timeplus

# Streaming SQL from Agent Telemetry to Agent Action

## Closing the Loop with Open Source



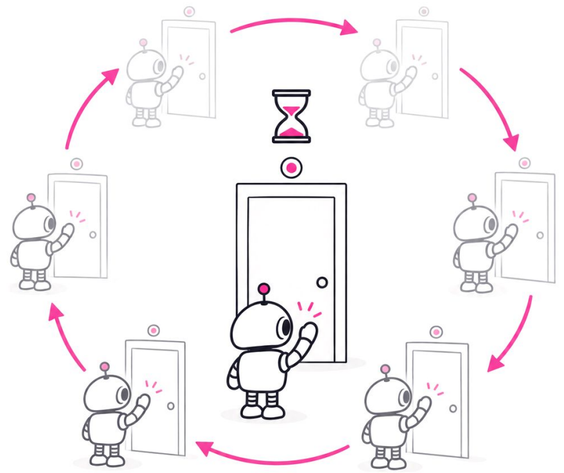
**Gang Tao**
Co-Founder & CTO
Timeplus

**OSACON** | November 2 2026, San Francisco

THE PROBLEM

# Three agent failures burn money within seconds

An agent takes a step every few seconds. A five-minute alert window sees dozens of steps.

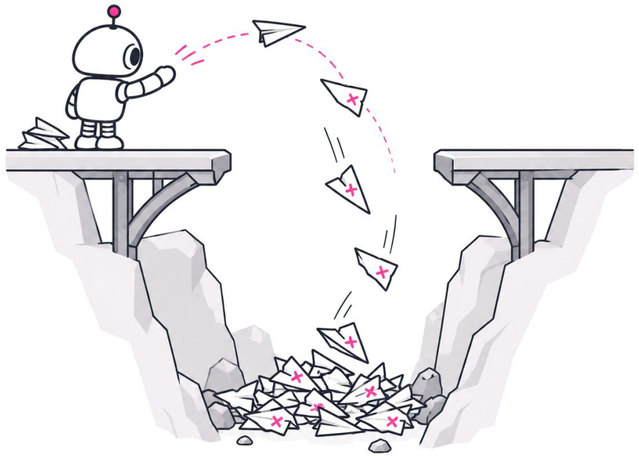


## 1 Polling loop

Told to wait for a deploy, it checks the status again and again. The deploy is stuck, and the agent can't tell.

IN THE TELEMETRY

**Same command, 4 times in a row**

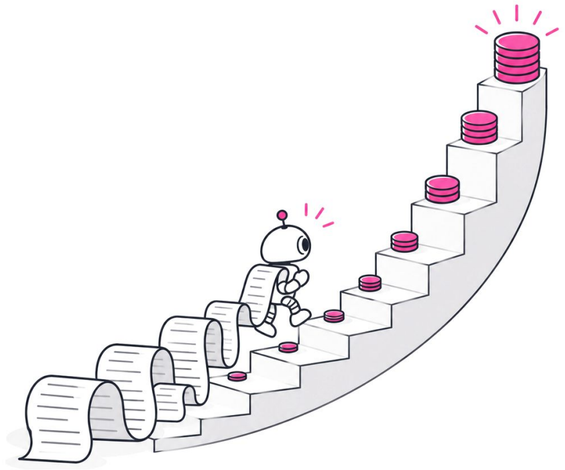


## 2 Retry storm

A dependency is down. The error looks transient, so the agent keeps retrying the same call.

