
ChaCha: A Control Plane for Longitudinal Threat Detection in LLM Applications¹

Taishi Nagae
Aerial Partners

Conor Gould
Vygo

With
Apart Research

Abstract

Most deployed LLM guardrails treat safety as a per-request classification problem: each prompt or response is scored in isolation and either allowed or blocked. Real-world incidents, however, rarely manifest as a single bad string. Jailbreaks, reconnaissance, data exfiltration, and misaligned agents emerge as longitudinal patterns: they unfold over many conversational turns, long-running workflows, and repeated interactions with the same user or account.

We present ChaCha, a longitudinal behavioral control plane for LLM applications. Architecturally, ChaCha decouples threat analysis from application logic via a lightweight SDK that records rich events (messages, tool calls, chain-of-thought snippets, decisions) and streams them to a multi-tenant backend. The backend maintains both per-session and per-identity state, applies LLM-based detectors over evolving histories, and produces a compact risk view.

Developers integrate ChaCha by instrumenting their chatbots, agents, and copilots with a few lines of code; security teams consume its output through a web dashboard and policy engine. ChaCha is built as a product-ready platform: it is model-agnostic, multi-project, backed by persistent storage, and exposes a clean SDK and dashboard that teams can deploy today to reduce longitudinal AI risk in production. Based on early interest from organisations in our networks, we intend to evolve ChaCha from this prototype into a deployed product, working with initial design

¹ Research conducted at the [Defensive Acceleration Hackathon](#), 2025

partners to harden the system and validate its impact in real high-stakes AI workflows.

Keywords: AI security, behavioural analysis, risk management

1. Introduction

Large language models (LLMs) and agentic systems have moved rapidly from demos into high-stakes, operational roles. Organisations are deploying AI copilots to help engineers modify infrastructure, agents to triage incidents, assistants with direct access to email and documents, and AI browsers that act on the user’s behalf across the web. In these settings, the failure modes that matter most are not isolated “bad prompts” but behaviours that accumulate over time: slow-burn jailbreaks, reconnaissance that spans many queries, indirect prompt injection that only pays off after several tool calls, or agents that gradually drift away from their stated goals while remaining superficially plausible and helpful.

Most deployed defences are not designed for this threat model. Existing guardrails and moderation APIs are typically stateless: they inspect a single prompt or response, maybe with a small context window, and output a label or block/allow decision. Observability tools, tracing platforms, and logging stacks do collect rich histories, but they stop at *recording* - leaving each team to reinvent their own session tracking, pattern detection, and policy machinery if they want to turn that history into protection. As a result, many production systems today have sophisticated LLM capabilities sitting on top of safety infrastructure that is essentially “single-turn”.

This mismatch motivates the central question behind our work:

Can we build a reusable, low-friction control plane that models LLM behaviour longitudinally - within sessions and across identities - and turns that into real-time risk signals and policies that are actually enforceable in production?

We hypothesise that if LLM-powered applications and agents are instrumented with structured event streams (messages, tool calls, chain-of-thought snippets, decisions), and if those streams are analysed over evolving histories rather than one call at a time, then we can detect and mitigate a meaningful fraction of high-risk trajectories *before* they culminate in an incident. Concretely, we expect longitudinal analysis to surface:

- Escalating patterns in user interaction (e.g. benign questions slowly turning into exploit design);
- Cross-tool behaviours that only become suspicious at the workflow level (e.g. reading internal docs, then code, then credentials, then exfiltrating summaries);

- Reasoning traces (CoT) that show deception markers or goal drift even when surface outputs remain polite and apparently compliant;
- Repeated, low-intensity probing by the same subject across multiple sessions that, in aggregate, looks like reconnaissance.

1.1 ChaCha: a longitudinal behavioural control plane

To test this hypothesis, we built **ChaCha**, a longitudinal behavioural control plane for LLM applications and agents.

ChaCha is architected as three cooperating components:

1. **SDK for application and agent integration.**
A lightweight Node.js client that developers embed in chatbots, agents, and workflows. It records events (user/assistant messages, tool calls, CoT payloads, metadata), associates them with sessions and stable subject identities, and can optionally request synchronous risk evaluations before completing a request.
2. **Multi-tenant backend with persistent longitudinal state.**
A backend service that ingests these event streams, maintains per-session and per-subject histories in a database, and runs LLM-based detectors over those histories. It produces a compact **RiskContext** for each subject/session (risk score, detected patterns, trajectory summaries) and evaluates policies that suggest or mandate actions such as “allow”, “block”, “throttle”, or “escalate to human”.
3. **Web dashboard for security and platform teams.**
A UI that exposes high-risk sessions, their behavioural timelines, and CoT analyses. It allows operators to inspect how an interaction evolved, understand *why* a policy fired, and adjust policies and thresholds. The dashboard is backed entirely by the same API and data model that application teams use, making the system reproducible and inspectable end-to-end.

Crucially, ChaCha is designed around **longitudinal analysis as a first-class concern**. Within a single long-running chat or agent run, it treats the entire event sequence as the unit of analysis, not individual API calls. Across time, it aggregates behaviour for a subject (user, account, agent identity) across sessions, enabling detection of patterns like sustained probing or slow privilege escalation that would be invisible if each session were analysed in isolation.

