

Customer Experience (CX) Agents in Production

Patterns, Challenges, and the Four Lenses That Keep a CX Agent System Working

A field white paper on running CX agents in production — grounded in four case studies: Vodafone (Fastweb), LATAM Airlines, Lyft, and Cloudflare.

Table of Contents

1. [Executive Summary](#)
 2. [Why CX Agents Are a Different Beast](#)
 3. [The Four-Lens Model](#)
 4. [Characteristics: What Makes a CX Agent Different](#)
 5. [The Hard Challenges](#)
 6. [Agent Patterns: The Recurring Shapes](#)
 7. [The Architecture That Holds It Together](#)
 8. [The Day-to-Day: Evaluation and Operations](#)
 9. [Case Studies: The Evidence](#)
 10. [Design Principles: What This Means for Builders](#)
 11. [How EYwALINK Closes These Gaps](#)
 12. [References](#)
-

1. Executive Summary

Customer experience is the fastest-moving category in agentic AI, and it is the one where the gap between "the demo works" and "it is still working for that customer, in that conversation, right now" is widest. The ROI is easy to measure — faster responses lift conversion, fewer escalations cut cost-per-contact, higher resolution keeps customers — but the moment a CX agent faces a real, unbounded, sometimes adversarial user, the problem stops being a model-accuracy problem and becomes a **systems problem**.

This white paper is structured around the problem, not the companies. It names:

- **What makes a CX agent categorically different** from a code-gen or research agent (Section 4).
- **The four hard challenges** that are operational and organizational, not "get the LLM to answer correctly" (Section 5).
- **The recurring agent patterns** that show up across every serious deployment (Section 6).
- **The architecture and operating discipline** that hold a system together in production (Sections 7–8).

The organizing frame is a **four-lens model** — **architecture, engineering, UX, and operations** — that all have to evolve together, driven by the same production data (Section 3). The four case studies in Section 9 are the evidence for every claim:

- **Vodafone / Fastweb "Super TOBI"** — the transactional, LLM-Compiler-shaped agent and its daily automated evaluation.
- **LATAM Airlines "Concierge" + "Compass"** — the out-of-scope coverage discovery and the ontology-driven knowledge pipeline.
- **Lyft** — the self-serve agent platform: structured prompts, CI-gated linting, calibrated evals, and SLO-backed alerting.
- **Cloudflare** — the white-page self-serve configuration agent: the user as operator, agent-level tracing, and SLO discipline at the infrastructure layer.

The conclusion (Section 10) is that **the architecture is a living thing, evals are the product gate, the user's "out of scope" is your coverage gap, observability is the feedback channel, the conversation is a knowledge source, and prompts are product specs, not code comments**.

2. Why CX Agents Are a Different Beast

When you put a CX agent in front of real customers, the question you ask changes. You stop asking "does it work?" and start asking "is it still working *for that customer, in that conversation, right now*?"