IN THE TELEMETRY

**success = false, 3 times in a row**



## 3 Runaway spend

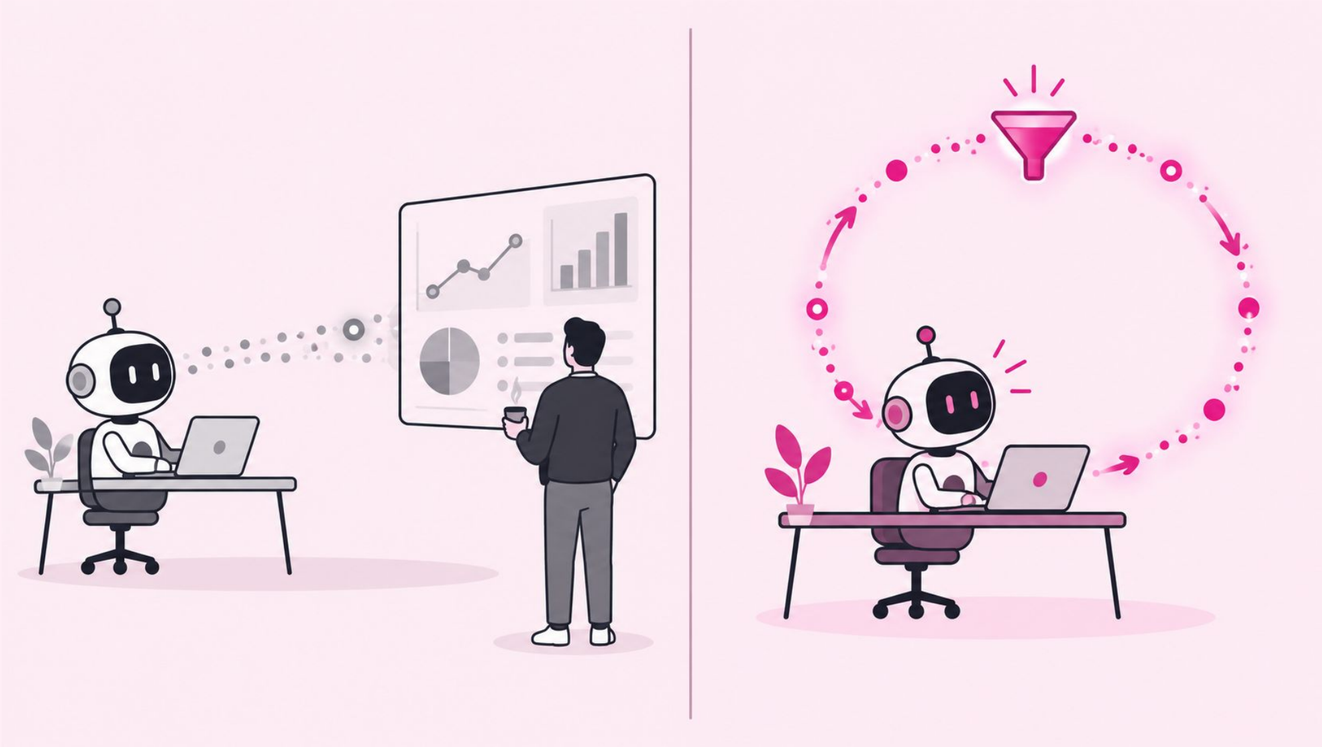
Each step re-reads everything so far, so every model call costs more than the last.

IN THE TELEMETRY

**Session cost crosses its budget**

# Agents are the first systems that can act on their own telemetry.

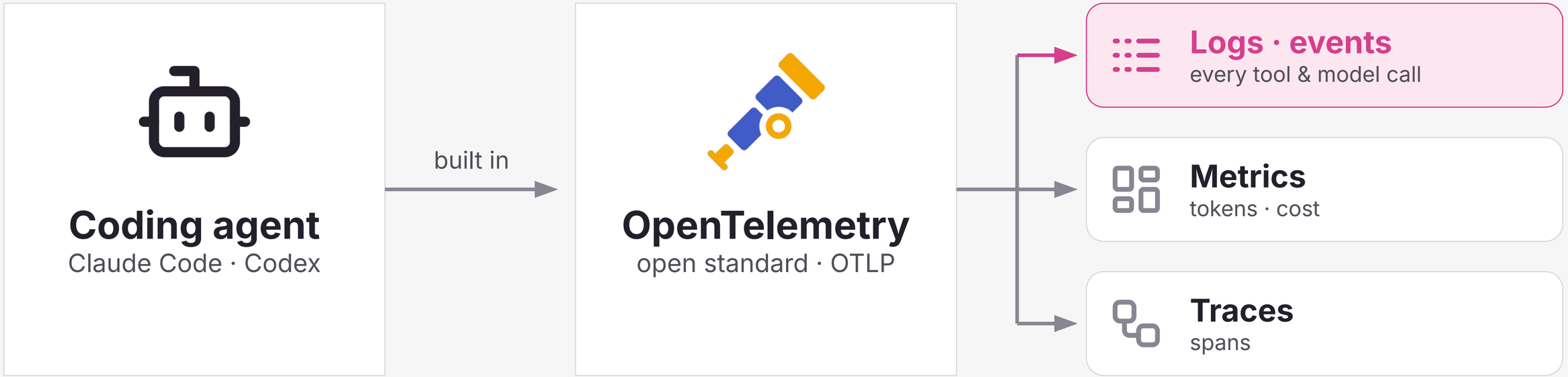
The same data pipeline can feed two readers: a dashboard for humans, and a hook that hands decisions back to the agent.



