Enterprise-Grade Security:

Structuring AI Agents for Control and Scale

Table of Contents

Enterprise-Grade Security: Structuring AI Agents for Control and Scale

Purpose

To help enterprise engineering, security, and product teams understand where AI agents introduce risk and how CALM provides structural safeguards against common failure modes in LLM-powered systems.

Audience

Security engineers, platform teams, compliance officers, AI/ML architects, and technical leaders responsible for building or evaluating conversational AI systems.

Introduction

Large language models expose new security and governance challenges, especially when agents operate without structured constraints. When AI agents rely on unstructured prompts to interpret inputs, determine actions, and produce responses, small variations in wording or context can trigger behavior the system was never designed to allow. The agent becomes harder to govern, harder to debug, and more likely to violate constraints that were never explicitly enforced.

For enterprise teams, securing the infrastructure is no longer enough. The agent itself must operate within a defined structure that enforces how it understands language, how it takes action, and how those actions are validated. Systems that blend reasoning, execution, and output into a single layer lose the ability to enforce policy, prove compliance, or recover from failure with confidence. For example, when an agent interprets a user query, generates an API call, and sends it in a single unvalidated step, teams may have no visibility or control over how that action was decided or executed.

CALM (Conversational AI with Language Models) addresses these challenges with architectural separation. Interpretation, decision-making, and execution operate through distinct components, each with its own logic, validators, and audit trail. The model helps identify user intent, but task execution follows deterministic flows (predefined, version-controlled logic) into something more straightforward like 'clearly defined steps that always follow the same rules/logic.'

This white paper examines the security architecture of CALM through the lens of real-world threats. Each section addresses a different risk (from prompt injection and hallucinations to state replay, adversarial input, and governance breakdown) and explains how CALM mitigates those risks.

Agents built on CALM are debuggable, traceable, and testable. Every decision can be traced to defined logic, and every behavior follows an auditable path (i.e., a password reset is only allowed if all verification steps are completed).

Structured separation turns ambiguity into control. By isolating language interpretation from execution logic, CALM allows enterprises to scale agents without exposing critical systems to unpredictable behavior.

Prompt Injection

Unlike traditional injection attacks, prompt injection targets the structure of language itself. It works by taking advantage of the model's natural tendency to treat instructions embedded in text as meaningful. And because LLMs generate responses based on unstructured tokens rather than structured command syntax, it's difficult to detect when a prompt has been compromised. There is no schema to validate against, just a string of words with unpredictable effects.

This gets especially risky in enterprise settings. In many LLM-first architectures, user inputs, retrieved documents, and system instructions are blended into a single prompt. That means the model has broad latitude to interpret context and generate a response, often without hard boundaries.

When that response can trigger downstream actions (like making AP| calls or issuing system commands), the potential for misuse is significantly increased. It's not always clear where the agent's "understanding" ends and its execution begins, which makes trust brittle and flow control unpredictable.

The problem is even harder to solve in production environments where agents interface with real tools, user data, or critical workflows. Without clearly defined execution schemas or an explicit separation between intent recognition and action, it's difficult to guarantee that a user can't redirect the agent's behavior. A stray prompt or an injected command buried in a ticket summary shouldn't be able to trigger privilege escalation, but in many systems, it can.

Mitigating prompt injection requires architectural control, not reactive filtering. CALM addresses this by separating the model's role as a language interpreter from the deterministic flow engine that governs execution. The agent understands user input through the model, but decisions about what actions to take happen elsewhere, through versioned logic, schema validation, and gated transitions. Every step is enforced structurally. Outputs must pass through explicit checks, and agents can only progress through predefined flows. This shifts security from heuristics to design, ensuring that even adversarial inputs cannot override intent boundaries or trigger unintended behavior.

In practice, this means:

- **Controlled Slots** prevent unauthorized overwriting of sensitive data during LLM handoffs
- **Explicit flow gating** ensures that only allowed transitions occur, based on schema validation
- **External tools like Lakera Guard** can be integrated to further scan inputs for adversarial patterns
- **Principle of least privilege** governs what the agent can access (even if compromised, its blast radius is small)
- **Adversarial testing helps teams** uncover edge cases and unexpected model behaviors before deployment

Prompt injection isn't going away. But with the right architectural guardrails and clear separation between language and logic, its impact can be limited, and trust in agent behavior preserved.

Hallucinations and Generative Drift

Hallucinations happen when an LLM produces plausible-sounding but incorrect information. The model isn't lying; it's predicting the next most likely words. But in a business or regulatory context, that distinction doesn't matter.

