KubeCon CloudNativeCon North America 2026

#KubeCon #CloudNativeCon

No RAG, No Regrets: Letting an LLM Drive 30 EKS Clusters in Production

Trevor Linton, mastercard



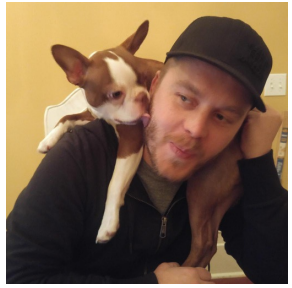

These are my views
not my employers.













KubeCon CloudNativeCon North America 2026

Mastercard Solution Studio

Home | APIs | API Bundles | Use Cases

Dark Light

API BUNDLES

About

- Account intelligence & PFM 2/7
- Subscription & card-on-file lifecycle 1/3
- Loyalty, offers & benefits 0/10
- Merchant resolution 0/3
- Issuer & commercial toolkit 0/5
- Acquirer & risk 0/4
- Sustainability / ESG 1/4

SDK

Global Open Finance SDK

US Open Finance requires a US IP address. US IP — The US region of this SDK calls Finicity APIs which only respond to US IP addresses. Connect via a US VPN if you are outside the United States. The AU and EU regions are not affected.

MASTERCARD OPEN FINANCE · SDK

One integration.
Three continents.

Mastercard is the most global Open Finance provider in the world today. Through a **single integration** you can activate connected account data, payments and insights across **Europe**, **Australia** and the **United States** — with one consistent SDK and one command-line tool.

3	1	100%
regions, one SDK	integration to maintain	consistent surface

Watch the CLI | Explore regions

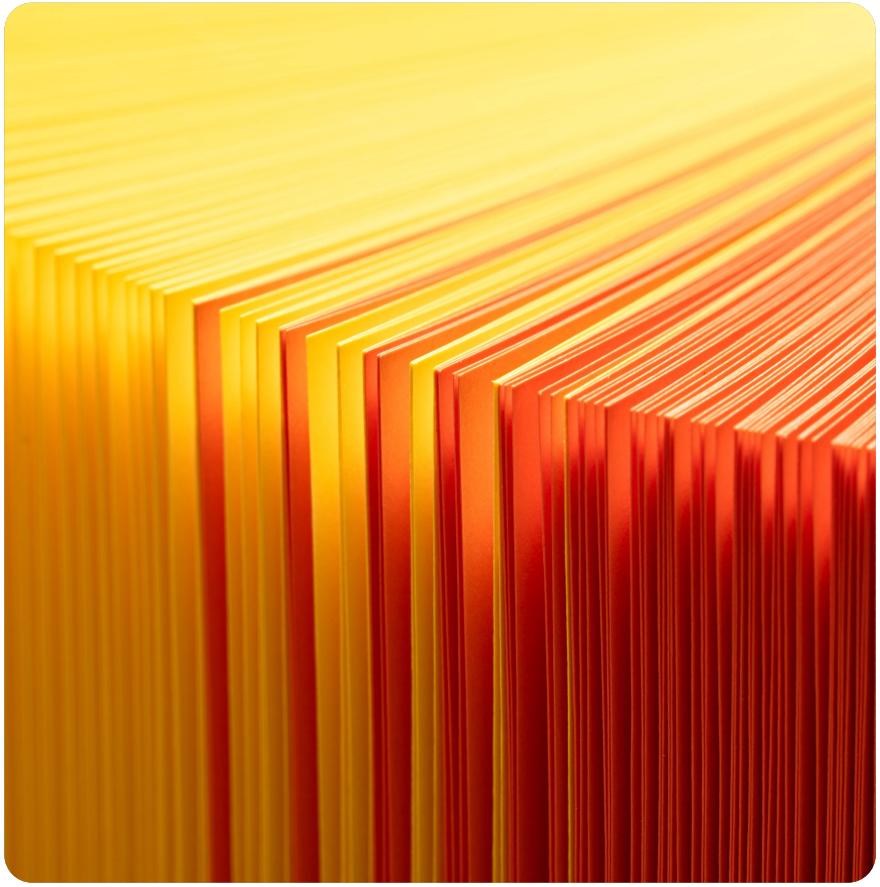


KubeCon CloudNativeCon North America 2026

Part 1: The Problem

With AI, we will see a 10x increase in productivity.