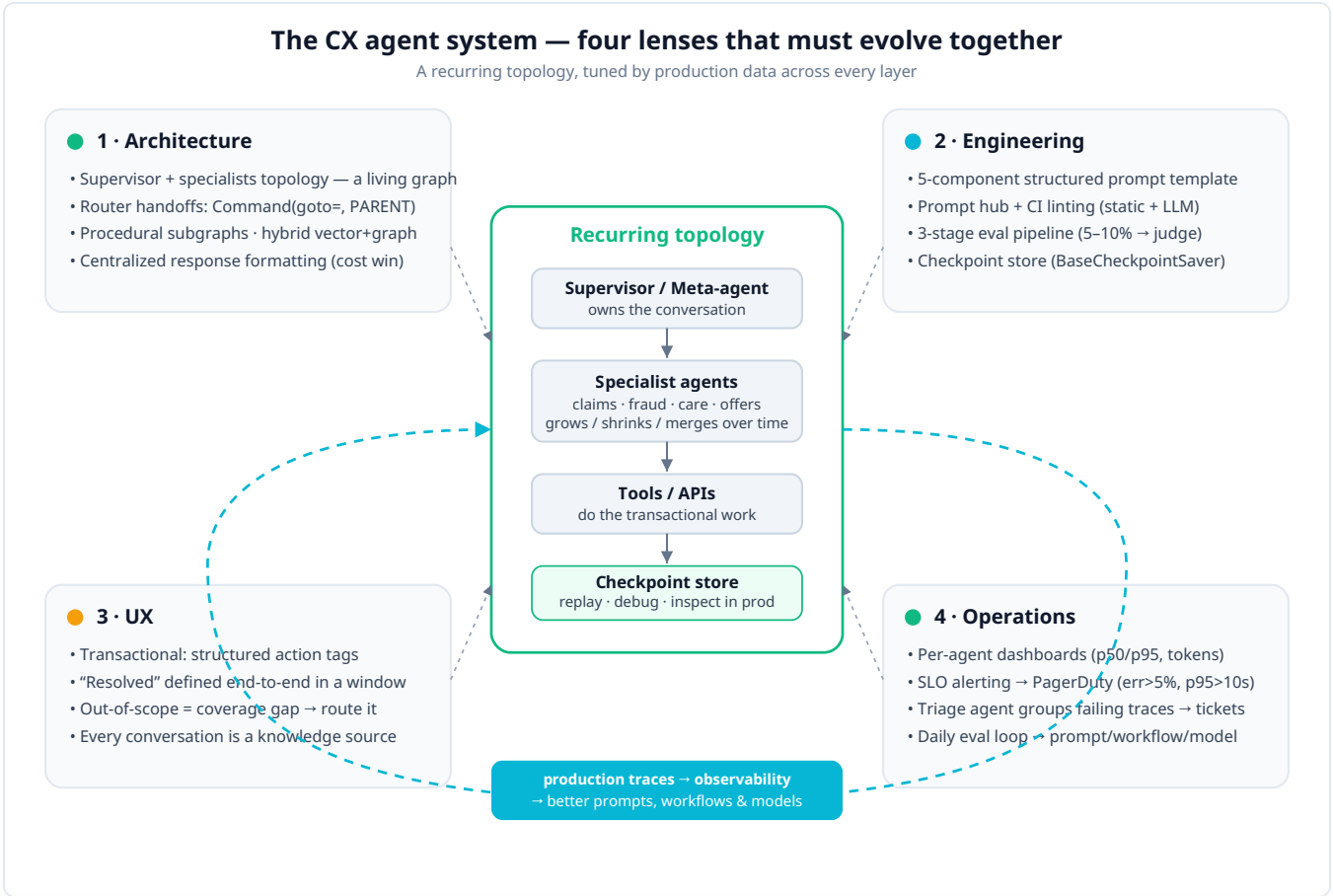
Three properties of CX make this true:

1. **The user is unbounded and adversarial.** A customer can ask anything — including things the agent has no tool for, no policy answer for, and no business case to handle. The "out of scope" boundary is not a product decision you make up front; it is a **data discovery problem** you find by reading traces, not by spec.
2. **The value is in doing, not just answering.** A CX agent that only answers questions is a Q&A machine. The value is in *executing* — activating an offer, disabling a service, updating a payment method, filing a claim. That requires **structured action tags** the frontend or backend can execute inside the conversation.
3. **"Resolved" has a specific, end-to-end meaning.** A claim is not resolved when the agent says "I've escalated it." It is resolved when it has collected evidence, run the tools, applied the decision, and explained the outcome — inside a defined time window.

The through-line: **a CX agent is not a single "agent." It is a system with four interacting layers — architecture, engineering, UX, and operations — that must evolve together.**

3. The Four-Lens Model

The single most important idea in this white paper: **treat a CX agent as a system of four lenses, not a model you pointed at a FAQ.** Each lens is a distinct discipline with its own artefacts, and all four are driven by the same production data.



The CX agent system — four lenses that must evolve together](cx-agents-four-lenses.svg

Lens	Question it answers	Key artefacts
Architecture	What is the topology, and how does it change?	Supervisor + specialist graph, router handoffs, subgraphs, retrieval layer
Engineering	How do we keep the agent correct as it grows?	Structured prompt templates, prompt hub, CI linting, eval pipeline, checkpoint store
UX	What does "resolved" mean for the customer?	Transactional action tags, end-to-end resolution, out-of-scope routing, knowledge capture
Operations	How do we know it is still working *right now*?	Per-agent dashboards, SLOs, alerting/paging, triage agent, daily eval loop

The lenses are not sequential phases. They are coupled: a change in the **architecture** (a new specialist) forces new **engineering** (prompt, tools, evals), changes **UX** (what the customer can now ask), and new **operations** (new SLOs, new dashboards). Production traces are the shared signal that moves all four at once.

4. Characteristics: What Makes a CX Agent Different

Before patterns or architecture, it helps to name what makes a CX agent categorically different from a code-generation agent or a research assistant.

4.1 The user is unbounded and adversarial

You cannot constrain the input. The agent's "out of scope" boundary is a **data discovery problem**: you find it by reading traces, not by spec.