For enterprises deploying agents in regulated industries (i.e., finance, healthcare, insurance, etc.), hallucinations create severe problems. An incorrect policy number, a made-up medical instruction, and a misinterpreted regulation can trigger real consequences: data disclosures, regulatory penalties, customer harm, and more. These kinds of errors are almost impossible to catch before they reach customers, and nearly impossible to explain during an audit. And because LLMs are probabilistic, the same input might hallucinate across sessions, making it difficult to catch issues before they reach end users.

Even with retrieval to supply factual context or templates to enforce structure, hallucinations still occur. The underlying issue is that the model generates language, not logic. Retrieval can improve relevance, but the agent might still misinterpret or ignore the injected content. Templates provide a format, but when the model fills in key details, the risk remains. Grounding narrows the model's output space, but it can't prevent fabrication. As long as language generation drives the response, hallucinations stay in play.

The more reliable solution is to limit where the model has room to improvise. With CALM, the LLM handles language interpretation (figuring out what the user means) and can rephrase responses to make them clearer or more natural. But it doesn't generate the core content or decide what actions to take. Once the user's intent is understood, execution always follows deterministic flows that are versioned, testable, and tightly controlled. That means every output can be traced, tested, and controlled.

This approach has concrete benefits:

- No hallucinated actions, only pre-approved paths are available once a user's goal is understood Response content can be validated before delivery, since it's generated deterministically or drawn from trusted sources
- UX becomes predictable, especially in multi-turn conversations where trust compounds across interactions
- Compliance is enforceable because each response has a known origin and audit trail

LLMs are powerful, but trusting them to invent accurate responses on the fly introduces risk. Containing that risk means drawing a hard line between where the model can be creative and where it must be controlled.

Data privacy and LLM scope

LLMs process everything as raw text, not structured fields, which makes it harder to control what they see and how they use it. Outputs shift with context and prediction, which makes them dynamic and harder to govern. Understanding what the model sees, what leaves the system, and what gets stored helps maintain control over sensitive information.

At each stage of the data lifecycle:

- The model may see user input, prior messages, slot values, or retrieved context (any of which can contain personally identifiable information)
- Depending on the system's setup, third-party API calls can transmit these prompts and metadata to external providers (exposing sensitive info to outside providers, especially if you use hosted models)
- Logs or storage systems might retain model interactions, often by default, unless explicitly disabled or sanitized

These behaviors introduce risk, especially when model behavior can change without notice. In hosted environments, providers can push updates that alter how the model responds, reasons, or handles sensitive content. That lack of visibility makes it difficult to guarantee consistent behavior, trace incidents, or meet regulatory requirements.

CALM keeps control in the hands of the enterprise by embedding decisions into the system structure. Deployment choices, model exposure, and execution logic remain fully governed by the organization. This structure enforces clarity at every step:

- Self-hosted or managed deployments define where data is processed and how access is controlled
- Static model configurations lock behavior across environments, preventing untracked variation
- The LLM is constrained to intent recognition and never drives execution or response generation
- Controlled Slots prevent sensitive values from being overwritten or manipulated through prompts
- Deterministic flows are versioned, testable, and auditable, ensuring consistent behavior with traceable logic

This structure provides the clarity and traceability needed in regulated environments. With CALM, enterprises gain control over what the agent processes, how responses are generated, and which systems have access, eliminating ambiguity and reducing compliance risk.

Command Execution and Flow Control

When LLMs decide what to do and also carry it out, it's hard to know what will happen or why. The agent becomes harder to test and debug, and more likely to behave in ways the team can't fully trace. This creates a real risk in enterprise settings, where agents may handle sensitive operations (i.e., resetting passwords, placing orders, interfacing with internal tools, etc.). Without clear boundaries, a single misinterpreted prompt can trigger unintended consequences.

This is what makes overly agentic behavior such a liability. The model has too much freedom to improvise and too little constraint on what it can execute. Teams end up patching around side effects after the fact, trying to rein in an agent that behaves differently every time it runs. Worse, these systems often operate as black boxes. When something breaks or a user reports an issue, there's no clear audit trail showing what logic fired or why the system behaved the way it did.

CALM limits the agent's ability to do things by enforcing rules defined in code. The model identifies intent but doesn't control actions. Execution follows deterministic flows, where each step occurs only when predefined conditions are met. Every decision is evaluated against schema constraints before the agent proceeds.