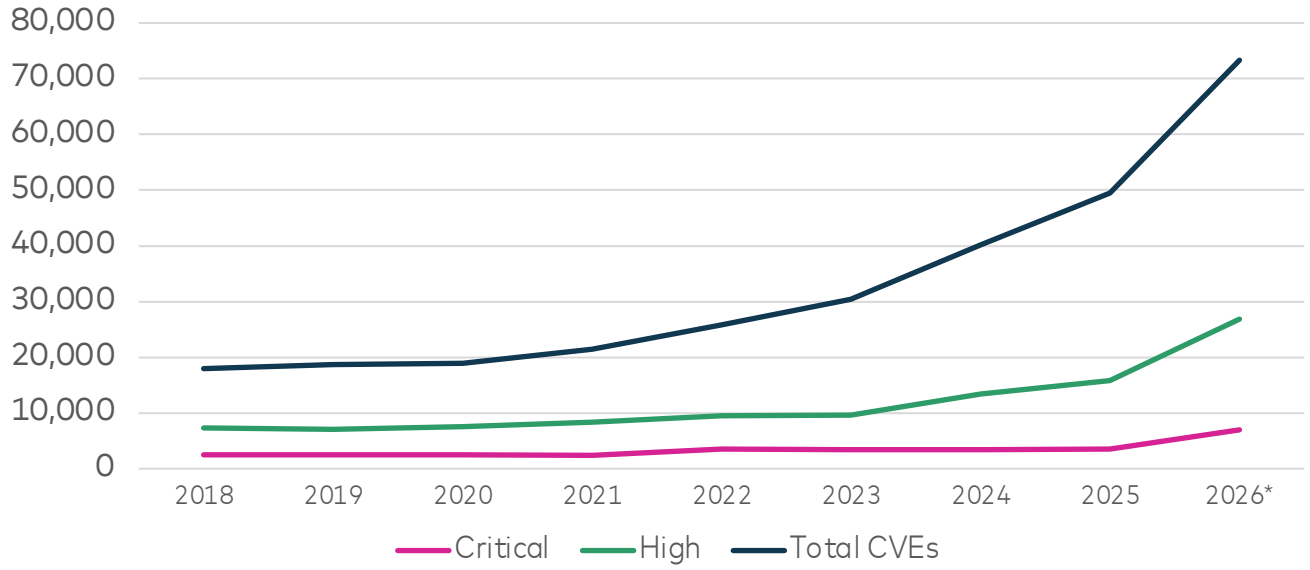
~ EVERY CTO & EVP



With AI, we will see a 10x increase in vulnerabilities.

~ EVERY CISO & BSO





2026 data estimated from first five months.
Source: projectdiscovery.io/blog/the-vulnerability-curve-bent-with-the-ai-curve

repeat in the next environment: dev → staging → production

1 Plan
know exactly what is changing

- **Run Pre-Validation Checks** — health, current versions, capacity
- **Review Breaking Changes** — removed APIs, renamed flags, migrations
- **Check Compatibility** — platform version, dependencies, CRDs
- **Read the Release Notes** — every release between current and target

Pick the target version once
It is pinned and promoted unchanged through dev, staging and production.

2 Code
describe the change as code

- **Update Infrastructure Code** — versions in IaC, charts and manifests
- **Update Configuration** — values, flags, feature gates, limits
- **Open a Merge Request** — every change is reviewed before it ships
- **Pass CI Checks** — tests, lint, policy and security scans

```
values.yaml
- version: 1.14.2
+ version: 1.15.0
```

3 Deploy
roll it out safely

- **File a Change Ticket** — approval and a change window — prod only
- **Provision New Resources** — node pools, IAM roles, storage — if needed
- **Deploy the Change** — merge; the pipeline or GitOps applies it
- **Watch the Rollout** — readiness, events and error rates

Production follows its ticket
Changes go out inside the approved change window, linked to the MRs.

4 Validate
prove it worked, or undo it

- **Run Post-Validation Tests** — health checks, smoke tests, metrics
- **Notify Stakeholders** — owners and channels hear the outcome
- **Close the Change Ticket** — record what happened — prod only

IF VALIDATION FAILS

- **Roll Back the Change** — revert to the last good version
- **Investigate Anomalies** — logs, events, dashboards, then fix

rollout stops: nothing is promoted until it passes

Repeat Per Environment
each environment runs all four phases; the version moves on only after it validates

Dev
first and lowest risk: catch problems early

Plan › Code › Deploy › Validate

validated
same version

Staging
production-like: rehearse the exact change

Plan › Code › Deploy › Validate

validated
same version

Production
+ change ticket

customer-facing: ticketed, inside a change window

Plan › Code › Deploy › Validate

TAKEAWAY Plan once, pin the version, and promote it one environment at a time. A failed validation stops the rollout where it is.

KubeCon CloudNativeCon North America 2026

repeat in the next environment: dev → staging → production

1

Plan

know exactly what is changing

- **Run Pre-Validation Checks** — health, current versions, capacity
- **Review Breaking Changes** — removed APIs, renamed flags, migrations
- **Check Compatibility** — platform version, dependencies, CRDs
- **Read the Release Notes** — every release between current and target

Pick the target version once
It is pinned and promoted unchanged through dev, staging and production.