LATAM's Concierge initially classified **13%** of messages as out of scope. They assumed users were testing the agent. When they reviewed the traces, **95% of those "out of scope" messages were legitimate passenger needs** — check-in, baggage, travel requirements, special services. It was a coverage gap, not a user problem. Adding a customer-care specialist dropped the out-of-scope rate to **1%** and lifted the return rate by **6%**.

4.2 The conversation is transactional, not just informational

The value is in **doing**. The agent must return **structured action tags** that the frontend or backend can execute inside the conversation, not just natural-language responses. Vodafone's Super TOBI returns structured action tags for activating an offer or updating a payment method; the agent stops being a chatbot and becomes a transactional workflow.

4.3 "Resolved" has a specific, end-to-end meaning

A driver damage claim is not resolved when the agent says "I've escalated your claim." It is resolved when the agent has collected photos, pulled data through tools, applied fraud signals, made a decision, and explained the outcome — all within a defined time window. Lyft's **65% deflection rate** and **35% AI resolution rate** are not vanity metrics; they are the direct result of defining "resolved" end-to-end and building the agent to that definition.

4.4 The conversation is a knowledge source, not just a cost center

Every interaction contains intent, preferences, unmet needs, and future signals. LATAM's Compass pipeline turns the unstructured corpus of conversations into structured knowledge: entities and relationships mapped against a **domain ontology** and written to a knowledge graph alongside the company's existing data lake. **The ontology — not the model — is the proprietary asset.** The model can be swapped; the ontology encodes the business's own definitions.

4.5 Real-time latency and cost matter operationally

A CX agent in a support queue has a p95 latency budget; a driver in a fare dispute will not wait 30 seconds. Every agent in production should have a **per-agent dashboard** tracking run volume, error rates, p50/p95 latency, token usage, tool-call success rates, and judge scores over time. The goal is not a pretty graph — it is an **operational alert that fires on a real SLO**.

4.6 The user can also be the operator

The CX pattern assumes a customer calling for help — a support agent answering a question or performing a transaction on a company's behalf. The same discipline applies when the user is **configuring their own environment** instead of asking a company for help. Cloudflare's agent lets a user set up their own Workers, DNS, and network configuration conversationally on a plain "white page" — no dashboard, just a chat. **The user is the operator.**

The out-of-scope boundary, the safety gates, the tool set, and the eval discipline are the same; what changes is the trust model. The agent acts on the user's **own** resources, with the user's **own** authority, not on a company's data behind a support queue. This extends the "transactional" characteristic (§4.2) to **configuration**: the agent does not just answer — it writes config, applies it, and reports back. The structured action tags pattern applies directly; the target is the user's environment, not a customer account.

5. The Hard Challenges

The hard parts are not "get the LLM to answer correctly." They are operational and organizational.

Challenge 1 · Evals are the new test suite, and harder than unit tests

The shift from "does it run?" to "is it ready to ship?" is the defining engineering challenge — and it is not just "run some evals":

- **Generic scores don't tell you what to fix.** "Helpfulness: 4/5" is non-actionable. **Binary pass/fail judgments, calibrated against human labels**, are the minimum bar.
- **Offline simulation is necessary but insufficient.** A simulated user that is too polite and too cooperative will let an agent pass offline that fails in production. You need fine-tuned simulated users with personas — refund seekers, AI skeptics — grounded in real customer verbatims.
- **The eval pipeline itself is a product.** A three-stage process: a small production rollout (5–10%), trace sampling, and LLM-as-a-Judge evaluators running automatically on production traces. Baseline metrics apply to every agent; domain-specific metrics are layered on top.
- **Daily cadence matters.** Running evals once before launch is a release gate; running them daily on a rolling window of production traces is an operational habit. The output — structured scores, explanations, violated guidelines — is what business stakeholders can actually act on.

Challenge 2 · Prompt quality is the real bottleneck, not the platform

When Lyft opened agent building to non-engineers, they assumed the hardest part would be the platform itself — tool bindings, graph edge cases, state management. They were wrong. The hardest part was **prompt quality**. Domain experts knew their issue types deeply but couldn't translate that knowledge into instructions an LLM would follow reliably. They saw agents that handled the happy path beautifully and fell apart on edge cases: a prompt that says what to do on a fare dispute but nothing when the driver changes topic mid-conversation; a tone section that says "be empathetic" without defining what that means; and — the most consistent failure mode — **missing out-of-scope definitions**, so the agent tried to answer questions it had no tools for.