THE SIGNAL

# Agents already speak OpenTelemetry

An open standard, already built into the coding agents we use.

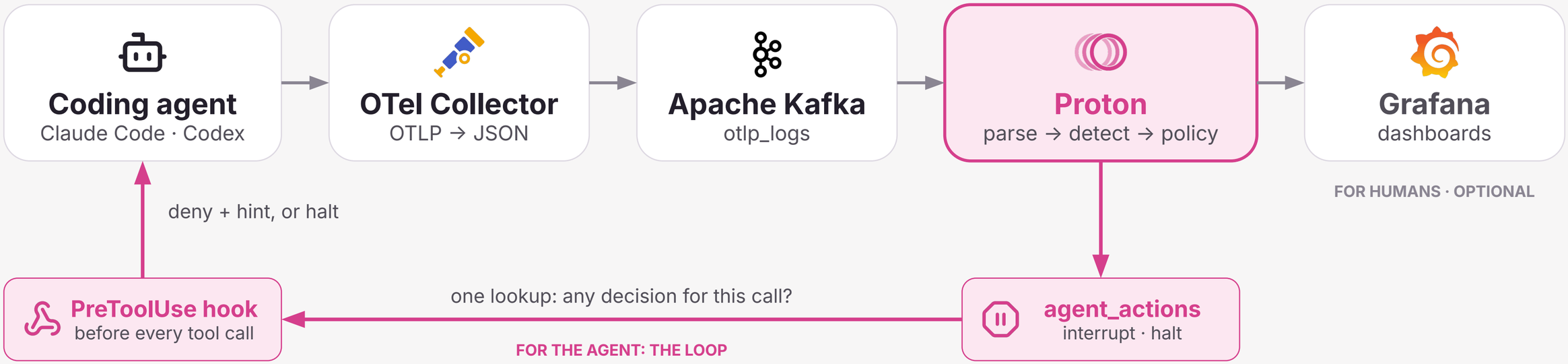



Nobody has to instrument anything. **The signal is already there.**

ARCHITECTURE

# The loop: signal out, decision back in

Standard OpenTelemetry out, streaming SQL in the middle, and the agent's own hook brings the decision back.




One stream, two readers. The pink loop runs with nobody watching; Grafana is for the human.

THE ENGINE

# Timeplus Proton: SQL that never stops running

An open-source streaming SQL engine. Every detector and policy in this talk is a query Proton keeps running.