2

Code

describe the change as code

- **Update Infrastructure Code** — versions in IaC, charts and manifests
- **Update Configuration** — values, flags, feature gates, limits
- **Open a Merge Request** — every change is reviewed before it ships
- **Pass CI Checks** — tests, lint, policy and security scans

```
values.yaml
- version: 1.14.2
+ version: 1.15.0
```

3

Deploy

roll it out safely

- **File a Change Ticket** — approval and a change window — prod only
- **Provision New Resources** — node pools, IAM roles, storage — if needed
- **Deploy the Change** — merge; the pipeline or GitOps applies it
- **Watch the Rollout** — readiness, events and error rates

Production follows its ticket
Changes go out inside the approved change window, linked to the MRs.

4

Validate

prove it worked, or undo it

- **Run Post-Validation Tests** — health checks, smoke tests, metrics
- **Notify Stakeholders** — owners and channels hear the outcome
- **Close the Change Ticket** — record what happened — prod only

IF VALIDATION FAILS

- **Roll Back the Change** — revert to the last good version
- **Investigate Anomalies** — logs, events, dashboards, then fix

rollout stops: nothing is promoted until it passes

Repeat Per Environment

each environment runs all four phases; the version moves on only after it validates

Dev

first and lowest risk: catch problems early

Plan › Code › Deploy › Validate

validated

same version

Staging

production-like: rehearse the exact change

Plan › Code › Deploy › Validate

validated

same version

Production

+ change ticket

customer-facing: ticketed, inside a change window

Plan › Code › Deploy › Validate

TAKEAWAY — Plan once, pin the version, and promote it one environment at a time. A failed validation stops the rollout where it is.

repeat in the next environment: dev → staging → production

1 Plan
know exactly what is changing

- **Run Pre-Validation Checks** — health, current versions, capacity
- **Review Breaking Changes** — removed APIs, renamed flags, migrations
- **Check Compatibility** — platform version, dependencies, CRDs
- **Read the Release Notes** — every release between current and target

Pick the target version once
It is pinned and promoted unchanged through dev, staging and production.

2 Code
describe the change as code

- **Update Infrastructure Code** — versions in IaC, charts and manifests
- **Update Configuration** — values, flags, feature gates, limits
- **Open a Merge Request** — every change is reviewed before it ships
- **Pass CI Checks** — tests, lint, policy and security scans

```
values.yaml
- version: 1.14.2
+ version: 1.15.0
```

3 Deploy
roll it out safely

- **File a Change Ticket** — approval and a change window — prod only
- **Provision New Resources** — node pools, IAM roles, storage — if needed
- **Deploy the Change** — merge; the pipeline or GitOps applies it
- **Watch the Rollout** — readiness, events and error rates

Production follows its ticket
Changes go out inside the approved change window, linked to the MRs.

4 Validate
prove it worked, or undo it

- **Run Post-Validation Tests** — health checks, smoke tests, metrics
- **Notify Stakeholders** — owners and channels hear the outcome
- **Close the Change Ticket** — record what happened — prod only

IF VALIDATION FAILS

- **Roll Back the Change** — revert to the last good version
- **Investigate Anomalies** — logs, events, dashboards, then fix

rollout stops: nothing is promoted until it passes

Repeat Per Environment
each environment runs all four phases; the version moves on only after it validates