The fix is a **structured five-component prompt template**:

1. **Identity** — who is this agent, what user type, what topic area
2. **Primary objective** — concrete verbs, not vague "help" or "handle"
3. **Scope** — both in-scope **and** out-of-scope, with explicit routing actions for out-of-scope

4. **Phased workflow** — numbered steps with entry conditions, branching for every if/else, and a terminal action for every phase
5. **Content guidelines** — concrete do/don't rules with example phrases, not abstract principles

Pair this with a **CI pipeline**: fast static rules (malformed template variables, duplicate intent slugs, spelling) followed by LLM-powered rules that detect prompt-injection vulnerabilities, contradictory instructions, and structural dead-ends. **All violations block the merge**. The author gets inline feedback and can fix issues without pulling in an MLE.

The key insight: **treat prompts like product specs, not code comments**. The more explicit the prompt, the more consistent the agent.

Challenge 3 · Observability is a day-one requirement, not a postmortem

You cannot run an agentic system in production without end-to-end trace visibility. The agent is a black box until you wire up tracing. The operational pattern:

- **Trace every invocation** across dev, staging, and production — which nodes ran, what the LLM saw, which tools were called, token usage, latency at every step.
- **Enrich traces with custom metadata** — user type, agent name, intent, conversation ID.
- **Per-agent cloned dashboards** — every agent in production gets its own dashboard with the metrics above.
- **Wire alerts to a paging system** — if error rate exceeds **5%** or p95 latency crosses **10 seconds** over a 15-minute window, the on-call engineer is paged automatically.
- **Build a triage agent** — a second agent that pulls failing traces, groups related issues, and opens tickets automatically. Humans design the fix; the triage agent sorts the noise.

Challenge 4 · The architecture is not finished at launch

The supervisor pattern, the subgraph structure, even which agent formats the final response — all of it gets tuned after observing real conversations. The architecture is a **living thing** that evolves with the data. You ship a supervisor that can hand off, return control, and re-route. The specialist set will grow, shrink, and merge based on what production traces show.

6. Agent Patterns: The Recurring Shapes

Across the three support deployments, the same core structure repeats:

- A **Supervisor or Meta-agent** owns the conversation
- **Specialist agents** handle discrete use cases
- **Tools / APIs** do the transactional work

The specific patterns within that structure are what matter.

Pattern	What it does	Where it shows up
Router with stateful handoffs	Separate router instances per user type; a subagent returns control to the parent on a domain shift — <code>Command(goto=..., graph=PARENT)</code>	Lyft (riders vs. drivers)
Specialized vs. configurable split	MLE-hand-built agents for complex, high-stakes flows; JSON-config + prompt-hub agents for the self-serve layer	Lyft
Procedural subgraph ("LLM Compiler")	Store procedures in a graph DB; the LLM picks *which* procedure and *in what order*, the procedure stays deterministic	Vodafone (Neo4j)
Hybrid retrieval (vector + graph)	Vector search finds chunks; the graph layer captures the relationships between them ("this supersedes that")	Knowledge-intensive CX
Centralized formatting	Move response formatting from each specialist back to the supervisor for consistency and cost	LATAM (15% cost cut)

The **specialized vs. configurable split** is the architectural enabler for self-service: new agents are defined by writing a prompt and a JSON config, not by engineering a new state graph. The platform handles graph construction, tool execution, checkpointing, tracing, and safety. The topology keeps growing.

7. The Architecture That Holds It Together

7.1 State persistence and checkpointing

For multi-turn conversations, the checkpoint store is not a detail — it is your ability to answer "what exactly did the agent see and decide at step 7?" Persist the full graph state, execution metadata, and parent checkpoint references after every node execution. That enables conversation replay, debugging, and state inspection in production. The interface is LangGraph's `BaseCheckpointSaver`; the implementation is storage-engine-agnostic — DynamoDB for a serverless setup, PostgreSQL for an on-prem one (Lyft uses `DynamoDBSaver`).