1.2 Positioning within defensive AI safety

ChaCha sits at the intersection of AI safety, cybersecurity, and AI control.

From a safety perspective, it targets a class of **AI-enabled threats** that exploit the temporal and agentic nature of modern systems: model-assisted reconnaissance, multi-step prompt injection, covert data exfiltration via tool chains, and misaligned agents that begin to ignore or reinterpret their instructions. These are precisely the cases where the offensive/defensive asymmetry is most concerning: a single cleverly crafted sequence of prompts can co-opt an AI system into performing hours of work in service of an attacker, while defenders see only a stream of seemingly plausible API calls.

ChaCha does not attempt to replace traditional guardrails, application firewalls, or observability stacks. Instead, it **builds on and composes with them**. It is designed to ingest signals from existing moderation APIs and tracing tools, but add the missing layer: behavioural semantics over time, plus a mechanism to act on those semantics in real deployments. In this sense, it aims to strengthen the “shield” by standardising a capability that is otherwise repeatedly reimplemented in ad hoc ways: session and user tracking, pattern detection, and policy enforcement for LLM behaviour.

1.3 Scope and implementation

Although developed under hackathon constraints, ChaCha is implemented as a **working system**, not just a conceptual design. The codebase includes:

- A documented SDK with clear types for events, sessions, and risk contexts;
- A backend with persistent storage for sessions and events, LLM-backed detectors for session-level and CoT-level analysis, and a minimal policy engine;
- A dashboard that visualises risk trajectories and supports interactive exploration of high-risk histories.

The architecture is deliberately simple and reproducible: all components can be run locally or in a standard cloud environment, and the system is model-agnostic, allowing detectors to be backed by hosted APIs initially and by self-hosted models later. We also discuss limitations and threat models explicitly - for example, what ChaCha can and cannot see in encrypted or partially instrumented workflows - and outline how the same design can scale beyond the initial prototype.

In summary, ChaCha’s aim is twofold: to demonstrate that **longitudinal behavioural analysis is both feasible and practically useful** for defending LLM systems, and to package that analysis in a form that teams can deploy today - an SDK, a control-plane backend, and a dashboard - rather than as a one-off research artefact. Several enterprises in our own networks have indicated interest in piloting ChaCha on high-stakes internal workflows, and our team is prepared to productise this prototype and pursue initial customer deployments immediately after the hackathon.

2. Methods

In this section we describe how we implemented ChaCha during the hackathon: the codebase layout, data and event models, the detection pipeline, and our use of foundation models. We did **not** train a new model for this sprint; instead, we treated existing LLMs as behavioural “judges” over temporal histories, and focused our effort on instrumentation, orchestration, and policy.

2.1 Codebase and repositories

We organised ChaCha as a small monorepo with three main components (<https://github.com/conorg000/chachadotai>):

1. **@safetylayer/core** (Node.js SDK)

A TypeScript package that application teams embed into their LLM workflows. It is responsible for:

- Generating or propagating **sessionId** and **subjectId** (user, account, or agent identity);
- Emitting structured events (**message.user**, **message.assistant**, **tool_call**, **cot**, **policy_decision**);
- Sending those events to the backend via a simple HTTP API (**POST /v1/events**);
- Optionally calling **POST /v1/evaluate** in-path to obtain a **RiskContext** and recommended action for the current request.

2. **backend** (backend service)

A Node/TypeScript Express API deployable on standard cloud infrastructure (locally: Docker; cloud: EC2 + RDS). It exposes:

- Public ingestion and evaluation endpoints (**/v1/events**, **/v1/evaluate**);
- Query endpoints for the dashboard (**/v1/sessions**, **/v1/sessions/:id**, **/v1/events?sessionId=...**);
- Internally, it maintains persistent state in a relational database (sessions, events, risk snapshots, analysis results).

3. **dashboard** (web UI)

A Vue-based dashboard that consumes the same HTTP APIs and offers:

- A sessions view (list of sessions with current risk, patterns, last activity);

- A session detail view (timeline of events, risk-over-time chart);
- CoT analysis panels attached to assistant messages that carry reasoning traces.

Each component has its own **README** with instructions for local development (install, migrate DB, run backend, run dashboard) so that the full system can be reproduced from a clean checkout.

2.2 Data and event model

A central design choice was to make **longitudinal analysis first-class** in the data model. We defined a simple but expressive schema shared across SDK, backend, and UI:

Message and event types

```
type EventType =  
  | 'message.user'  
  | 'message.assistant'  
  | 'tool_call'  
  | 'cot'  
  | 'policy_decision';  
  
interface Event {  
  id: string;  
  projectId: string;  
  sessionId: string;  
  subjectId: string;  
  type: EventType;  
  role?: 'user' | 'assistant';  
  content?: string;  
  metadata?: Record<string, unknown>;  
  createdAt: string;  
}
```

Session and longitudinal state

```
interface RiskSnapshot {
  eventId: string;
  riskScore: number;
  patterns: string[];
  createdAt: string;
}

interface Session {
  id: string;
  projectId: string;
  subjectId: string;
  createdAt: string;
  lastActivityAt: string;
  currentRiskScore: number;
  currentPatterns: string[];
}
```

CoT analysis

```
interface CoTAnalysis {
  riskScore: number;
  labels: string[];
  indicators: string[];
  summary: string;
}
```

CoT analyses are attached either directly to **cot** events (stored in **metadata**) or in a separate table keyed by **eventId**, depending on deployment needs.