Dev
first and lowest risk: catch problems early

Plan › Code › Deploy › Validate

validated
same version

Staging
production-like: rehearse the exact change

Plan › Code › Deploy › Validate

validated
same version

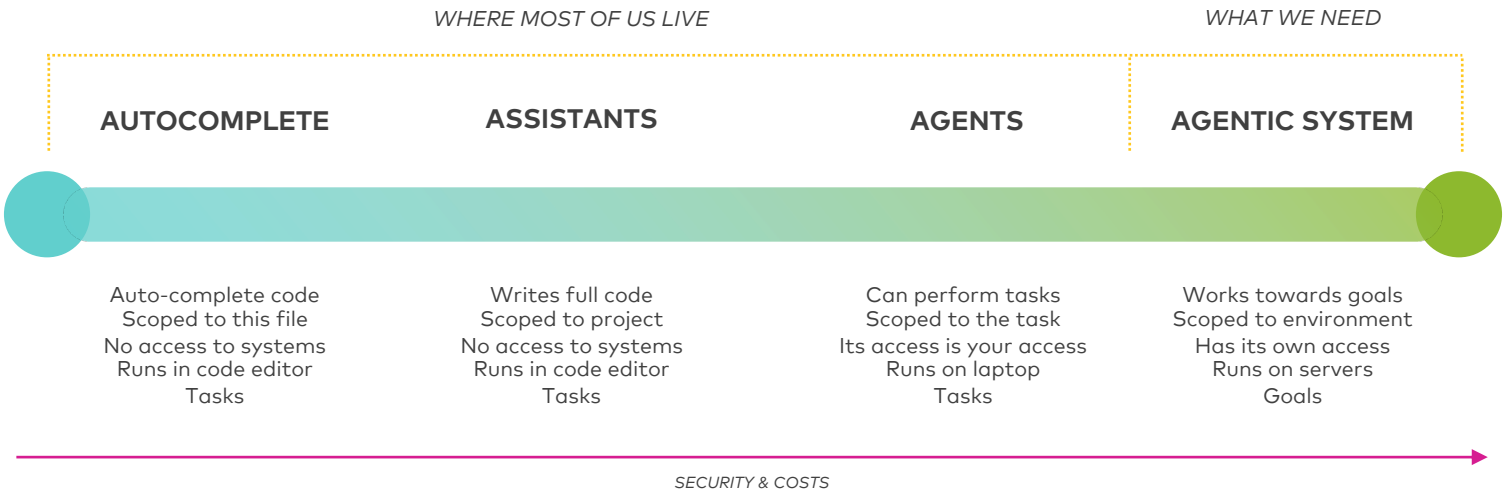
Production
+ change ticket

customer-facing: ticketed, inside a change window

Plan › Code › Deploy › Validate

TAKEAWAY Plan once, pin the version, and promote it one environment at a time. A failed validation stops the rollout where it is.

KubeCon CloudNativeCon North America 2026


WHERE MOST OF US LIVE
WHAT WE NEED
AUTOCOMPLETE
ASSISTANTS
AGENTS
AGENTIC SYSTEM
Auto-complete code
Scoped to this file
No access to systems
Runs in code editor
Tasks
Writes full code
Scoped to project
No access to systems
Runs in code editor
Tasks
Can perform tasks
Scoped to the task
Its access is your access
Runs on laptop
Tasks
Works towards goals
Scoped to environment
Has its own access
Runs on servers
Goals
SECURITY & COSTS