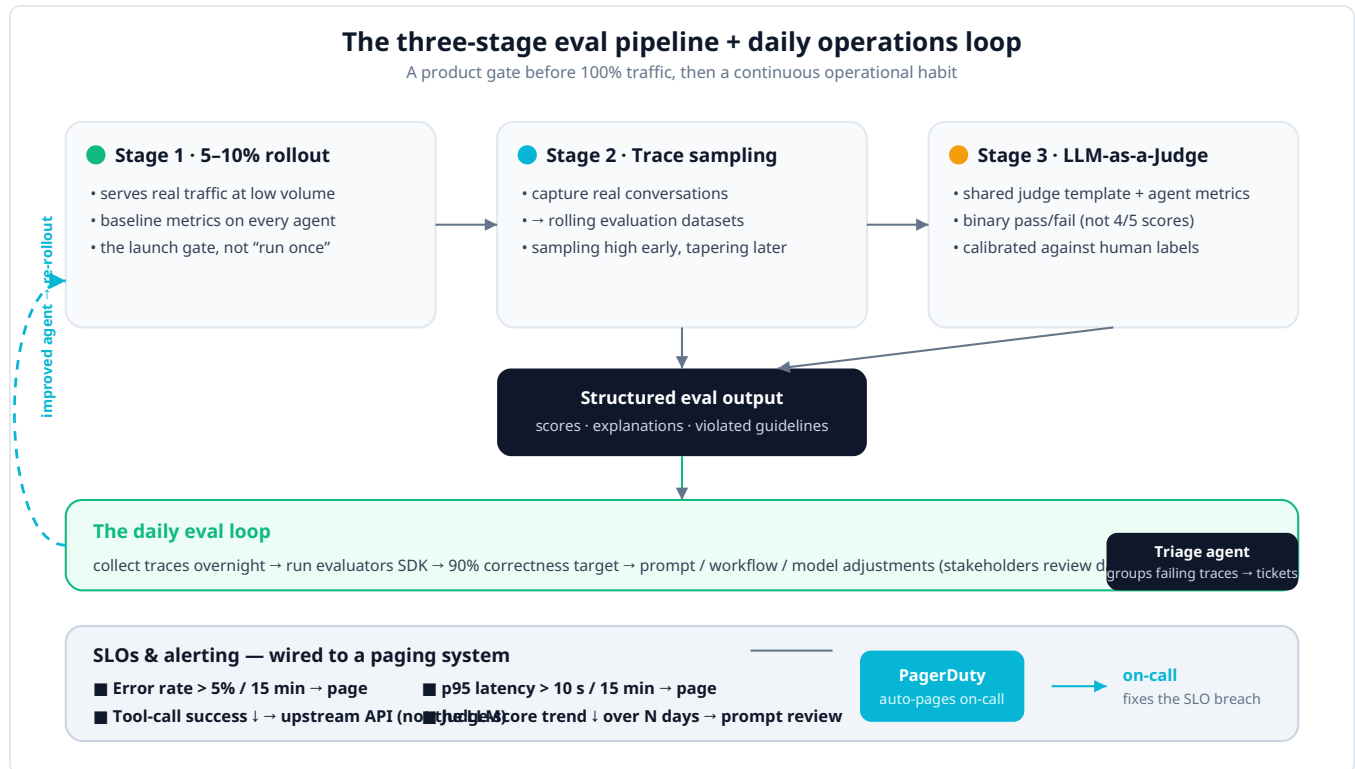
7.2 The ontology layer (for knowledge-intensive CX)

If your agent answers questions about policies, products, or procedures, the retrieval layer should not be pure vector search. Build an ontology-driven pipeline: parse unstructured sources (conversations, interviews, legal docs) → map entities and relationships against a domain ontology → write to a knowledge graph → run hybrid retrieval over the graph at query time. **The ontology encodes the business's own definitions** of "check-in deadline," "baggage allowance tier," "claim eligibility." That is the durable asset; the model is a tool.

7.3 The prompt hub as a deployment concern

A prompt hub is not just version control for prompts. It is the **deployment concern** between domain-expert knowledge and running agent behavior. It should support: structured templates with required fields; prompt linting (static + LLM-powered) before activation; prompt A/B evaluation with human reviewers; and continuous scoring on production traces with automatic degradation alerts. The prompt hub is where prompt quality becomes an **engineering discipline** rather than an individual's craft.

7.4 The eval pipeline as a product gate



The three-stage eval pipeline + daily operations loop[cx-agents-eval-pipeline.svg]

Before any agent rolls out to 100% of traffic, it must pass a three-stage process:

1. **Small production rollout (5–10%)** — the agent serves real traffic at low volume.
2. **Sample production traces** — real conversations captured as evaluation datasets.
3. **Run LLM-as-a-Judge evaluators** — using a shared judge template from the prompt hub, extended with agent-specific metrics.

The evaluators run automatically on production traces, with sampling rates that start high during initial rollout and taper as confidence grows.