**Open source**
Apache-2.0 · github.com/timeplus-io/proton

**One binary**
C++, no JVM, no cluster to stand up. Runs on a laptop.

**ClickHouse-derived**
Columnar, vectorized, familiar SQL.

**Live + history**
One engine for “what's happening” and “what happened.”

SAME SQL, TWO MODES

```
-- streaming: never finishes, pushes each new row
SELECT session_id, tool_name
FROM agent_events
WHERE NOT success;

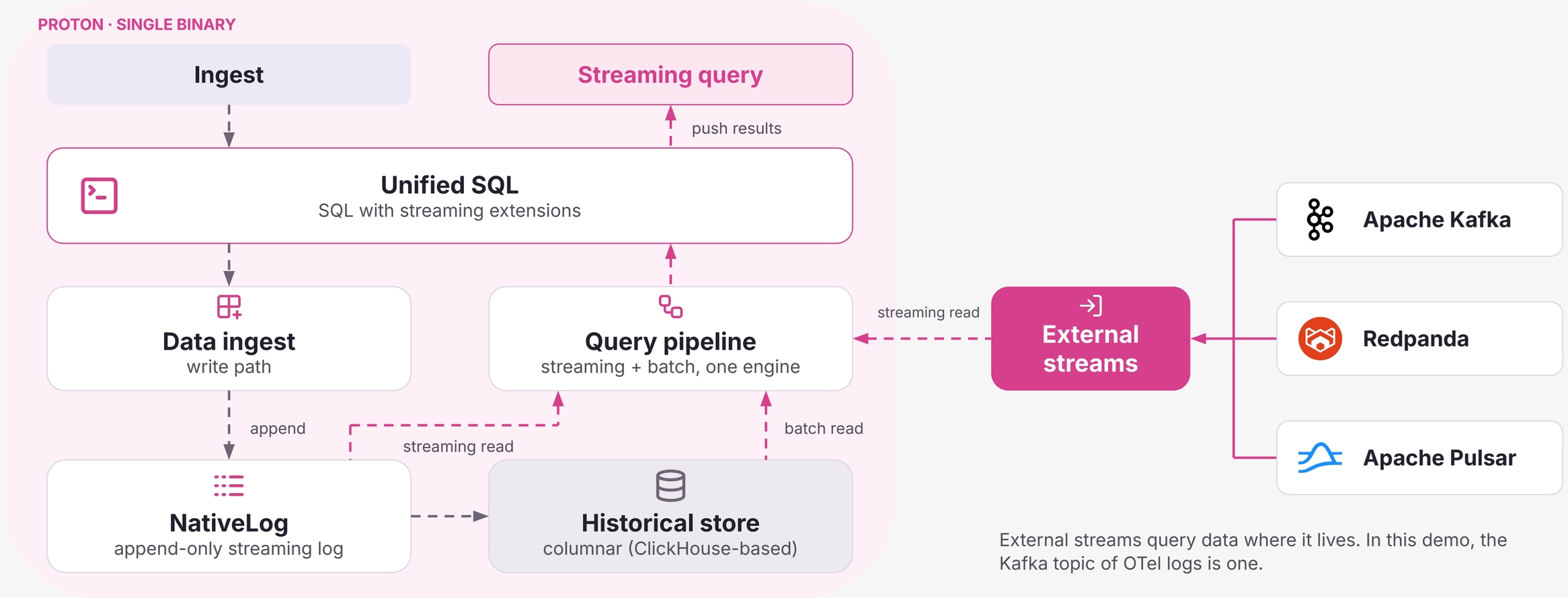
-- historical: scans what's stored, then returns
SELECT tool_name, count() AS failures
FROM table(agent_events)
WHERE NOT success
GROUP BY tool_name;
```

**STREAMING SQL** A query over a stream that never ends: results update as each event arrives. A **materialized view** is one that runs forever and writes its results into another stream.

THE ENGINE

# Inside Proton

Ingest once into a streaming log. Query it live, or as history, with the same SQL.