Data Problems

01 Infrastructure deals with time sensitive data that's streaming.

02 Typically, its very large amounts of data.

03 Need the ability for strong search, transformation and filtering capabilities.



Security Problems

01 An infrastructure agent can access much more sensitive systems.

02 Infrastructure agents typically access a lot more systems too.

03 Access needs to follow a strict least privileged system for production access.

04 Writes and changes should always have a human in the loop.

05 Should have a very clear auditable way of examining what an agent does.

06 Executing code in a shell is a non-negotiable from security.



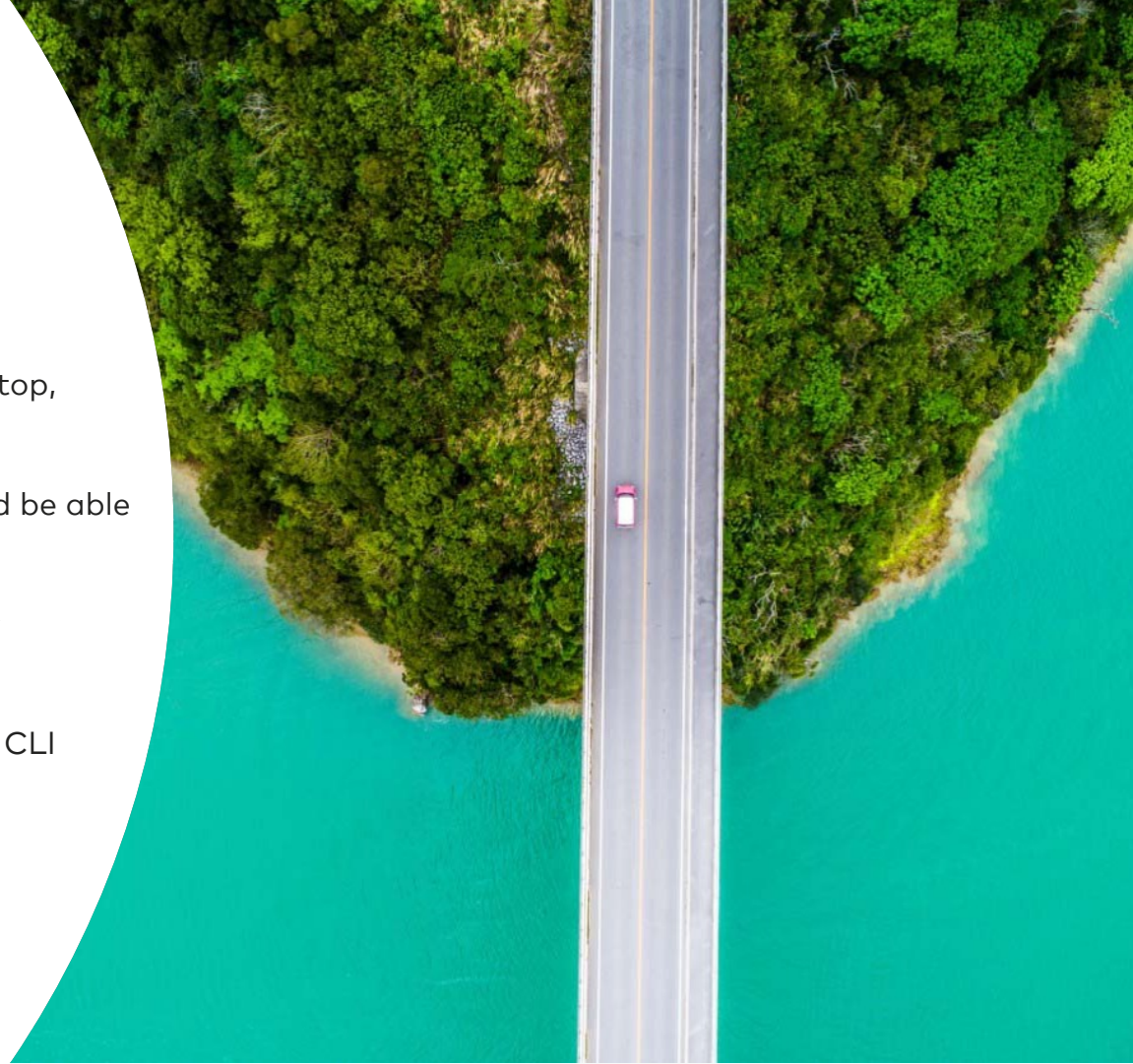
Runtime Problems

01 Agents should run outside of my laptop, on its own server.

02 Must be able to survive a restart and be able to resume a task.

03 Should be able to run on an event or schedule, not a user input

04 There's no way of approving shell or CLI commands executed by the agent.



Cost Problems

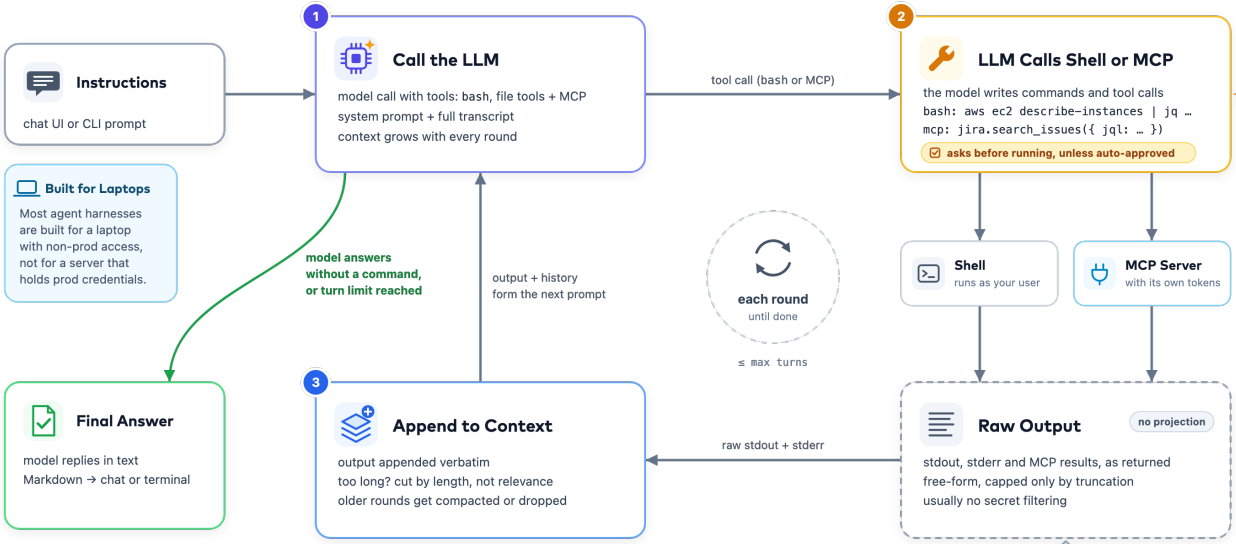
01 Large datasets mean expensive RAG bills and busted context windows

02 Agents running outside a laptop requires budgets and attribution.

03 Stricter token budget and strategies for minimizing data are needed.



Part 2: Agent Loops



Raw Output, Up Close

With no projection, whatever a command or MCP tool returns is appended to the context as-is, up to a size cap.

```
mcp jira.search_issues({"jql": "project = OPS"})
{"issues":[{"key":"OPS-1412","fields":{"summary":"…","comment":{"comments":[…]}}},…
… 1.8 MB more
$ cat .env
cat: .env: Permission denied
$ kubectl get secret db-creds -o yaml | grep password
  password: aHVudGVyMg==
```

Every byte goes into the context: megabytes of JSON, or a blind cut after a fixed number of characters.

The file sandbox blocks .env …

… but the same secret arrives base64-encoded, and the pattern filter lets it through.

Sandboxing the Shell

Each layer narrows the risk; none closes it.

Scoped, Short-Lived Tokens
Least-privilege credentials that expire in minutes shrink the blast radius.
A leaked token still works until it expires, and scopes tend to grow.

Pre-Ingestion Filtering
Scrub secrets, passwords and tokens from output before the model reads it.
Patterns miss base64, split lines and formats no rule anticipated.