8. The Day-to-Day: Evaluation and Operations

8.1 The daily eval loop

Collect traces overnight. Run the evaluators SDK. Generate structured output — scores, explanations, violated guidelines. Business stakeholders review daily and feed prompt adjustments back. The target is a **90% correctness threshold**, not "it looks fine."

8.2 The simulation infrastructure

Before a new agent ships, test it against synthetic conversations with mocked tool responses. The simulation should include personas grounded in real customer verbatims (refund seekers, AI skeptics, multilingual customers), edge cases production has revealed (topic shifts, out-of-scope questions, tool failures), and pairwise A/B comparison of prompt revisions with human reviewers. The goal: **catch regressions before a single real customer sees the output.**

8.3 The triage agent

A second agent that pulls failing traces from the observability platform, groups related issues, and opens tickets automatically. It doesn't fix anything. It sorts the noise so the human engineer spends their time on the fix, not the triage.

8.4 SLOs and alerting

Signal	Threshold	Action
Error rate	sustained > 5% over 15 min	page on-call
p95 latency	sustained > 10 s over 15 min	page on-call
Tool-call success rate	drop below threshold	upstream API problem, not the LLM
Judge-score trend	declining over N days	prompt review (not a page)

The goal is not a pretty Grafana screen. It is a structured pipeline that turns bad interactions into better prompts, workflows, and models — and pokes the on-call when something crosses a line.

8.5 Agent-level telemetry

The discipline above is not CX-specific. Cloudflare's 2026 agent-tracing launch names the same failure mode from the infrastructure side: **"An agent can return HTTP 200 and still fail. It may choose the wrong tool, pass stale context to a subagent, or spend tokens in a retry loop."** Traditional

application telemetry shows the HTTP request or the database query — but not the agent behavior that caused it.

Cloudflare's answer is **agent-aware spans nested inside the Workers infrastructure spans** — model calls, tool execution, subagent handoffs, and token usage as first-class trace nodes, exportable to any OpenTelemetry-compatible destination. Two views: **replay a session** (the full conversation — system instructions, model thinking, tool calls and their results) and **check the trace** (the execution waterfall — where time went, which subagent did the work, which D1/KV calls fired). The same SLO discipline as the Lyft case — but the telemetry is **agent-level, not HTTP-level**.

9. Case Studies: The Evidence

Vodafone / Fastweb — Super TOBI

- **Architecture** — Supervisor + use-case agents, each with a defined API set, following the "LLM Compiler" pattern. The Super Agent routes to either a procedural subgraph (troubleshooting procedures stored in Neo4j) or a hybrid vector + graph retrieval pipeline.
- **UX** — Returns structured action tags for transactional operations (activating an offer, disabling a service, updating a payment method). The agent is a workflow, not a Q&A machine.
- **Engineering** — Daily automated evaluation with the Evaluators SDK: collect traces overnight, run evaluators, generate structured output (scores, explanations, violated guidelines). **90% correctness target.**
- **Operations** — Daily eval cadence. The structured output is designed for non-technical stakeholders — they can read it and act without understanding the underlying graph.

LATAM Airlines — Concierge and Compass

- **Architecture** — Tool-per-agent layout with a supervisor delegating to specialists for flights, bookings, destinations, and customer care. Moved response formatting from each specialist back to the supervisor for a **15% cost reduction** — a finding they only caught by reading individual traces, not aggregate dashboards.
- **UX** — Initially classified **13%** of messages as out of scope; traces revealed **95%** were legitimate passenger needs. Added a customer-care specialist; out-of-scope rate dropped to **1%**, return rate up **6%**.
- **Engineering (Compass)** — Ontology-driven knowledge extraction: parse unstructured sources, map entities and relationships against a domain ontology, write to a knowledge graph. **The ontology is the proprietary asset.**
- **Operations** — Trace-level observability revealed both the duplicate-formatting cost issue and the out-of-scope discovery. A triage agent pulls failing traces and groups related issues.

Lyft — Self-Serve Agent Platform

- **Architecture** — Router pattern with separate instances for riders and drivers.
`Command(goto=..., graph=Command.PARENT)` for mid-conversation handoffs. The specialized vs. configurable split enables self-service.
- **Engineering (prompt discipline)** — Five-component structured prompt template. Git-backed prompt-linting CI pipeline (static + LLM-powered rules, blocking merges). Three-stage

eval pipeline: 5–10% rollout → trace sampling → LLM-as-a-Judge, with **binary judge outputs** instead of scores. `DynamoDBSaver` for state persistence.

