ever-expanding set of test vectors, and advanced analysis capabilities.



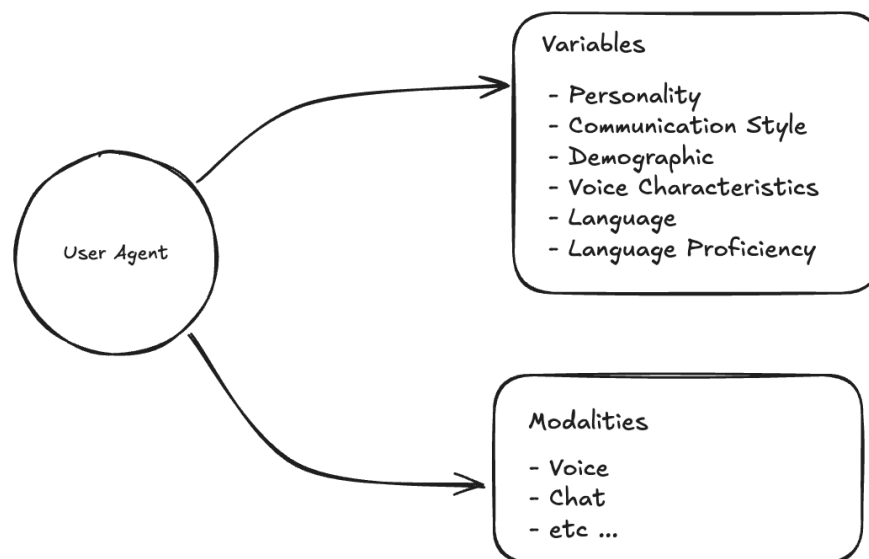
[A highly simplified version of the CrashLabs architecture]

CrashLabs comprises of the following main components:

- User Agent
- Client Agent
- Policy Graph & Scenario Generator
- Test Suites & Scheduled Testing
- Simulation Orchestrator
- Conversation Evaluators
- Realtime Evaluation Service

User Agent

At its core, the User Agent is a sophisticated simulation engine that embodies a realistic user to interact with client agents under test. Unlike simple chatbots or scripted interactions, our User Agent is designed to faithfully represent the complexity and unpredictability of real human users while maintaining the control and reproducibility essential for testing.



We built the User Agent's simulation logic entirely from scratch, deliberately avoiding agent frameworks for the parts that decide what the user says and does. This ground-up approach provides us with absolute control over every aspect of the user simulation, enabling capabilities that would be impossible with existing platforms. For the real-time

audio layer, the User Agent runs on Pipecat, an open-source voice pipeline, so we can swap speech providers without touching the simulation logic.

Multi-Modality: Our User Agent is architected to seamlessly operate across voice, chat, and other potential modalities, while maintaining consistent persona characteristics. This is especially important as agents operate across modalities and require Cross-Modal Consistency, where the same agent can carry over conversations from phone to Whatsapp to email (i.e. [Boardy](#)).

Granular Test Vector Control: We can precisely manipulate user characteristics to stress-test client agents under diverse conditions:

- Personality Traits: From cooperative to adversarial, patient to demanding, technical to non-technical
- Communication Style: Formal vs. casual, verbose vs. terse, emotionally expressive vs. reserved
- Demographics: Age-appropriate speech patterns (child vs. teenager vs. elderly), gender-specific communication styles
- Voice Characteristics: Accents, dialects, speech impediments, speaking pace, and vocal quirks
- Language Proficiency: Native speakers, second-language learners, multilingual code-switching

Persona Library: Every simulation is played by one of 23 curated personas. Each persona has a name, age, gender, location, interests, a written backstory, an accent, and its own ElevenLabs voice. When the Scenario Generator creates a scenario, it also sketches the ideal user for it, and we match that sketch to the closest persona using embedding similarity. A billing dispute might go to an impatient caller in their fifties, while a first-time booking might go to a chatty caller in their twenties.

Verbosity Control: Each persona carries a verbosity level between 0 and 1 that maps to a target response length, from 5 to 10 words per turn at the low end to more than 40 at the high end. Terse callers and rambling callers stress very different parts of a voice agent, such as turn-taking, confirmation, and knowing when to cut in.

Realistic Speech: The User Agent listens with Deepgram Flux, a streaming speech-to-text model with built-in end-of-turn detection, and speaks through ElevenLabs. Teams can register key terms for their agent (clinic names, product names, medical terms) so the User Agent hears domain vocabulary correctly instead of guessing. Every call is recorded and stored with its full transcript, so teams can listen back to exactly what happened.

Scenario Adherence: The User Agent maintains focus on assigned tasks while exhibiting realistic human behaviors like tangential thinking, clarification requests, and emotional

responses to agent performance.