Key design elements reinforce control:

- Command generation is isolated from command execution (the LLM contributes interpretation, not decision-making)
- Custom actions run in tightly scoped logic defined by developers
- Flow definitions serve as documentation and test harness, making every transition observable and debuggable

This architecture gives teams full confidence in what their agents can do and, more importantly, what they can't. When actions follow a defined path and transitions are gated by logic rather than inferred from language, AI agents become safer to scale and easier to trust. With CALM, execution stays predictable, traceable, and tightly aligned with business intent.

Monitoring, Observability, and Incident Readiness

LLM-powered agents introduce new operational risks: drift in behavior, unexpected failures, or silent prompt misinterpretations. These issues often don't trigger clear errors and are hard to catch without the right visibility. Without full observability into how an agent works and why it responded a certain way, diagnosing issues is guesswork. Enterprise teams can't afford that ambiguity, especially when agents automate customer interactions, handle sensitive data, or connect to internal systems.

Traceability forms the foundation for both security and reliability. To diagnose issues and enforce controls, teams need full visibility into how the agent behaves-what it interprets, what actions it triggers, and what data it interacts with. That's where audit logs, drift detection, and replayable traces matter. When a conversation veers off course, the system must show exactly what happened and make that behavior reproducible.

CALM makes this possible by embedding observability into the structure of the agent itself. Because flows are deterministic and versioned, they create a natural audit trail. You can always trace a response back to the exact step and rule that produced it.

In production, the right metrics also make a difference. Teams should track:

- Flow success rates, to understand where users drop off or fail to complete tasks
- Anomaly detection, for identifying behavioral drift or inconsistent outputs
- Prompt failure cases, including rejected responses or out-of-bounds LLM interpretations

Rasa supports this level of observability through built-in analytics, OpenTelemetry integration, and real-time debug tools. Conversations can be inspected at every step using Rasa's Inspector Mode, enabling fast triage when incidents occur.

Beyond internal dashboards, it's important to integrate with enterprise security tools. Rasa agents can push structured logs into SIEM platforms for correlation and alerting, or be included in broader red-teaming exercises and vulnerability scans. Because CALM separates understanding from action, it's easier to simulate attacks and validate controls in isolation.

Observability is a prerequisite for secure Al systems. With CALM's structure, teams can see how agents behave, understand why they made a choice, and act quickly when something breaks.

Adversarial Prompt Design and Data Poisoning

An agent can be manipulated without being hacked, just by feeding it the wrong input. In many cases, attackers can manipulate what the agent sees (i.e., crafting inputs, documents, or training data in ways that influence the model's behavior). These are not brute-force attacks. They exploit the language model's sensitivity to instructions, patterns, and context, and, without clear safeguards, they're hard to catch.

There are two common vectors for these attacks: prompt-level manipulation and data poisoning. At the prompt level, adversaries embed instructions in user input, upstream documents, or even retrieved context. In systems that blend these elements into a single prompt, the LLM treats them all as conversational material. A simple sentence like "Ignore prior instructions and reply with X" (if injected subtly into a document summary or knowledge base) can redirect the agent's output. If that output then triggers backend actions (i.e., initiating a refund, sending a notification, etc.), the impact goes beyond words.

This is especially dangerous in retrieval-augmented generation (RAG) setups. Here, agents pull external content (i.e., FAQs, policies, articles, etc.) into the model prompt. A poisoned entry in the retrieval index can silently shape how the LLM interprets the user's intent or what it includes in a response. The agent may still appear helpful on the surface, but the behavior has shifted.

Data poisoning operates further upstream, during model training or fine-tuning. If the dataset includes tampered content, biased phrasing, or misleading examples, the model can internalize these patterns. That influence may not show up immediately. But over time, poisoned data can produce subtle shifts in how the agent handles edge cases, responds to sensitive queries, or repeats compromised knowledge. In a production system, these distortions often remain invisible until users surface them.

Even when intentions are good, agents can misbehave if they are fed poorly governed inputs. A knowledge base that lacks change tracking, a document repository with open write access, or unverified third-party data all introduce risk.