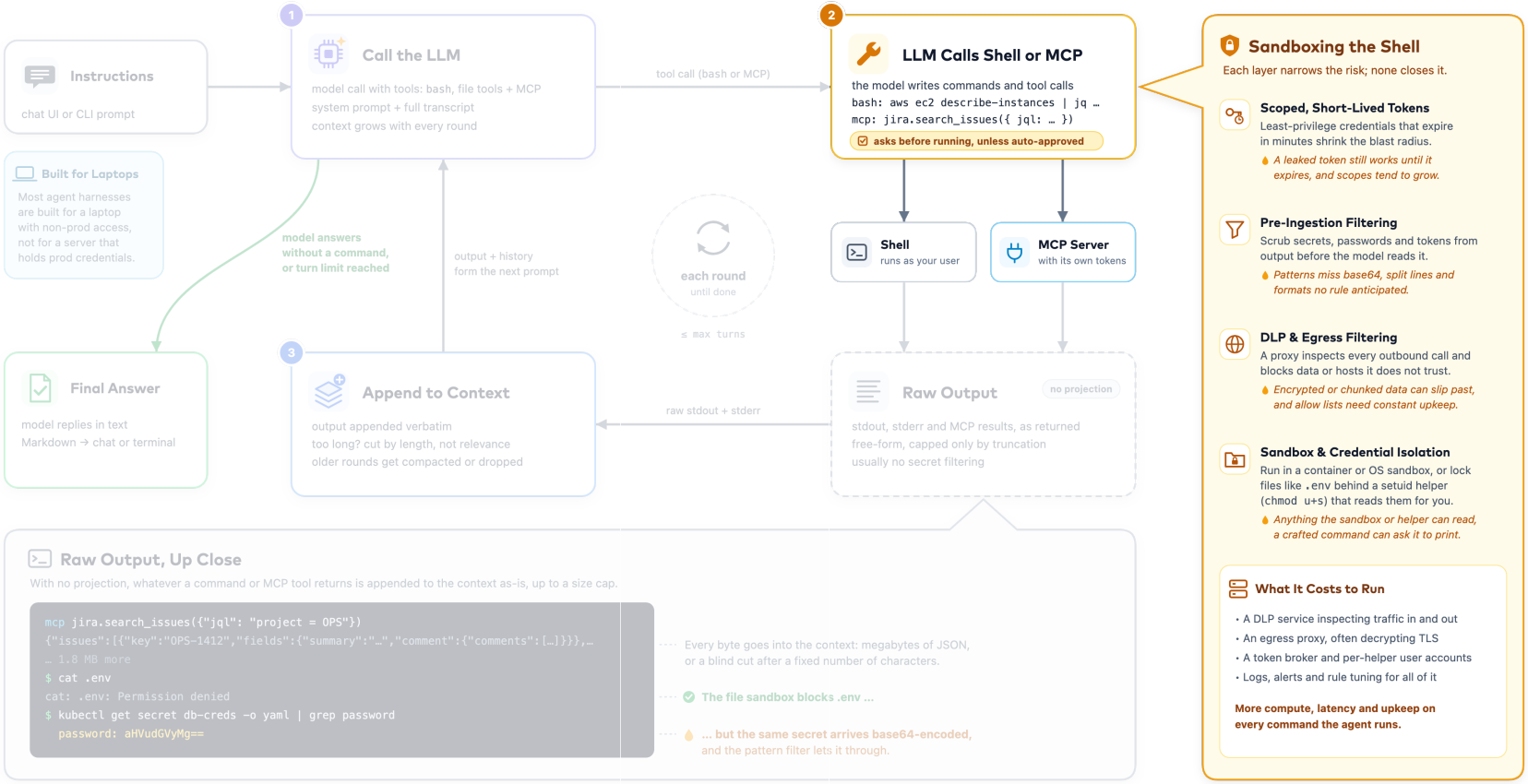
DLP & Egress Filtering
A proxy inspects every outbound call and blocks data or hosts it does not trust.
Encrypted or chunked data can slip past, and allow lists need constant upkeep.

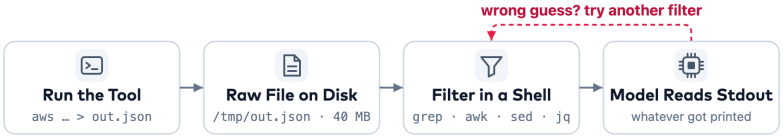
Sandbox & Credential Isolation
Run in a container or OS sandbox, or lock files like `.env` behind a setuid helper (`chmod u+s`) that reads them for you.
Anything the sandbox or helper can read, a crafted command can ask it to print.

What It Costs to Run

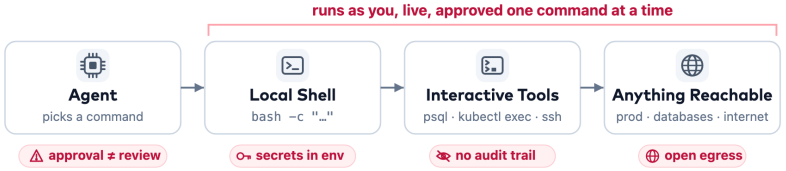
- A DLP service inspecting traffic in and out
- An egress proxy, often decrypting TLS
- A token broker and per-helper user accounts
- Logs, alerts and rule tuning for all of it

More compute, latency and upkeep on every command the agent runs.