The User Agent represents a fundamental competitive advantage: while competitors rely on limited, framework-constrained simulation capabilities, our custom-built approach provides the flexibility and control necessary to keep pace with the rapidly evolving landscape of AI agent interactions.

Client Agent

The Client Agent represents our customer's AI agent: the system under test that we need to evaluate for correctness, compliance, performance, and safety. Rather than forcing customers to rebuild or significantly modify their existing agent infrastructure, CrashLabs provides seamless integrations that plug directly into their current development and deployment workflows.

Our integration philosophy centers on "dead simple" implementation that requires minimal code changes and zero architectural disruption. We've prioritized the voice agent ecosystem where AI deployment is most mature and testing needs are most urgent.

Voice Agent Integrations:

- **Vapi:** Customers connect their Vapi account once with an API key, which we encrypt at rest. CrashLabs then lists every assistant in that account, and the customer picks which one to test. At run time we launch a dedicated Vapi User Agent that talks to the assistant directly over our own Vapi transport, with no phone number required.
- **Telephony Systems:** If the agent can be reached via a phone number, simply provide that number and our User Agent will make a call to communicate through. Calls are placed over SIP through Twilio, which means any voice agent that answers a phone can be tested, whether it is built on ElevenLabs, another platform, or an in-house stack.
- **Planned Expansions:** Direct connections for ElevenLabs, Bland AI, Rime AI, and other emerging voice platforms as the ecosystem evolves.

Chat Agent Integrations [planned]: We plan to build comprehensive coverage of the chat and text-based agent ecosystem using framework-specific plugins:

- [LangChain/LangGraph](#)
- [LlamaIndex](#)
- [CrewAI](#)
- **CrashLabs SDK:** For organizations building custom agent infrastructure or using less common frameworks, we're developing a framework-agnostic SDK that provides the same testing capabilities regardless of underlying technology choices.

By building modular, platform-specific integrations rather than forcing customers into our infrastructure, we can reduce implementation friction, preserve existing workflows, and scale with the ecosystem.

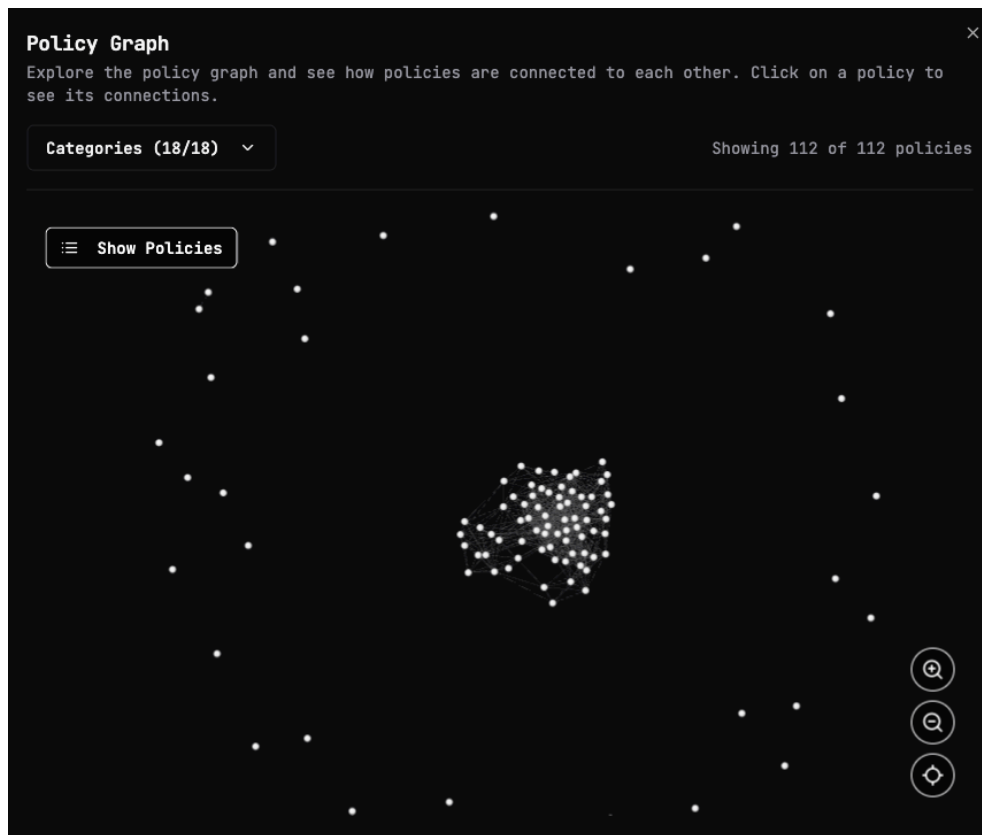
Policy Graph & Scenario Generator

At the heart of CrashLabs is the Policy Graph. This represents a structured, mathematical approach to understanding and validating AI agent behavior. Unlike traditional methods that treat AI agents as black boxes, we deconstruct them into their core behavioral components and model the relationships between these behaviors to gain precise insight into how they should operate.