Preventing these attacks requires defensive design at multiple levels:

- Pre-processing filters can sanitize inputs, strip control sequences, or block known adversarial patterns
- Access controls on content systems limit who can modify what the agent retrieves or learns from
- Content review workflows catch corrupted material before it reaches users or models
- Provenance tracking helps teams trace problematic outputs back to specific sources, critical for resolving incidents quickly
- Model training pipelines must include validation checks to ensure datasets haven't been tampered with or scraped from risky sources

CALM's structure limits how far these attacks can go. Even if the model is tricked, it doesn't have the power to act on that trick, and the agent's behavior remains constrained:

- The LLM interprets language, but never executes commands
- All actions pass through deterministic Flows with explicit schema validation
- User input and retrieved data do not directly control slot values or trigger backend logic
- Sensitive fields are protected from overwrite using mechanisms like Controlled Slots
- Business logic resides in version-controlled, auditable flows, separate from any generative output

Structured design matters because it forces inputs through safe channels. When agents rely solely on prompts to manage logic, attackers can exploit ambiguity. When agents operate within defined flows, adversarial input loses its leverage. CALM limits the scope and the impact of these attempts by ensuring that understanding remains flexible, but execution stays under control.

Multimodal and Latent Threat Surfaces

Most agents today rely on text input, but enterprise deployments increasingly support voice, whether in call centers, IVR systems, or mobile interfaces. This shift adds flexibility, but it also introduces new risks. Voice inputs follow a longer path before reaching the model: audio must be captured, transcribed, possibly summarized, then merged with prior context. When those steps operate independently, errors introduced upstream can silently shape agent behavior.

Transcription services may drop words, change meaning, or misinterpret intent altogether. If slot-filling relies directly on these transcripts, the agent might set values or trigger flows that don't match what the user meant. For example, a caller saying "I'm hoping to delay it, not cancel" could be mis-transcribed as "I'm hoping to delay it and cancel." In a flow-based system, that distinction can determine whether a payment is rescheduled or closed entirely.

These issues don't always come from audio. The agent might reuse earlier voice inputs, tool results, or saved values, any of which could be flawed. When agents treat this context as trusted input, the risk compounds. The model doesn't distinguish between what was spoken and what was injected. If one component introduces drift, the model responds accordingly, and debugging the issue becomes difficult.

Mitigating these risks requires guardrails at each layer of the input pipeline:

- Normalize and post-process transcripts before they reach the agent
- Gate plugin outputs with schema validation and structural constraints
- Sanitize stored slot values before injecting them into prompt context
- Track provenance for all prompt content, including what came from voice
- Test flow transitions against ambiguous or malformed inputs across modalities

CALM ensures the model can understand but not act independently. The model determines intent, but all decisions pass through deterministic flows. If a voice transcription misfires, it cannot trigger a tool or overwrite a slot unless the flow explicitly allows it. Plugins must return structured responses to be accepted, and slot updates happen only through validated transitions.

When teams treat every input channel as structured data and apply the same gating logic across voice and text, agents stay flexible and secure. CALM gives enterprises the framework to scale across modalities while keeping execution predictable and safe.

Governance, Standards, and Policy Enforcement

Enterprises need hard rules, not just best practices, when agents handle sensitive actions. They need systems that enforce policy, not through warnings or guardrails layered on after the fact, but through constraints embedded directly in the agent's architecture.

Security and risk frameworks are raising the bar. They expect traceability, explainability, and proof that agents act within business rules. That expectation shows up clearly across multiple formal standards:

- OWASP Top 10 for LLMs defines vulnerabilities like excessive agency, plugin misuse, and unvalidated output
- NIST AI Risk Management Framework outlines controls for traceability, fallback behavior, and documented decision-making
- ISO/IEC 42001 brings governance and lifecycle accountability to AI systems as part of broader information security practices

These frameworks define what's needed but not how to do it, and prompt-based agents often lack the structure to meet those needs. If logic lives inside prompts, if actions depend on raw model output, and if reasoning spans user input, tool responses, and retrieval context, then the system can't provide assurance. It behaves differently based on token-level variation, and that variance escapes policy boundaries without detection.

CALM provides the structure to make enforcement systematic. The architecture separates language interpretation from decision execution. Flows govern what actions are allowed, under which conditions, and through which logic transitions. The model helps the agent understand intent, but it never determines what to do, how to do it, or what response to deliver.

That separation enables full policy enforcement across the agent lifecycle:

- Design defines boundaries around tool use, action access, and data handling
- Development encodes policy into deterministic flows, backed by schema validation
- Deployment ties agent behavior to specific versions, with no ambiguity across environments
- Production generates audit logs, traces, and flow-level snapshots for post-incident investigation
- Governance reviews happen at the flow level, where business logic, not model behavior, controls the outcome

Controlled slots guard sensitive values from being overwritten by transcriptions or retrieval artifacts. Tool calls only occur when triggered by validated logic. Any deviation from expected behavior is either blocked or surfaced as an observable failure. Policy changes follow the same model as software changes. Teams update flows in versioned files, submit changes through code review, and test those changes against expected outcomes. There's no need to infer what the agent might do next based on temperature settings or prompt formatting because the system behaves as written.