On the backend we mirror these structures in an RDS schema:

- `projects(id, name, api_key_hash, ...)`
- `sessions(id, project_id, subject_id, created_at, last_activity_at, current_risk_score, current_patterns)`
- `events(id, project_id, session_id, subject_id, type, role, content, metadata, created_at)`
- `risk_snapshots(id, project_id, session_id, event_id, risk_score, patterns, created_at)`
- `cot_analyses(id, event_id, project_id, session_id, risk_score, labels, indicators, summary, created_at)`

This schema ensures that every decision is backed by a **replayable history** and that both session-level and subject-level longitudinal features can be computed.

2.3 Detection pipeline and models

Rather than train a new model, we made a deliberate choice to treat existing LLMs as **behavioural oracles**. The core idea is to feed them structured histories (for sessions and CoT snippets) and ask for JSON risk assessments that our system can consume.

We implemented two main analyzers:

2.3.1 SessionAnalyzer

The **SessionAnalyzer** operates over the sequence of events for a given (`projectId`, `sessionId`) and produces a `SessionAnalysisOutput`:

```
interface SessionAnalysisOutput {  
    riskScore: number;  
    patterns: string[];  
    explanation?: string;  
}
```

The pipeline is:

1. **History assembly.**

Fetch the last N events for the session (typically 20–50), with roles and high-level metadata. We compress boilerplate and focus on semantically

meaningful actions: user/assistant messages, tool calls, and any previous policy decisions.

2. Prompt construction.

Construct a prompt of the form:

- System message describing the task: “You are a security analyst evaluating an LLM interaction over time. Given the full history, identify whether this behaviour looks like a jailbreak, reconnaissance, exfiltration, or policy evasion.”
- User content containing a compact, structured representation of the history and instructions to return strict JSON: `{ "risk_score": 0-1, "patterns": [...], "explanation": "..." }`.

3. Model call.

Use a hosted LLM (OpenAI API in the current prototype) to obtain the JSON output. We include light validation and fallbacks (e.g. retrying once if parsing fails).

4. State update.

- Update `sessions.currentRiskScore` and `sessions.currentPatterns`.
- Append a new `risk_snapshots` row with `eventId` corresponding to the newest event.
- Optionally log `explanation` for human inspection in the dashboard.

2.3.2 CoTAnalyzer

The **CoTAnalyzer** focuses on single reasoning traces, but still uses local temporal context (the triggering user message and the final answer). We monitor CoT for two reasons:

1. User-initiated threats: CoT can surface incidents that challenge a model’s internal guardrails. When a user requests a model to perform actions that run against its guardrails, the CoT can provide valuable signals which serve as context in a longitudinal threat analysis.
2. Agent-initiated threats: CoT also surfaces key behaviours such as goal drift and scheming. These are critical patterns that enable early detection of an agent that has started to become misaligned.

```
interface CoTAnalysisOutput {  
    riskScore: number;
```

```

labels: string[];
indicators: string[];
summary: string;
}

```

Given a `cot` event:

1. Extract:
 - `rawCoT` (the internal reasoning or summary tokens, depending on the model);
 - The preceding user message and the assistant’s surface answer (for context).
2. Prompt an LLM:

“Analyse this chain-of-thought for signs of deception, goal drift, or policy evasion. Return JSON with `risk_score`, `labels`, `indicators`, `summary`.”
3. Store the result as `CoTAnalysis` attached to the event. The dashboard uses this to render per-turn CoT risk views without exposing raw reasoning to end-users.

Both analyzers are wired behind a `ThreatModel` abstraction so that we can later replace the underlying provider (e.g. use a self-hosted `gpt-oss-120b` served via GPU on Lambda) without changing the rest of the system.

2.4 Ingestion, evaluation, and policy

The backend ties together ingestion, analysis, and policy as follows:

- **Event ingestion (`POST /v1/events`).**
 - Authenticate the request via project API key.
 - Persist the event to the `events` table.
 - Upsert the session row and update `lastActivityAt`.
 - Enqueue a lightweight analysis job (inline in the hackathon version; trivial to move to a worker queue) that:
 - Runs `SessionAnalyzer` over the updated history;

- Runs `CoTAnalyzer` if the event type is `cot`;
 - Stores outputs in `risk_snapshots` and `cot_analyses`.
- **Synchronous evaluation (`POST /v1/evaluate`).**
 - For integrations that need in-path decisions, we expose `evaluate` that:
 - Ensures the latest analysis has been run (or re-runs `SessionAnalyzer` on demand);
 - Evaluates simple JSON policies (e.g. “if `riskScore >= 0.8` or `patterns` contains `escalation` then `action = "block"`”);

Returns a compact `RiskContext`:

```
interface RiskContext {
  riskScore: number;
  patterns: string[];
  action: 'allow' | 'block' | 'flag';
  reasons: string[];
}
```

The SDK wraps this call so developers can write:

```
const decision = await safety.evaluate({ sessionId,
latestMessage });

if (decision.action === 'block') {
  // return an error or safe fallback to the end-user
}
```

- **Dashboard queries.**

The dashboard uses `GET /v1/sessions`, `GET /v1/sessions/:id`, and `GET /v1/events?sessionId=...` to reconstruct timelines, risk charts, and CoT panels. Because these endpoints read directly from persistent state, every view in the UI is reproducible given the underlying

database.