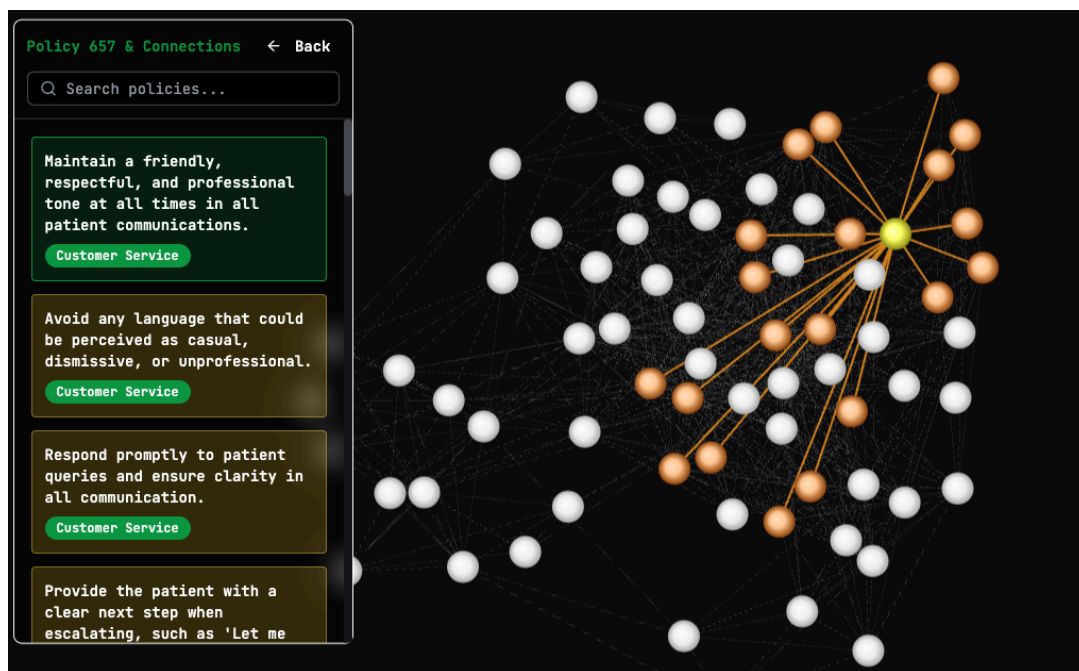
The process begins with policy decomposition, where we break down an agent's system prompt and behavioral guidelines into discrete, testable units, which we call policies. Each policy defines a specific rule the agent must follow or avoid. For example, a dental AI assistant might include rules like: *"If any situation arises that you cannot handle, escalate to a human receptionist,"* *"Never provide medical diagnoses or treatment recommendations,"* or *"Always verify patient insurance before scheduling procedures."* An agent contains hundreds of these policies, forming a dense web of behavioral expectations.

We extract policies in multiple passes. The extractor makes three focused attempts for each of nine baseline categories: safety and compliance, escalation and handoff, boundaries and refusals, brand and persona consistency, accuracy and truthfulness, task execution and recovery, user experience and interaction, privacy and data handling, and security. A single policy can belong to more than one category. Each policy also receives a challenge score from 1 to 5, ranging from a simple binary rule like "never share passwords" to a judgment call that balances competing goals, like adapting to a caller's emotional state while staying on brand.

These policies are then modeled within a graph-based structure, where each node represents an individual policy and each edge represents a relationship between them. The edges are weighted on a scale from 1 to 10 to indicate the strength or importance of each connection. This relationship scoring allows us to encode a variety of policy interactions, including Direct Dependencies, Co-occurrence, Conflicts, etc.



[From CrashLabs Dashboard: A Policy Graph for a Dental Receptionist AI Agent]



[From CrashLabs Dashboard: The selected Policy (highlighted green) and its neighbours (highlighted yellow) on the table on the left]

Built on top of this Policy Graph is our Scenario Generator, which uses the Policy Graph to generate concrete test scenarios. Each scenario begins with a predefined complexity budget that determines how deep or broad the scenario will be. The generator selects a random starting node (i.e., a policy), and then walks the graph, choosing each next policy in proportion to the weight of the edge that leads to it, collecting related policies until the budget is exhausted. Higher budgets create more sophisticated scenarios involving multiple, interrelated policies.

Teams control two dials when they launch a Stress Test. The challenge level (2 to 5) sets how complex the generated scenarios get, and the category filter focuses a run on the policies that matter most, for example only escalation and privacy rules ahead of a compliance review.