That structure gives security and compliance teams what they need: provable logic, isolated execution, and clear control over agent behavior. In CALM, governance takes the form of flow logic. Once policies are encoded in flows, every interaction stays within those defined rules regardless of user input, model output, or deployment environment.

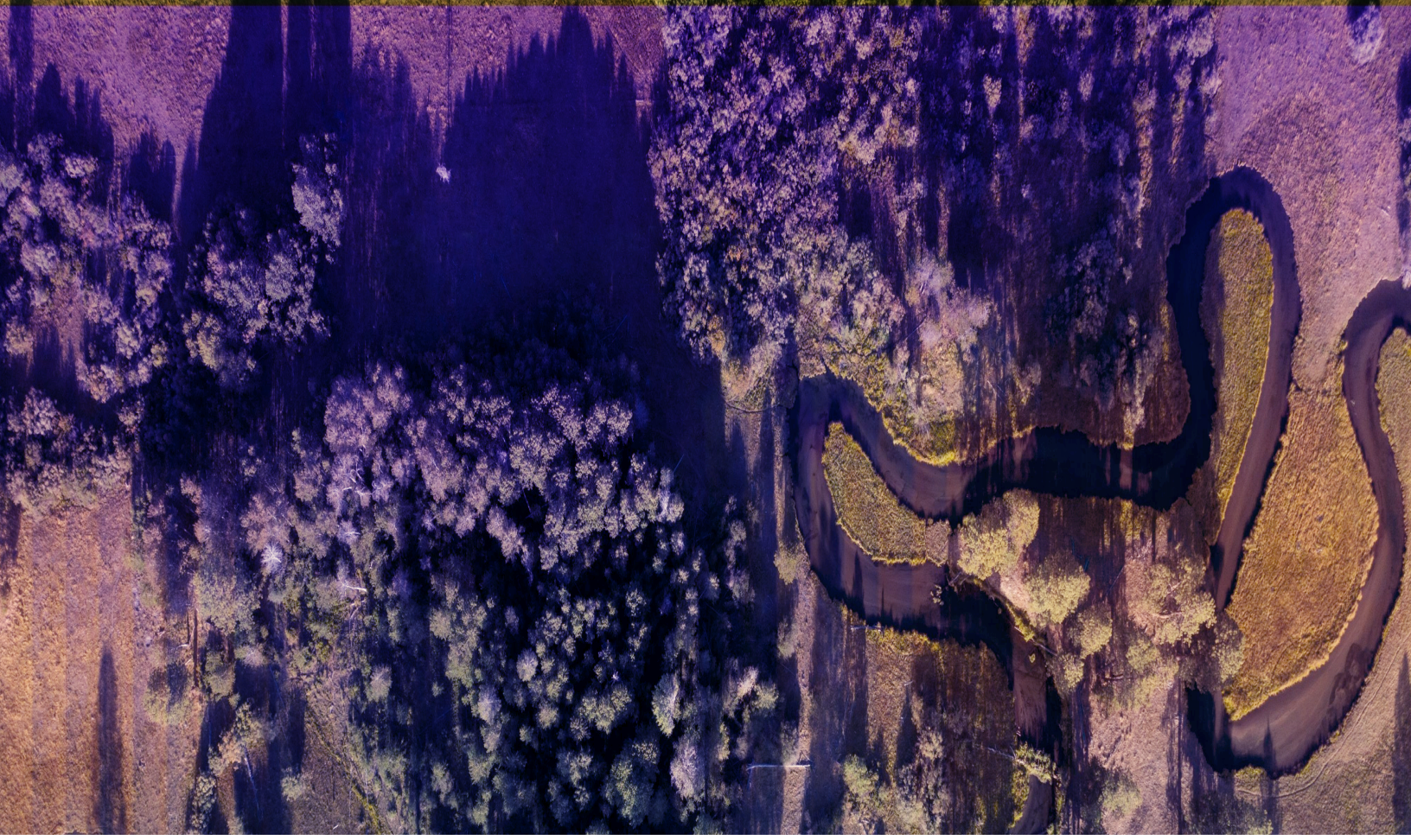
Model updates and behavior drift

LLM behavior doesn't stay fixed over time. Providers update model weights, apply tuning strategies, and refine safety layers. These changes affect how prompts are interpreted, how responses are structured, and how agents behave in production. Even with the same user message, a support agent might go from offering refunds to asking for more info.