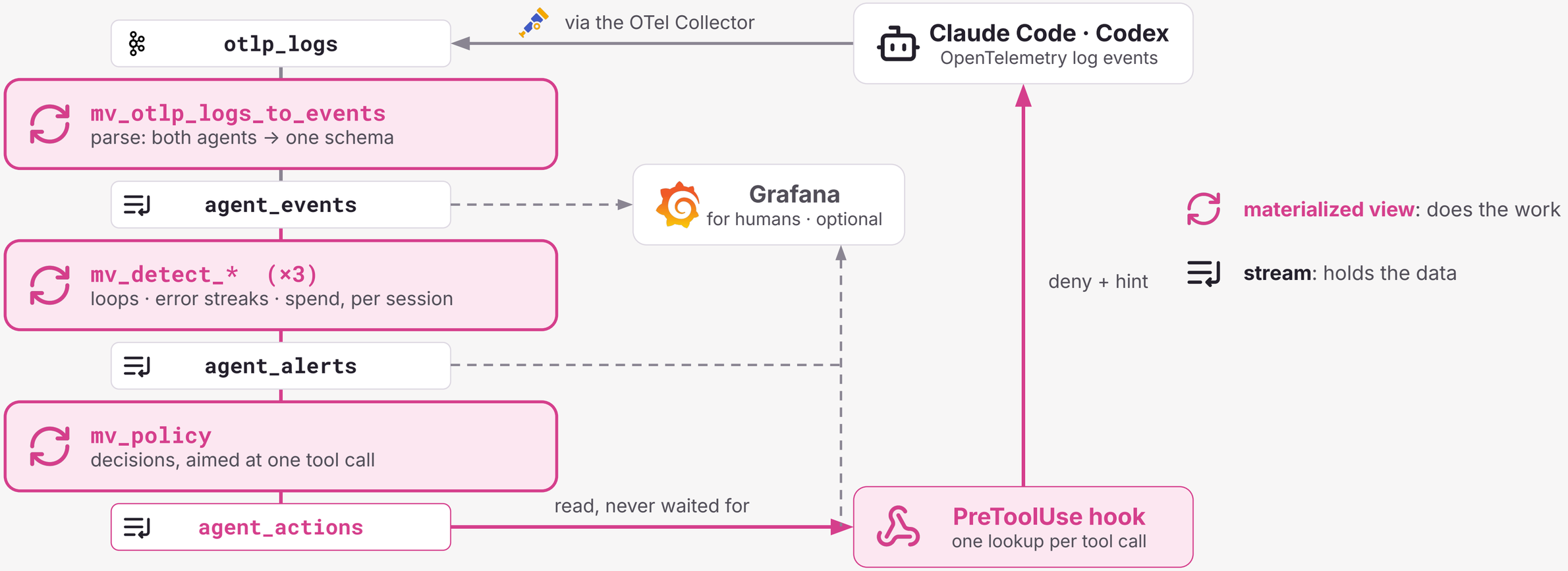



External streams query data where it lives. In this demo, the Kafka topic of OTel logs is one.

THE PIPELINE

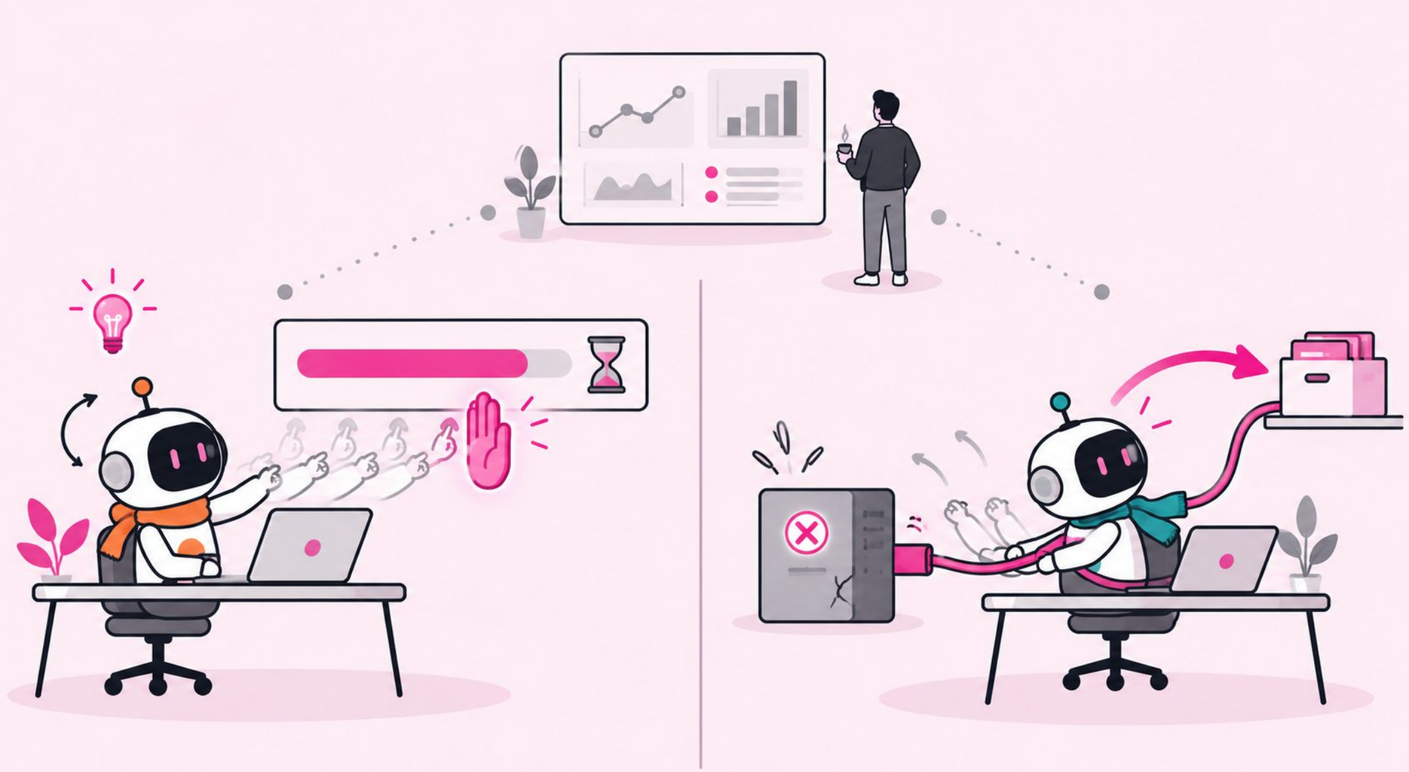
# The pipeline: streams hold data, views do the work