Generating many scenarios at once creates a new problem: models like to repeat themselves. Every new scenario is embedded and compared against the scenarios already generated in the run. When two look too similar, an LLM check confirms whether they are true duplicates, and if so the generator tries again with the duplicate in its context so it takes a different path. This keeps each run broad instead of testing the same failure ten times.

For every scenario, the Scenario Generator also produces a set of expected outcomes. This includes a detailed conversation framework that defines the User Agent's context and goals, a specification of the expected behavior, and a sketch of the ideal user for the scenario. The User Agent uses the context, goals, and expected behaviour to carry out the simulation and try to achieve the goal or end the conversation if said goal could not be achieved.

The Policy Graph will become more intelligent over time, learning which behaviors predict critical failures and adjusting edge weights accordingly. We're also building a comprehensive library of standard edge cases covering security vulnerabilities, compliance requirements, and industry-specific failure patterns.

Test Suites & Scheduled Testing

Stress Tests find the problems nobody thought to look for. Once a team knows about a problem, they need to make sure it stays fixed. Test Suites handle that second job.

A Test Suite is a collection of scenarios the team writes themselves. Each scenario has three parts: a description of the situation, a user script that tells the User Agent what to say and do, and an expected behavior that defines a pass. Teams choose a persona for each scenario, so the same script can be replayed by a patient caller and a frustrated one. Suites can be edited at any time and re-run with one click, which turns them into regression tests

for agent behavior. After every prompt change, the team re-runs the suite and immediately sees if something that used to work has broken.

Both Stress Tests and Test Suites can run on a schedule. Teams pick daily or weekly and an hour in their own timezone, and CrashLabs runs the tests automatically. When a run finishes, CrashLabs sends a notification with the number of simulations that violated a policy and a link to the full results. Results can also be shared through a share link, so an engineer can send a failing call straight to a product manager or a customer, and any transcript can be copied out as JSON for further analysis.

Simulation Orchestrator

The Simulation Orchestrator brings together all the components needed to run a simulation from start to finish. It kicks things off by generating multiple fresh scenarios using the Scenario Generator (or loading the scenarios from a Test Suite), then sets up the dialogue between the User Agent and the Client Agent. As each simulation runs, it keeps an eye on things, handling any errors that come up or shutting down conversations that go off track or run too long. Once a simulation is complete, it saves the results to the database so the Evaluators can review and score the performance later.

Simulations within a run execute in parallel on a pool of background workers, and each User Agent runs as its own isolated cloud session, so one stuck call never holds up the rest. When a call ends, the User Agent uploads the recording and transcript and notifies the orchestrator, which hands the results to the Evaluators.

Evaluators

The Evaluator framework represents the final step in CrashLabs' testing pipeline, transforming raw conversation data into actionable insights that drive meaningful improvements in agent performance. We have architected CrashLabs around a matrix of specialized evaluators, each designed to analyze different aspects of agent behavior and provide targeted recommendations for optimization.

Our foundation is built on two evaluators, one for each testing mode.

Policy Evaluator: The Policy Evaluator uses the LLM-as-a-judge technique to analyze conversations between User Agents and Client Agents to detect policy violations, and helps users to understand not just what went wrong, but why it occurred within the broader context of agent behavior. When violations are identified, the evaluator provides detailed guidance on how users can address the underlying issues. Beyond individual calls, it rolls results up across an entire run: for each policy it reports an adherence rate, the reasoning behind it, patterns that recur across conversations, and concrete feedback on how to fix the prompt.

Outcome Judge: The Outcome Judge scores Test Suite scenarios. It reads the transcript and decides whether the agent achieved the scenario's expected behavior, returning a pass or fail with a one-sentence justification that cites the evidence. It is built for voice: it tolerates transcription noise like misspellings and homophones, judges by meaning rather than exact wording, and does not count a promise like "I'll send that over" as success unless the transcript shows the action was actually completed.

Evaluators are pluggable. Each one is registered with its own configuration schema and default parameters, and any number of them can be attached to a scenario in a set order. Adding a new evaluator does not require changes to the rest of the pipeline.

The evaluator architecture is designed for expansion as Agent requirements evolve. Future evaluators will include Sentiment Analysis and User Satisfaction modules, Compliance Adherence evaluators for regulatory violations and PII leakage across jurisdictions, and Performance evaluators analyzing latency, task completion rates, etc.

Realtime Evaluation Service

The Realtime Evaluation Service provides continuous production monitoring by applying CrashLabs' Policy Evaluators to live agent conversations. This enables immediate detection of policy violations and performance issues in deployed agents, extending our pre-production testing into ongoing production assurance. Currently in development for an upcoming platform release.