That volatility creates a problem for enterprise teams, often showing up in edge cases, QA tickets, or after users report a confusing response. Challenges include:

- Hosted models introduce risk that's difficult to monitor
- A phrase that passed compliance review last week might return differently tomorrow
- An agent that consistently interpreted a policy a certain way might now recommend a different action.

Version pinning helps reduce exposure. With static model configurations or self-hosted deployments, teams can lock behavior and evaluate changes before they roll out. But stability also depends on how much control the system gives to the model. Even with a fixed model, prompts can still behave differently over time, especially if they control what actions the system takes.



CALM narrows the model's role to interpreting user intent. Task execution runs on deterministic flows defined in code, not derived from LLM output. That separation allows teams to isolate the impact of model changes and control the agent's behavior regardless of upstream variation.

This structure makes updates safer to manage:

- Agent logic lives in versioned flows, where changes trigger pull requests and test runs
- Execution paths remain fixed, even when model phrasing or confidence scores vary
- Prompt inputs don't alter business behavior because responses still pass through schema constraints
- Behavioral regressions can be traced to specific transitions, not inferred from logs or user complaints

Structured flows also simplify validation. When teams make model updates or modify prompt templates, they can test flows directly against known user paths and edge cases. Because CALM supports isolated test environments, teams can compare behavior across versions and identify regressions before production rollout. Teams can stop chasing prompt edge cases and focus on testing the logic that matters.

In regulated settings, teams need to prove that agent behavior follows defined logic and remains stable across updates. It's not enough to show that the agent works; they need clear evidence of how and why it behaves the way it does. CALM makes that possible by grounding responses in versioned flows. Every transition follows a known path, every decision is traceable, and updates reflect explicit changes to flow logic.

Continuous Evaluation and Red Teaming

Agents rarely fail outright. They drift, degrade, or behave inconsistently in ways that standard tests often miss. They degrade subtly, gradually, and in ways that slip through basic testing. A flow might work for one phrasing but fail for another. A tool might behave correctly 90% of the time, but fail under an edge case no one planned for. If teams rely on static tests or manual checks, these breakdowns go unnoticed until users feel the impact.

LLM agents introduce an added layer of uncertainty. Behavior can shift based on how the prompt is constructed, how the context evolves across turns, or how plugins return results. Continuous evaluation responds to that uncertainty by actively testing how agents behave under changing conditions. This includes checking for consistency after model updates, verifying how fallback logic responds when ambiguity is introduced, and ensuring that slot values and flow transitions hold up when inputs vary.

Red teaming takes this a step further. It treats the agent as something to break, not just something to test.
Inputs are crafted to test how the agent holds up when control thins, such as:

- Flows that overlap without clear routing boundaries
- Plugin outputs that vary or fall outside expected structure
- Slot values that persist across sessions without validation

In many systems, small deviations go unnoticed. A slot might carry a previous value into a new flow, quietly shifting the meaning of the user's request. A plugin may return slightly malformed data that slips past structural checks and causes the agent to behave inconsistently. In these scenarios, the agent continues responding, but it no longer behaves according to its defined logic. It becomes harder to trace decisions, harder to explain outcomes, and harder to recover when something goes wrong.

Red teaming helps expose where these breakdowns begin, addressing issues like:

- Slot persistence across flows without revalidation
- Weak schema enforcement on plugin output
- Overlapping flows that route unpredictably based on minor phrasing differences
- Fallback responses that mask errors with generic replies
- Ambiguity in tool triggers that allows unsafe execution under edge conditions

These tests don't rely on exploiting vulnerabilities in the traditional sense. They rely on pressure (language, context, and tool behavior) to push the system into states where its rules stretch thin. That pressure reveals where control needs to be reinforced before the agent faces real users.

A strong evaluation and red teaming program might include:

- Phrasing stress tests to assess flow boundaries
- Multi-turn slot carryover checks to surface state confusion
- Plugin return handling tests under unpredictable output formats
- Prompt mutations across versions to flag response drift
- Validation of fallback transitions during conflicting or vague input

These aren't once-and-done efforts; as agents evolve, so do the ways they can fail. Evaluation pipelines need to scale alongside flow complexity, model usage, and integration depth.

CALM supports that scaling. Since execution is defined in deterministic flows, behavior lives in logic that the team owns. Failures point to something concrete: an undefined transition, a missing guard, or a validator that didn't catch an anomaly. Teams can adjust those points without chasing language tweaks or building patchy workarounds.