Every pink card is a materialized view: always running, nothing scheduled.



# Observe first.

# Then one SQL view closes the loop.



SUMMARY

# Why Proton: agent monitoring is a streaming problem

Three reasons streaming SQL, and Proton, fit agent monitoring.

1

## Reacts in milliseconds

Agents act every few seconds. Proton reacts to each event, per session, in about 100 ms.

2

## Every rule is just SQL

Loops, retry storms, runaway spend: each is one materialized view. Readable, reviewable, always running.

3

## Open, runs anywhere

One Apache-2.0 binary on the OpenTelemetry and Kafka you already run. Even air-gapped, next to the agent.

Agents emit the signal. **Proton turns it into a decision before the next tool call.**

# Try it yourself!

## TRY IT

| | |
|---|---|
| Demo | **github.com/timeplus-io/osacon-2026**<br>`cd demo && docker compose up` |
| Proton | **github.com/timeplus-io/proton** |

## JOIN THE COMMUNITY

| | |
|---|---|
| Timeplus | **timeplus.com** |
| Slack | **timeplus.com/slack** |

## TALK TO ME · GANG TAO

| | |
|---|---|
| LinkedIn | **linkedin.com/in/taogang/** |
| Email | **gang@timeplus.com** |

Scan for timeplus.com

BACKUP

# Reference slides for Q&A

Agent events · detector and policy SQL · the hook · config · the parse view · timing · open problems · full pipeline. All of it is in the repo.

BACKUP · AGENT EVENTS

# Coding agents already emit every step as OpenTelemetry

Claude Code and Codex export a log event for every tool call and model call. A few settings turn it on.

**CLAUDE CODE**

```
claude_code.tool_result
  session.id      9f2c41e0-…
  tool_name       Bash
  success         true
  duration_ms     412
  tool_input      {"command":"./check_deploy.sh"}

claude_code.api_request
  model           claude-sonnet
  input_tokens    1240
  output_tokens   310
  cost_usd        0.031
```

**CODEX (APACHE-2.0)**

```
codex.tool_result
  conversation.id  0199a7c2-…
  tool_name        shell
  success          false
  duration_ms      380
  arguments        {"command":["bash","-lc",
                     "./fetch_metrics.sh"]}

codex.sse_event
  input_token_count    18400
  output_token_count   612
```

Different names for the same facts. One SQL view maps both agents onto one schema.

BACKUP · DETECT

# A detector is a materialized view

Per-session state, fires once per threshold, runs continuously, and doesn't care which agent it watches.

03_DETECTORS.SQL

```
CREATE MATERIALIZED VIEW mv_detect_tool_loop
INTO agent_alerts AS
SELECT session_id, 'tool_loop' AS detector,
       call_target AS target, ...
FROM (
  SELECT session_id, call_target,
    concat(tool_name, '|', call_target) AS call_key,
    lags(call_key, 1, 100) AS prev,
    array_first_index(x -> x != call_key, prev)
      AS streak     -- identical calls in a row
  FROM agent_events
  WHERE event = 'tool_result'
  PARTITION BY session_id
)
WHERE streak >= 4 AND streak % 4 = 0
```

1 **`PARTITION BY session_id`**
Each session keeps its own state.
Thousands run side by side.