- **UX** — Deflection rate **65%**, AI resolution rate **35%**. Missing out-of-scope definitions were the most consistent prompt failure mode; the fix was to require explicit out-of-scope routing actions in every prompt.
- **Operations** — Per-agent dashboards (run volume, p50/p95, token usage, tool-success rates, judge scores). PagerDuty integration: error rate > 5% or p95 > 10 s over 15 minutes auto-pages on-call.
- **Results** — Agent development time from **~6 months to ~2 weeks** for new configurable agents. **100%** of production agents have automated eval pipelines. Hallucination and contradiction rates down **20%** with guardrails built from eval metrics. AI resolution rate up **16%**.
- **What's next** — Completing the Git-backed prompt-linting pipeline; building simulation infrastructure for synthetic-conversation testing; adding pairwise A/B evaluation with human reviewers; moving from sampled to continuous scoring on all production traces with automatic prompt-degradation alerts.

Cloudflare — The Self-Serve Configuration Agent

- **Architecture** — A surface agent that lets a user configure their own Cloudflare environment — Workers, DNS, network settings — conversationally on a plain "white page." The user is the operator, not a support caller. The agent translates intent to config actions and applies them on the user's own resources.
 - **UX** — No dashboard; the interface is a chat. The "white page" is deliberate: the agent *is* the configuration surface. Structured action tags apply — the agent returns the config changes it made, not just a text response.
 - **Engineering (agent tracing)** — Cloudflare's 2026 agent-tracing platform captures agent-level spans (model calls, tool execution, subagent handoffs, token usage) nested inside the Workers infrastructure spans. OpenTelemetry-compliant, exportable to any OTLP destination — the same data that helps debug the agent powers evals, analytics, and cost reporting.
 - **Operations** — "An agent can return HTTP 200 and still fail." Agent-aware tracing with two views: **replay a session** (full conversation: system instructions, model thinking, tool calls and results) and **check the trace** (execution waterfall: where time went, which subagent did the work, which D1/KV calls fired). The same SLO discipline as the Lyft case — but the telemetry is agent-level, not HTTP-level.
 - **What it extends** — The "user as operator" archetype (§4.6). Same disciplines as Lyft's configurable agent split, applied to user-facing configuration rather than internal support. The trust model differs: the agent acts on the user's own resources with the user's own authority, so the safety gates are scoped to the user's environment, not a company's data.
-

10. Design Principles: What This Means for Builders

1. **The architecture is a living thing.** Don't ship a fixed topology. Ship a supervisor that can hand off, return control, and re-route. The specialist set will grow, shrink, and merge based on what production traces show.
 2. **Evals are the product gate.** Generic scores don't tell you what to fix. Use binary pass/fail judgments, narrow behavior-specific rubrics where every failure points to a specific change, judges calibrated against human labels, offline simulation as a launch gate, and online annotation as the continuous loop.
 3. **The user's "out of scope" is your coverage gap.** When the agent says "I can't help with that," the first question is: *is that actually a legitimate need we haven't built for yet?* The out-of-scope rate is a coverage metric, not a user-behavior metric.
 4. **Observability is the feedback channel — wire it into alerting.** Tracing from day one is table stakes. The next step is a paging system: error-rate thresholds, p95 SLOs, tool-call success rates. The goal is a structured pipeline that turns bad interactions into better prompts, workflows, and models — and pokes the on-call when something crosses a line.
 5. **The conversation is a knowledge source.** Agents are not just answerers; they are sensors. Every interaction contains intent, preferences, unmet needs, and future signals. The ontology is the durable asset; the model is a tool.
 6. **Prompts are product specs, not code comments.** The more explicit the prompt — identity, objective, in-scope and out-of-scope with routing actions, phased workflow with entry/exit conditions, concrete content guidelines — the more consistent the agent. Build a structured template, build a linting pipeline, and make domain experts pass through CI before a prompt reaches production.
-
1. **The user can be the operator.** The CX pattern generalizes: when the user configures their own environment (Cloudflare's white-page agent), the disciplines are the same — structured prompts, evals, agent-level tracing, SLOs — but the trust model shifts. The agent acts on the user's own resources with the user's own authority, so the out-of-scope boundary and the safety gates are scoped to the user's environment, not a company's data. **The architecture is the same; the permission model is different.**
-

11. How EYwALINK Closes These Gaps

EYwALINK's four strategic pillars are a **problem-first decomposition** of exactly what this white paper describes. Each pillar corresponds to a section; the product portfolio closes the specific gaps the post names.