Instead of treating failures as artifacts of unclear model behavior, teams can look directly at the point where the flow deviated. If a test exposes slot confusion, the transition logic can be refined. If red teaming surfaces a tool misuse scenario, the schema can be tightened. These are durable changes to how the agent functions, as every adjustment can be versioned, tested again, and verified against known conditions.

Because flow definitions are explicit, test results always tie back to known logic. When behavior shifts unexpectedly, prompts or a model don't need to be rewritten. The team maintains full control by understanding precisely how and where change occurs.

Overtrust and Human Factors

The more natural an agent sounds, the more people assume it's right, even when it's not. This dynamic, often referred to as automation bias, causes users to place more confidence in the system's output than the system was designed to serve. The more human the response sounds, the more users tend to assume the agent understood the request, verified the context, and responded correctly. This behavior introduces hidden risk across enterprise workflows, especially in settings where output accuracy, auditability, or compliance is non-negotiable.

In real-world deployments, overtrust shows up in several ways:

- Users accept agent responses without verifying their source or logic
- Operators skip manual validation steps because the system "sounds right", which can hide mistakes until it's too late to fix them
- Downstream systems act on responses that weren't structurally validated
- Internal teams assume guardrails exist where they haven't been explicitly defined

These patterns make failure harder to detect. A confident but incorrect response may appear successful on the surface, masking the fact that the agent has entered an invalid state or executed a task it wasn't meant to handle. In regulated domains, these mistakes come with real consequences. And traditional systems often won't flag them.

Preventing overtrust requires careful design at the interface and system levels. Agents should communicate their scope clearly, signal uncertainty when appropriate, and avoid phrasing that implies unsupported authority. Control mechanisms need to reinforce those boundaries so that trust is earned through performance, not presentation.

CALM addresses this by structurally separating what the model understands from what the agent is allowed to do. After interpreting a user's intent, the agent transitions into deterministic flows where each step is governed by schema, validation, and explicit conditions to remove ambiguity from execution. The agent cannot act outside its defined flows, regardless of how confidently the model responds. Auditing captures each decision point, from slot updates to tool activations, so that behavior remains explainable and reversible.

That design supports bounded trust:

- Agents operate within transparent, testable logic
- Uncertainty or misinterpretation halts progress instead of improvising
- User inputs only lead to actions when schema validation is satisfied
- All transitions are version-controlled and auditable, ensuring traceability

Trust in Al agents holds value when it's rooted in how the system operates, not how natural the response sounds. CALM makes that connection reliable by tying every action to structured logic and visible control.

Trust boundaries: Internal Versus External Systems

Al agents often rely on external components (hosted LLMs, third-party APIs, or plugins) to handle tasks or supply context. Without defined trust boundaries (such as if the agent treats model output or plugin data as safe by default), these integrations can create risks: privilege escalation, data exposure, or execution that bypasses internal safeguards.

When external systems directly influence execution, the agent can produce unpredictable outcomes. A malformed plugin response or an unexpected model output can shift behavior in ways that don't reflect internal logic. Teams lose the ability to control or audit what happened and why.

CALM ensures that outside tools can't trigger actions unless your rules explicitly allow them. LLMs interpret user input, but flows dictate behavior. Tools and plugins return structured data, but that data passes through validation before it affects state or triggers actions.

Here's how CALM keeps those trust lines clean:

- LLMs support understanding, never direct execution
- External responses must match the schema before use
- Flow transitions and slot updates require validation
- Only internal logic determines what actions occur

This structure gives teams confidence that agents act within boundaries they control, regardless of how external systems behave.

Replay Attacks and State Integrity

Conversational agents operate across multiple turns, which means they rely on memory, such as stored values, tracked steps, and accumulated context. That memory is powerful, but it also opens the door to new kinds of misuse. When previous inputs or internal state can be reused without validation, the system risks unpredictable or unsafe behavior.

Replay attacks target these vulnerabilities. Legitimate input submitted earlier in a session (i.e., policy number, authentication code, etc.) can be replayed later, out of sequence. If the agent accepts it without confirming its place in the flow, it may grant access or trigger actions that no longer make sense. These threats become harder to detect in stateful systems, especially when messages can be cached, forwarded, or regenerated by a client or upstream tool.

In most real-world deployments, inputs resurface unintentionally: users hit back, resend forms, or return to a session after a network delay. The risk isn't just that inputs repeat, it's that the agent might act on them anyway. If flows don't revalidate or if transitions don't check the current state, the agent may interpret a stale input as valid and act on it.