2 **`lags(call_key, 1, 100)`**
That session's recent calls; streak counts identical ones in a row.

3 **`streak % 4 = 0`**
Fires on the 4th identical call, then every 4th while it keeps going.

Same shape: failure streaks (3, 6, 9) and spend (each multiple of the budget).

BACKUP · DECIDE AND DELIVER

# The agent's own hook delivers the decision

The policy is SQL. Detection runs in the background; the hook only applies decisions that already exist.

04_POLICY.SQL

```
CREATE MATERIALIZED VIEW mv_policy
INTO agent_actions AS
SELECT
  session_id,
  multi_if(detector = 'budget', 'halt',
           escalate,            'halt',
           is_pattern,          'interrupt',
                                'log') AS action,
  target,       -- the call it is aimed at
  ... AS hint   -- the sentence the model reads
FROM (SELECT ...,
  date_diff('millisecond', _tp_time, now64(3)) < 2000 AS live,
  lags(if(is_pattern AND live, detector, ''), 1, 2) AS prev
  FROM agent_alerts PARTITION BY session_id)
-- escalate = 3rd live hint this session
```

1. **The agent reaches for its next tool call.**
   ./check_deploy.sh, for the fifth time.

2. **Its PreToolUse hook asks Proton.**
   Any decision aimed at this call? One lookup, ~0.1 s.

3. **A match is blocked, once.**
   The hint is the reason the model reads. Other calls run.

4. **The model changes course.**
   Ignore it twice: halt. A person types [resume] to lift it.

BACKUP · THE HOOK

# Our hook: proton_gate.py

Stdlib Python in Claude Code's own hooks. One lookup per tool call, and it fails open.

DEMO/HOOKS/PROTON_GATE.PY (SIMPLIFIED)

```
def handle(payload):
    event = payload["hook_event_name"]
    if event == "UserPromptSubmit":
        if "[resume]" in payload["prompt"]:
            resume(session)        # writes a 'resume' row
    if event == "PreToolUse":
        rows = fetch_actions(session, call)  # one query
        if state.halted:
            return deny(halt_reason)
        hit = [r for r in rows if r["matches"]]
        if hit:
            state.consumed += hit  # block once
            return deny(hit[-1]["hint"])
        return None                # allow
    if event == "PostToolUse" and state.halted:
        return {"continue": False}

# error or no answer in 1.5 s -> allow (fail open)
```

1. **Wired into Claude Code's hooks.**
   Pre/PostToolUse and UserPromptSubmit run it.

2. **One query per tool call.**
   Fresh decisions aimed at this call. ~15 ms.

3. **Deny once, with the hint.**
   The model reads it as the tool result. Other calls run.

4. **Halt, resume, fail open.**
   [resume] lifts a halt. No answer in 1.5 s: allow.

BACKUP · CONFIGURATION

# Turning on the signal is configuration

Proton OSS has no OTLP receiver, so Kafka is the bridge: the Collector writes OTLP/JSON, Proton reads the topic.

### CLAUDE CODE · ENV

```
CLAUDE_CODE_ENABLE_TELEMETRY=1
OTEL_LOGS_EXPORTER=otlp
OTEL_EXPORTER_OTLP_PROTOCOL=
  http/protobuf
OTEL_EXPORTER_OTLP_ENDPOINT=
  http://otel-collector:4318
OTEL_LOG_TOOL_DETAILS=1
OTEL_LOGS_EXPORT_INTERVAL=500
```

### CODEX · CONFIG.TOML

```
# config.toml
[otel]
log_user_prompt = true

[otel.exporter.otlp-http]
protocol = "binary"
endpoint = "http://otel-collector:4318/v1/logs"
```

### OTEL COLLECTOR → KAFKA

```
exporters:
  kafka/logs:
    brokers: [kafka:9092]
    topic: otlp_logs
    encoding: otlp_json

service:
  pipelines:
    logs:
      receivers: [otlp]
      exporters: [kafka/logs]
```

BACKUP · PARSE

# One view maps both agents onto one schema

Proton: Apache-2.0, single binary, ClickHouse-derived. The same SQL runs on live streams and on history.