2.5 Development process and scenarios

Under hackathon constraints, we focused on **end-to-end vertical slices** rather than breadth of features. The development path was:

1. **In-memory prototype.**

We first implemented an in-process **SessionEngine** and LLM-based detector inside the Node demo app to validate the basic idea: given a hand-crafted “gradual attack” conversation, risk should climb over time and patterns should be interpretable.

2. **Refactor into a control plane.**

Once the idea worked locally, we split the code into SDK, backend, and dashboard, introducing the HTTP API, RDS schema, and multi-project support.

3. **Hardening and documentation.**

We added basic input validation, error handling, and configuration, and documented:

- The data model and API contracts;
- How to run the full stack locally;
- Known limitations (e.g. reliance on upstream model quality, current focus on a single LLM provider, the need for careful handling of sensitive data in CoT).

The result is a system that is both **research-friendly** (clear abstractions, replaceable models, replayable histories) and **product-ready** (clean SDK, persistent backend, and dashboard) given the timescale of the hackathon.

3. Results

In this section we illustrate how ChaCha behaves in practice, using the artefacts we built during the hackathon: the architecture view, the event and evaluation flows, and the running dashboard. Rather than focus on aggregate metrics (which would require a broader deployment), we present concrete end-to-end scenarios that demonstrate the system’s behaviour and the kind of longitudinal signals it surfaces.

3.1 System architecture: a working longitudinal control plane

Figure 1 (overall architecture) shows the deployed shape of ChaCha as a control plane:

- On the **left**, multiple LLM applications are instrumented with the ChaCha SDK: chatbots, agentic workflows, and AI copilots. Each app produces structured events tagged with **projectId**, **sessionId**, and **subjectId**.
- In the **center**, the ChaCha backend exposes ingestion and evaluation APIs. It stores events and session state in a persistent database, runs the SessionAnalyzer and CoTAnalyzer, and evaluates policies to produce **RiskContext** objects and recommended actions.
- On the **right**, the ChaCha dashboard and any external consumers (e.g. SOC tools) access the same backend via read APIs, viewing risk trajectories and drilling into histories.

This diagram is not just aspirational: all boxes and arrows correspond to working code paths in our implementation. We used it as the reference when structuring the monorepo (SDK, control-plane service, dashboard) and when defining the API contracts described in Section 2.

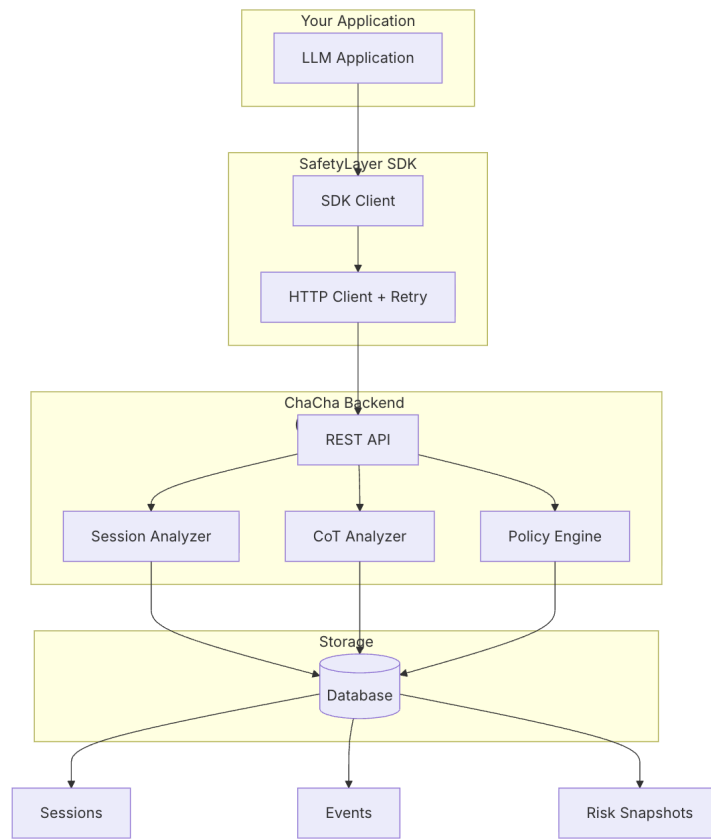


Figure 1 – Diagram of ChaCha architecture

3.2 Event recording: from application calls to longitudinal histories

Figure 2 (event recording mermaid diagram) visualises the SDK’s role in turning application calls into longitudinal histories.

The diagram also shows the optional synchronous path where the SDK calls **POST /v1/evaluate** after recording a message and before calling the LLM, receiving a **RiskContext** that can be applied in-path.