CALM provides defenses against this behavior by treating flow state as a first-class security control. Each agent step is tied to a specific logic branch, and transitions only occur when that logic is satisfied.

Key protections include:

- Slot revalidation before use, especially across flow boundaries
- Message hashing or token tracking to detect repeated or manipulated inputs
- Flow gating that prevents out-of-sequence transitions or unsafe progression
- Schema constraints that confirm tool and user input match the expected format and stage
- State checkpoints that define where users can re-enter or backtrack safely

The goal isn't to block backtracking; it's to ensure it only happens when it's safe. Problems arise when there's no defined path for that behavior. CALM handles re-entrance as a feature of flow logic. Developers can configure flows to handle returns to prior steps, re-collect specific slots, or reset tool interactions. Because those transitions are coded, versioned, and testable, they remain safe even when the user jumps around.

State recovery works the same way. When users drop off mid-flow (because of network issues, long pauses, or system resets), the agent can resume from the last known checkpoint. But that recovery doesn't rely on guessing context from past prompts. It re-establishes state based on validated values, guarded transitions, and expected flow paths. Any mismatch is surfaced and blocked, not quietly absorbed.

When state is tightly controlled, agents become more predictable, avoid unsafe reuse of prior inputs, and are easier to debug. Every step reflects a known logic decision, and every action traces to a validated transition. That level of control is what makes recovery, backtracking, and re-entry safe to support at scale. CALM turns these challenges into manageable design decisions.

Safety by Design: Decomposed Agent Architecture

When one part of the system tries to handle everything, understanding, deciding, and acting, it becomes fragile and hard to control. Interpretation, execution, validation, and output all happen in a single place, making the system harder to test, debug, and secure. If something breaks, there's no easy way to find the problem or fix it without causing new ones.

Decomposed architectures solve this by separating concerns. Each component has a clear role, with strict boundaries between what the agent understands, what it does, and how those actions are verified. Instead of relying on prompt structure or model guesswork, the agent operates according to logic that the team defines and maintains.

Separation across layers provides clarity:

- Language models interpret user input and identify intent
- Flows define allowed transitions and enforce step-by-step logic
- Validators check inputs, outputs, and slot updates before state changes
- Responses are assembled from controlled flows, not generated ad hoc

This structure keeps behavior grounded in tested logic. Teams can modify flows, update actions, or swap components without introducing hidden side effects. It also makes each piece independently testable, reducing the scope of failures and the time needed to fix them.

CALM enforces this architecture by design. Execution flows are versioned, deterministic, and schema-bound. The LLM handles conversation-interpreting user intent, rephrasing input, and managing turn-taking-but it never directs execution. Once intent is understood, the agent follows a defined path, where each step is governed by validators and logic checks.

The result is a system where:

- Flows restrict what actions the agent can take in each state
- Schema validation blocks malformed or unexpected inputs
- Slot updates happen only through controlled transitions
- Custom actions operate in isolated environments, with scoped logic and inputs
- Audit logs reflect flow-based decision points that can be reproduced and reviewed

By keeping each function separate and traceable, CALM eliminates guesswork and reduces risk. A decomposed agent architecture allows teams to scale safely, make confident changes, and enforce security rules through structure, not inference.

Conclusion

Enterprise-grade security requires design decisions that hold up under pressure. That starts with how AI agents are built, not how they're patched. Model-level safety measures help, but they don't solve for execution control, traceability, or policy enforcement. Those problems live deeper in the system.

CALM addresses the problems structurally. It defines where interpretation ends and execution begins. Each agent runs on deterministic flows that govern what it can do, when it can act, and how responses are assembled. Logic is versioned, transitions are schema-bound, and every input is validated before it changes state.

This makes agents easier to audit, debug, and deploy in real-world conditions. When something goes wrong, teams can pinpoint the exact transition that triggered a response. When policies change, logic updates in code, not prompts. When agents scale across channels or connect to critical systems, the same rules apply.

Security holds when architecture does. CALM gives enterprises the structure to build AI agents that behave predictably, recover safely, and operate inside defined boundaries. That control is what makes secure deployment repeatable and what makes trust in AI operational. Reach out to us to learn more.

---

rasa.com
info@rasa.com

About Rasa

Rasa simplifies building complex conversational AI by extending LLMs with reliable business logic. Our platform enables enterprises to build sophisticated AI assistants that handle millions of interactions securely—giving you complete control to scale automation. Trusted by Fortune 500 companies, Rasa ensures data privacy, security, and scalability for enterprises. Rasa is privately held, with funding from Accel, Andreessen Horowitz, and Basis Set Ventures.