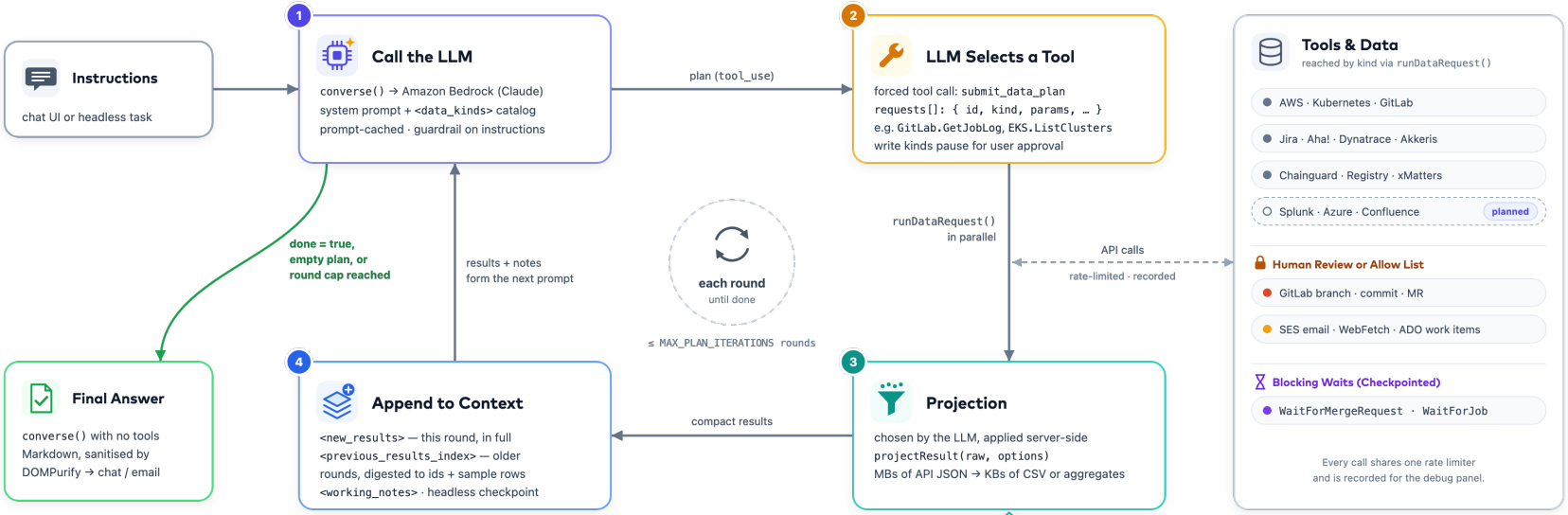

Instructions
chat UI or CLI prompt
Built for Laptops
Most agent harnesses are built for a laptop with non-prod access, not for a server that holds prod credentials.
1
Call the LLM
model call with tools: bash, file tools + MCP
system prompt + full transcript
context grows with every round
tool call (bash or MCP)
2
LLM Calls Shell or MCP
the model writes commands and tool calls
bash: aws ec2 describe-instances | jq …
mcp: jira.search_issues({ jql: … })
asks before running, unless auto-approved
Shell
runs as your user
MCP Server
with its own tokens
model answers without a command, or turn limit reached
output + history form the next prompt
each round
until done
≤ max turns
Final Answer
model replies in text
Markdown → chat or terminal
3
Append to Context
output appended verbatim
too long? cut by length, not relevance
older rounds get compacted or dropped
raw stdout + stderr
Raw Output
no projection
stdout, stderr and MCP results, as returned
free-form, capped only by truncation
usually no secret filtering
Raw Output, Up Close
With no projection, whatever a command or MCP tool returns is appended to the context as-is, up to a size cap.
mcp jira.search_issues({"jql": "project = OPS"})
{"issues":[{"key":"OPS-1412","fields":{"summary":"…","comment":{"comments":[…]}}},…
… 1.8 MB more
$ cat .env
cat: .env: Permission denied
$ kubectl get secret db-creds -o yaml | grep password
password: aHVudGVyMg==
Every byte goes into the context: megabytes of JSON, or a blind cut after a fixed number of characters.
The file sandbox blocks .env …
… but the same secret arrives base64-encoded, and the pattern filter lets it through.
Sandboxing the Shell
Each layer narrows the risk; none closes it.
Scoped, Short-Lived Tokens
Least-privilege credentials that expire in minutes shrink the blast radius.
A leaked token still works until it expires, and scopes tend to grow.
Pre-Ingestion Filtering
Scrub secrets, passwords and tokens from output before the model reads it.
Patterns miss base64, split lines and formats no rule anticipated.
DLP & Egress Filtering
A proxy inspects every outbound call and blocks data or hosts it does not trust.
Encrypted or chunked data can slip past, and allow lists need constant upkeep.
Sandbox & Credential Isolation
Run in a container or OS sandbox, or lock files like .env behind a setuid helper (chmod u+s) that reads them for you.
Anything the sandbox or helper can read, a crafted command can ask it to print.
What It Costs to Run
• A DLP service inspecting traffic in and out
• An egress proxy, often decrypting TLS
• A token broker and per-helper user accounts
• Logs, alerts and rule tuning for all of it
More compute, latency and upkeep on every command the agent runs.