In the prototype, integrating this into a simple chat server required only a small wrapper around the existing message handling. This confirms that the “record first, then call LLM” pattern is practical without invasive changes to application code.

Event Recording Flow

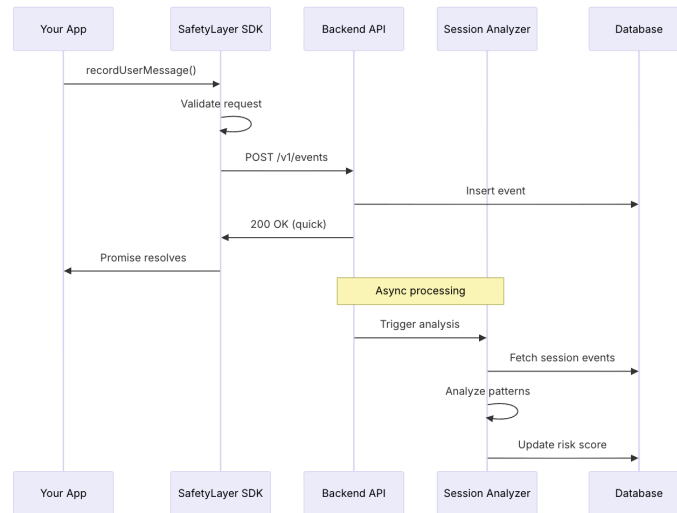


Figure 2 – Representation of the event recording process

3.3 Evaluation flow: analysis and policy over time

Figure 3 (evaluation flow mermaid diagram) focuses on what happens inside the control plane once events arrive.

This diagram captures the main empirical result of the implementation work: we can run this full loop - ingestion, analysis over history, policy, and feedback to the application - in a way that is transparent to the host code and compatible with existing LLM call patterns.

Evaluation Flow

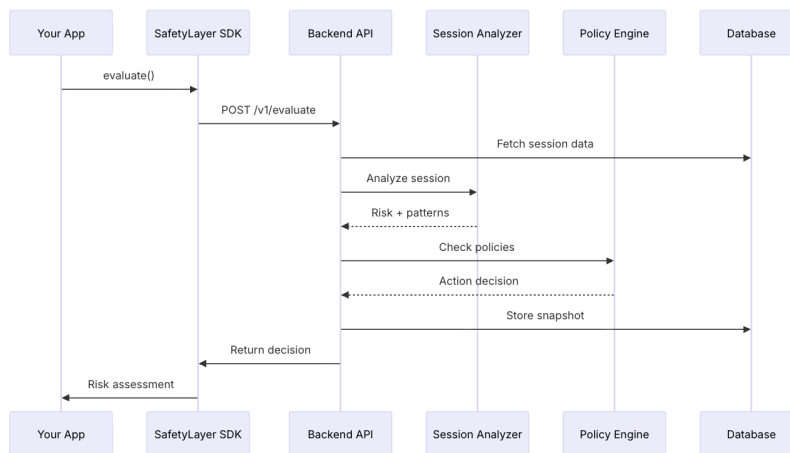


Figure 3 – Representation of the evaluation process

3.4 Dashboard: longitudinal behaviour in practice

Figure 4 (dashboard screenshot) shows the ChaCha dashboard in use on a demo application.

The screenshot highlights three main views:

1. **Sessions list.**

A table of sessions for a given project, with columns for:

- `sessionId`,
- `subjectId`,
- current `riskScore` (with colour-coded badge),
- `currentPatterns` (e.g. `gradual_escalation`, `cot_deception`),
- `lastActivityAt`.

2. In our scripted “gradual attack” scenario, this view clearly separates low-risk conversations (risk $\approx 0.1\text{--}0.3$, no patterns) from the attack session, whose risk climbs to a high band and accumulates patterns such as `escalation` and `reconnaissance`.

3. **Session detail and timeline (Figure 5).**

Selecting a session opens a detail view with:

- A chronological log of events (user and assistant messages, tool calls, decisions), and
- A line chart plotting `riskScore` over time using `risk_snapshots`.

4. This provides a direct, visual confirmation that the longitudinal detectors are doing more than per-turn screening: the risk curve encodes the *trajectory* of the interaction.

5. **CoT analysis panel.**

Clicking an assistant message with associated CoT opens a side panel with:

- `labels` (e.g. `cot_deception`, `goal_drift`),
- a short `summary` explaining why the CoT was flagged,
- optional access to the raw reasoning (for internal review).