02_PARSE_OTLP_LOGS.SQL

```
CREATE MATERIALIZED VIEW mv_otlp_logs_to_events
INTO agent_events AS
SELECT
  if(attrs['session.id'] != '',
     attrs['session.id'],
     attrs['conversation.id'])         AS session_id,
  attrs['tool_name']                   AS tool_name,
  agent_call_target(tool_args)         AS call_target,
  attrs['success'] != 'false'          AS success,
  ...
FROM (
  SELECT array_join(...logRecords...) ...
  FROM otlp_logs_kafka     -- read from Kafka
)
```

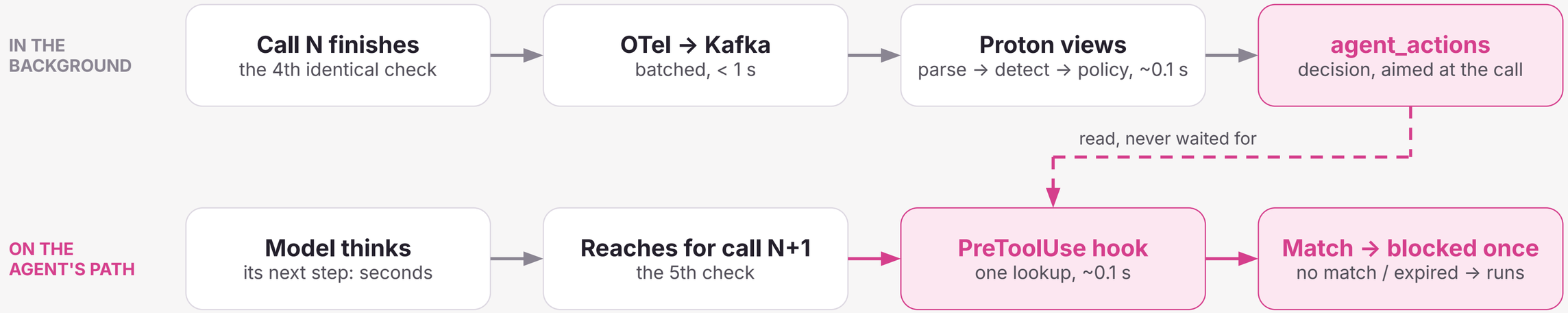
| agent_events | Claude Code | Codex |
|---|---|---|
| **session_id** | session.id | conversation.id |
| **tool_args** | tool_input | arguments |
| **tokens** | input_tokens | input_token_count |
| **cost_usd** | cost_usd | not reported |

**Compare the target, not the raw arguments.**
Agents rewrite the description and the sleep on every retry. agent_call_target() strips them, so the loop still shows up.

BACKUP · TIMING

# How a decision reaches the agent

Detection runs in the background. Enforcement is one lookup on the agent's next call.




| | Where it runs | Waits? | Good for |
|---|---|---|---|
| **Single-call rules** | hook / agent permissions | yes, on the agent's path | never run rm -rf, never read secrets |
| **Patterns across calls** | streaming SQL | no, in the background | loops, retry storms, spend, drift |

BACKUP · OPEN PROBLEMS

# What the loop does not solve yet

Where this needs more work.

**Semantic loops**

Same intent, different commands. Catching them needs embeddings in the pipeline, which adds latency and cost.

**Between tool calls**

The hook acts at the next tool call. A model that is only generating text can't be interrupted this way.

**Multi-agent causality**

When one agent's output sends another into a loop, the pattern spans sessions.

**Sensitive content**

Tool arguments and prompts in telemetry are a governance problem. Tool details are off by default for a reason.

**Fully local**

Codex can run on a local model. With that, the whole loop runs without outbound network access.

BACKUP · FULL PIPELINE

# Every stream and view in the demo

One Kafka topic in, one stream of decisions out. Nothing else in the SQL.

from Claude Code and Codex, via the OTel Collector