EYwALINK Pillar	White-paper anchor	What it adds
Pillar 1 · AI Strategy & Architecture	§5.4 "architecture is not finished" + §6 patterns	Executable EA maps the TOGAF target state to the supervisor/specialist topology, and keeps it a living artifact that evolves with production data instead of a diagram that goes stale at launch.
Pillar 2 · AI Safety & Privacy	§4.1 "unbounded and adversarial" + prompt linting	Safety gates, out-of-scope definitions, and prompt-injection detection in the linting pipeline are compliance requirements for regulated CX (telco, aviation, healthcare). No data leaves client infrastructure: the ontology, eval judges, and knowledge graph all run on client-owned infrastructure.
Pillar 3 · Rapid AI-Led Application Development	§9 Lyft "6 months → 2 weeks" + the configurable split	"Rapid" means structured prompt templates with CI gates, not improvised prompts. The configurable split is the mechanism: domain experts write the prompt, the platform handles graph construction, tool binding, checkpointing, and tracing.
Pillar 4 · AI Observability & Operations	§8 Implementation and Operation	Model-health tracking, automated performance-regression detection, agent-failure detection, SLA-backed incident response, and a fractional AI-operations team that is always on and always accountable.

The product portfolio maps one-to-one onto the gaps this white paper names:

- **Observability Stack** (OTel Collector, Phoenix tracing, Prometheus/Grafana/Loki) = §8.4 "per-agent dashboards + PagerDuty wired to SLOs." The difference: **self-hosted, on client infrastructure, no data leaves the perimeter.**
- **SuperWeb Testing** (four-phase E2E pipeline) = §8.2 simulation infrastructure + §7.4 three-stage eval pipeline — synthetic-conversation testing with mocked tool responses before a single real customer sees the output.
- **Loop Factory** (self-improving LangGraph engine) = §8.3 the triage agent plus the ADLC iterate loop — the agent that reads failing traces and opens tickets is a Loop Factory instance.

- **Executable Enterprise Architecture** (TOGAF ADM made executable and AI-agentic) = §10.1 operationalized — the target-state diagram is the starting state of a continuous cycle where production data feeds back into the architecture.
- **The configuration-agent archetype** (Cloudflare's white page) = the "user as operator" pattern (§4.6 / §10.7). Pillar 3 (rapid AI-led development) is the mechanism for building these; Pillar 4 (observability) is what makes the user trust the agent acting on their own resources. The white-page UX — no dashboard, just a chat — is the minimal surface a configuration agent needs. The difference from Lyft: the user owns the resources, so Pillar 2 (no data leaves client infrastructure) is the **more** relevant constraint, not the less.

The positioning gap this white paper surfaces is the one EYwALINK exists to close: **the reference stack (LangGraph + LangSmith + PagerDuty) is a proprietary platform the client does not own.** The operational discipline — structured prompts, calibrated binary evals, daily eval cadence, SLO-backed alerting, a triage agent — is the same either way. What changes when you **own** the infrastructure is that the out-of-scope boundary, the ontology, the trace data, and the eval judges are all **client assets** rather than platform dependencies. The zero-lock-in, private-AI position is not a compromise; it is the prerequisite for Pillar 2 — and Pillar 2 is what makes the whole system viable in a regulated CX environment.

12. References

- [How Lyft Built a Self-Serve AI Agent Platform for Customer Support with LangGraph and LangSmith](#) — Akshay Sharma, Lyft SCX Data Science & MLE team, 27 May 2026
 - [The Agent Development Lifecycle](#) — LangChain, 9 May 2026
 - [Customer Experience Agents in Production: Lessons from Lyft, Vodafone, and LATAM Airlines](#) — LangChain field notes
 - [Introducing: Cloudflare Agents](#) — Nevi Shah, Matt Simpson, Fred Schott, Cloudflare, Aug 2026
 - [What is an AI agent?](#) — Cloudflare Learning
 - [Agent setup — Cloudflare Developer Docs](#) — Cloudflare
 - All numbers and quotes trace to the sources above.
-

Appendix A · Diagram Sources

- `cx-agents-four-lenses.puml` → `cx-agents-four-lenses.svg` (the four-lens system + recurring topology)
- `cx-agents-eval-pipeline.puml` → `cx-agents-eval-pipeline.svg` (three-stage eval pipeline + SLO alerting)

Render (no Java required — public PlantUML server):

```
python3 whitepaper/scripts/plantuml_render.py cx-agents-four-lenses.puml
```

White paper by the EYwALINK team. Source: [eywalink-site / whitepaper](#) (regenerate with the `whitepaper` build script). Companion interactive page: `src/pages/resources/cx-agents-whitepaper.astro`.