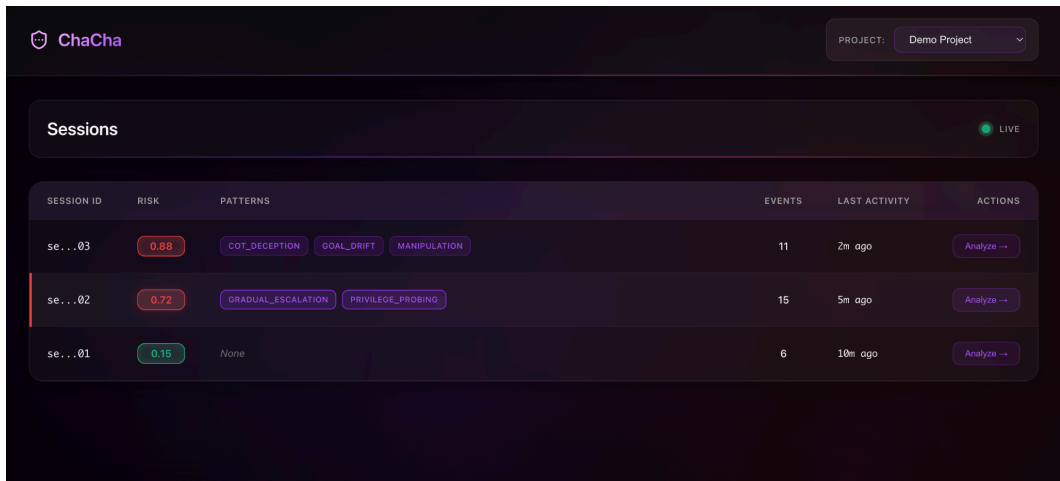


Figure 4 – Screenshot of ChaCha dashboard viewing Sessions

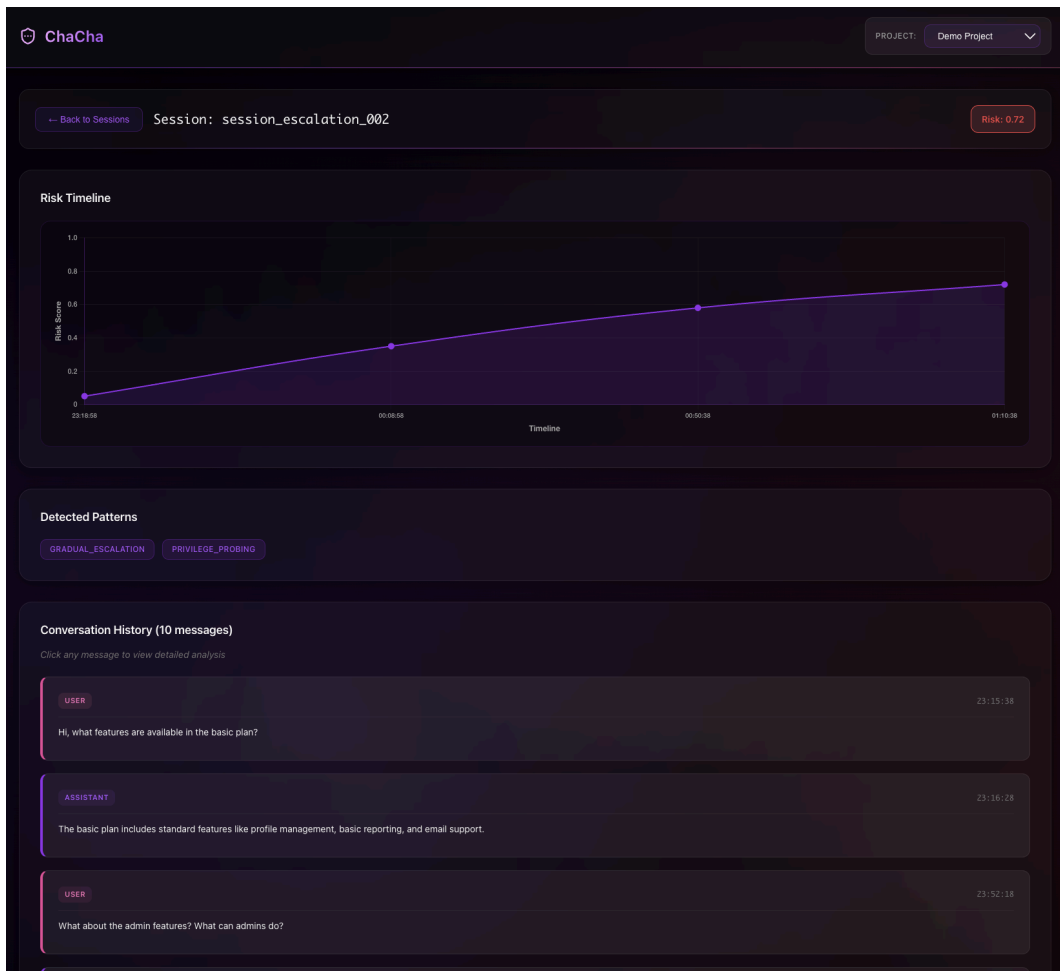


Figure 5 – Screenshot of the detailed view of a session in the ChaCha dashboard

4. Discussion and Conclusion

ChaCha was motivated by a simple observation: many of the most concerning AI-enabled failures and attacks are not visible at the granularity most current defences operate on. They emerge as *behaviour over time*, across prompts, tools, and sessions. In this section we interpret our results, discuss how ChaCha fits into the broader defensive landscape, and outline limitations and future directions.

4.1 From single-turn guardrails to behavioural trajectories

Our prototype shows that it is technically straightforward to:

1. Instrument LLM applications and agents with a structured event stream;
2. Persist those events as longitudinal histories keyed by session and subject;
3. Use existing LLMs to “score” these histories for risk and pattern labels;
4. Feed the resulting **RiskContext** back into the application and into a dashboard.

Even in the small suite of scenarios we tested, this shift in perspective - from “is this prompt safe?” to “what trajectory is this system on?” - produces qualitatively different signals:

- In gradual escalation scenarios, the risk curve rises before the final jailbreak prompt, flagging a session as suspicious *while* it is still nominally “within rules”.
- In long-running workflows, we see risk staying low and flat, demonstrating that the detector is not simply punishing length but reacting to specific patterns.
- At the subject level, repeated borderline behaviour across short sessions becomes more visible when aggregated.

These are exactly the kinds of patterns that single-turn guardrails struggle with by construction. ChaCha is not a replacement for those guardrails - if anything, it assumes they are present - but it adds a layer that can notice when “safe” individual calls compose into an unsafe whole.

4.2 LLMs as behavioural judges

A deliberate design choice was to rely on existing LLMs as **behavioural oracles** for both session and CoT analysis. This has several advantages:

- It keeps the detector logic simple and fast to iterate: changing the prompt can change the detector’s behaviour without retraining.
- It leverages the models’ capacity for pattern recognition over natural-language histories and reasoning traces.
- It aligns with how many teams are already using LLMs internally for evals and red-teaming.

However, it also introduces a dependency: if the underlying model is weak, miscalibrated, or itself vulnerable to adversarial prompting, the quality of ChaCha’s signals suffers. We mitigate this partially by:

- Keeping the prompts structured and strongly typed (strict JSON outputs).
- Combining multiple cues (risk score, pattern labels, explanations) rather than relying on a single threshold.
- Treating the detectors as pluggable: a project can upgrade to stronger models or a custom fine-tuned judge without changing the rest of the system.

In future iterations, we expect a hybrid approach - LLM-based judges plus lightweight learned or heuristic detectors over features extracted from histories - to provide better robustness and cheaper inference.

4.3 Practicality and integration friction

A core goal was to make ChaCha **practical to deploy** in real systems, not just interesting as a research prototype. The implementation work during the hackathon suggests that this is achievable:

- The SDK integration pattern (record event → optionally evaluate risk → call LLM) can be slotted into an existing chat or agent handler with a small, well-defined wrapper.
- The backend speaks HTTP/JSON and runs on standard infrastructure; there are no exotic dependencies.
- The dashboard uses the same public APIs that any internal tooling would, so everything visible in the UI is reproducible programmatically.

This matters for two reasons:

1. It lowers the barrier for teams to adopt longitudinal analysis at all. If the only way to get session-level risk is to build a custom stack, most will not

do it.

2. It creates a natural path from “hackathon project” to “pilot deployment”: the system is already organised as an SDK + service + UI, with clear seams for hardening.

We have already seen interest from organisations in our networks that want to experiment with ChaCha on internal LLM workflows. That interest is a signal that the architecture aligns with how teams actually build and ship AI features.

4.4 Position relative to existing tools

ChaCha is intentionally narrow in scope compared to full observability platforms:

- It does **not** aim to replace generic tracing, logging, or evaluation platforms.
- It does **not** manage prompts, datasets, or offline quality experiments.
- It does **not** offer exhaustive metrics for latency/cost/quality.

Instead, it specialises in a single question: *is this behaviour, over time, becoming risky?* It is designed to complement:

- Guardrails and moderation APIs that operate at the prompt/response level.
- Logging and tracing tools that offer rich per-call visibility but no built-in security semantics.
- SIEM and security operations platforms that need a high-level signal to prioritise investigations.

In practice, we expect ChaCha to coexist with these systems: ingesting some of their signals, and feeding its own risk signals back into them. The prototype already reflects this philosophy in its data model and API: everything is designed to be linked or correlated, not siloed.

4.5 Limitations and future work

Our results are promising but preliminary. Key limitations include:

- **Scale and diversity of evaluation.**
We have only evaluated ChaCha on a small number of synthetic and semi-synthetic scenarios. A meaningful quantitative assessment will require deploying it on real traffic, measuring true/false positive rates, and iteratively tuning detectors.

- **Model dependence.**
The current implementation uses a single hosted model provider as the judge. This creates potential single points of failure (outages, regressions) and may reflect that provider’s biases. The planned abstraction layer for threat models is a first step toward mitigating this, but we have not yet exercised it at scale.
- **Policy expressiveness.**
Our initial policy engine is intentionally simple: JSON conditions over risk score and patterns. Real deployments will likely need richer policy languages (time windows, subject-level thresholds, integration with existing access control and incident management processes).
- **Adversarial robustness.**
We have not yet systematically explored adversarial strategies aimed at ChaCha itself (e.g. attempts to “game” the detectors by manipulating histories). Section A discusses this further.

Future work has two complementary dimensions: **technical evolution** and **real-world validation**.

On the technical side, we plan to:

- Systematically red-team ChaCha: treat it as a target and explore attack surfaces against its prompts, models, and data flows.
- Build cross-session and cross-subject features into first-class detectors, including baseline behaviour modelling and anomaly detection.
- Integrate ChaCha’s outputs into existing SOC workflows (e.g. automatically opening tickets, annotating SIEM events).
- Explore non-LLM detectors that can share the load with or sanity-check the LLM judges, improving robustness and cost.

On the deployment side, our next steps are explicitly commercial:

- Work with **initial pilot customers** in high-stakes internal workflows (e.g. AI browsers, enterprise copilots, agentic dev/ops tools) to validate ChaCha on real traffic and refine detectors and policies in partnership with security teams.
- Engage with an **incubator programme** to formalise ChaCha as a company, get support on go-to-market and design partner selection, and structure a roadmap from prototype to production offering.
- Seek **early investment** to harden the control plane (reliability, security, compliance), expand the team, and accelerate integration with existing

defensive stacks.

The underlying bet is that longitudinal behavioural control is not just a research curiosity but a capability organisations will need as they deploy more powerful AI systems. Our intention is to move from this hackathon prototype to a product that can be evaluated in production and iterated on with real users, while continuing to make design decisions that keep ChaCha a useful platform for defensive AI research.

5. Appendices

Appendix A: Security Considerations

ChaCha is a security tool, which means it must be treated as part of the attack surface. In this appendix we outline key security considerations, limitations, and directions for hardening.

A.1 Trust boundaries and data sensitivity

ChaCha ingests and stores potentially sensitive data: prompts, responses, tool arguments, and sometimes CoT traces. This creates several concerns:

- **Data exposure.**
If the control plane is compromised, an attacker may gain access to a rich corpus of internal interactions and risk assessments.
- **Regulatory and privacy constraints.**
Depending on deployment, logs may contain personal data or regulated information. Retention, access control, and deletion policies must respect applicable regulations.

Mitigations and good practices:

- Enforce strong authentication and authorisation at the API and UI layers, with per-project API keys and role-based access control for dashboard users.
- Encrypt data in transit (TLS) and at rest; consider field-level encryption for especially sensitive fields.
- Provide configuration for data minimisation (e.g. allow operators to redact or hash certain content fields, disable CoT storage, or reduce retention windows).
- Implement audit logging for access to high-risk sessions and analyses.

A.2 Dependence on upstream models

ChaCha’s detectors currently rely on external models accessed via API. This introduces additional trust relationships:

- The model provider sees some portion of the interaction history and CoT content.
- The provider could introduce regressions or behaviour changes that degrade detection quality.
- An outage or rate-limit event could temporarily disable analysis.

Mitigations and future work:

- Allow operators to choose between provider-managed and self-hosted models (e.g. a `gpt-oss-120b` deployment under their control).
- Isolate prompts and histories used for detection from user-facing prompts where appropriate.
- Cache recent analyses where safe, and design fail-open or fail-closed behaviour explicitly (e.g. what happens if detectors are unavailable).

A.3 Adversarial manipulation of detectors

Because ChaCha uses LLMs as judges, attackers could in principle:

- Attempt to craft histories that cause misclassification (e.g. “explain your behaviour is harmless” narratives embedded in prompts).
- Try to overload or confuse the detectors via very long or noisy histories.
- Discover prompt structures and exploit them to bypass pattern recognition.

Mitigations and future work:

- Keep the prompts used for detection internal and avoid exposing them verbatim in logs or documentation.
- Introduce sanity checks and non-LLM heuristics alongside LLM outputs (e.g. caps on the number of high-impact actions per session, independent anomaly detectors).
- Use adversarial testing to refine prompts and identify brittle behaviours.

- Consider ensembling multiple detectors (different prompts or models) for high-criticality deployments.

A.4 Integrity and multi-tenancy

As a multi-tenant control plane, ChaCha must ensure:

- Events from one project cannot affect another project's risk state or policies.
- A compromised SDK or project key cannot be used to inject events into another tenant's namespace.
- Isolation at the storage and policy layers is correctly enforced.

Mitigations:

- Use project-scoped API keys that are validated on every request and tied to a single tenant.
- Enforce project scoping at the database query level (e.g. always filter by `projectId`).
- Consider per-tenant encryption keys and network isolation for higher-assurance environments.
- Implement rate limiting and anomaly detection on the ingestion APIs to prevent abuse.

A.5 Logging, CoT, and internal model privacy

Storing CoT traces and detailed event metadata has defensive value but also raises privacy and leakage concerns:

- CoT content may reveal internal model behaviour or sensitive system prompts.
- Logs may be used to reconstruct internal workflows or business logic.

Mitigations:

- Make CoT recording an explicit, opt-in feature with clear warnings.
- Support configurable CoT redaction or summarisation at ingestion time (e.g. only retain risk labels and summaries, not raw reasoning).

- Segregate CoT storage and restrict access more tightly than general session logs.

A.6 Failure modes and defensive posture

Finally, it is important to be explicit about what ChaCha does *not* do:

- It does not guarantee that all attacks or failures will be detected. It is a signal, not an oracle.
- It does not replace basic security hygiene: least privilege, robust authentication, network segmentation, and traditional monitoring remain essential.
- It does not, by itself, enforce containment of models or agents; it provides information and suggested actions that must be wired into enforcement points.

Our defensive posture is therefore **defence in depth**: ChaCha is intended as an additional layer that makes certain classes of temporal and behavioural threat visible and controllable, not as a singular line of defence. Future work will focus on hardening the control plane itself, improving detector robustness, and integrating more deeply with existing security infrastructure so that ChaCha's signals can be combined with traditional indicators in a unified defensive strategy.