- **A round trip per filter:** each grep, awk or jq pass is another shell command and another LLM turn.
- **Raw data on disk:** full payloads, secrets included, sit in files the next cat pulls into context.
- **Text tools guess at structure:** nested JSON, CSV quoting and multi-line logs break the pipe.
- **Unbounded output:** forget to pipe through `head` and thousands of lines flood the context window.



- **Direct execution:** commands run live on real systems; approving each one is not reviewing the change.
- **Secrets in reach:** env vars, `~/.aws`, kubeconfig and tokens are one `env` or `cat` away.
- **Interactive blind spots:** what happens inside psql, kubectl exec or ssh leaves no audit trail.
- **Network is hard to fence:** curl, package installs and DNS from a shell can reach any host.
- **Injection becomes execution:** instructions hidden in tool output can turn into commands.



Projection Interface

Options ride alongside `params` in each request and are stripped before the API call; the projector walks the raw response and returns a compact envelope.

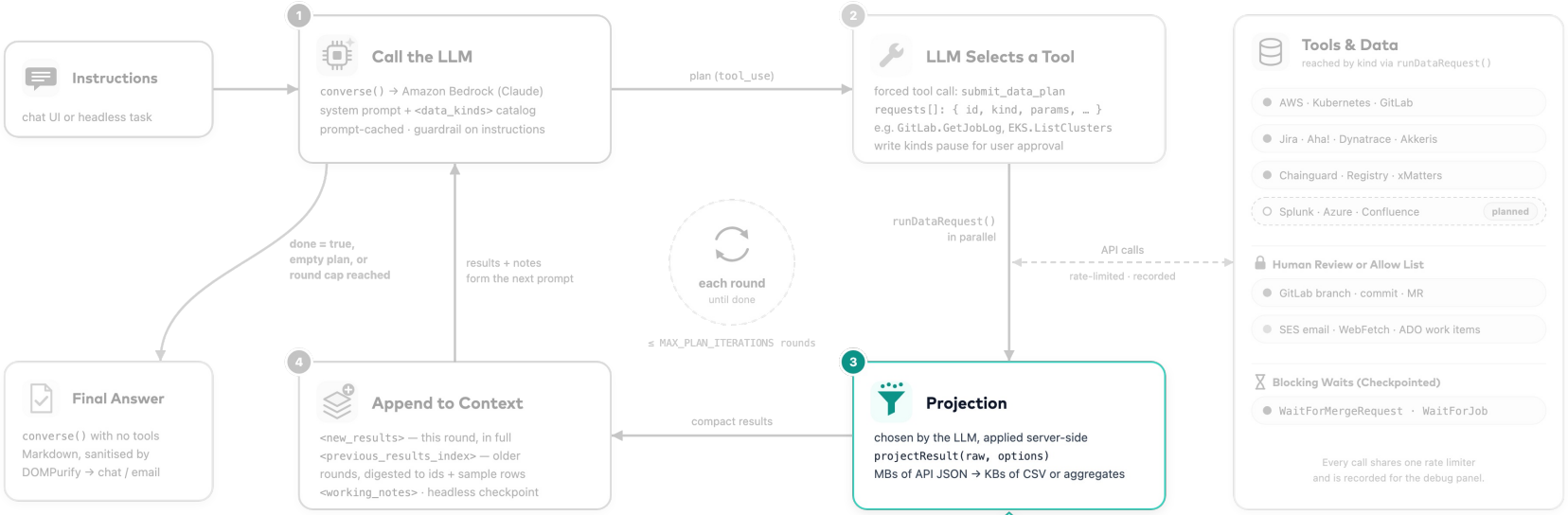
REQUEST OPTIONS

`fields?`	`string[]`	dotted paths; "[]" walks arrays, "[0]" indexes · up to 50
`where?`	`{ label: value }`	equals, in-list or glob "*prod*" · case-insensitive · up to 20
`groupBy?`	`string[]`	per-group results for count, sum, max and min
`mode?`	`"rows" | "count" | "sum" | "max" | "min"`	CSV rows (default), or just totals — the cheapest
`limit?`	`number`	each numeric field, overall and per group
`offset?`	`number`	row cap after filtering; off by default
		rows to skip before limit — page with offset + limit

ENVELOPE RETURNED TO THE LLM

`{ shape, total, returned?, truncated?, offset?, fields?, data }`

- csv → header row + one row per match, columns named after the fields
- count → { total, by_<field>: { value: n } } — a few hundred bytes
- sum / max / min → { rows, <mode>: { <field>: n }, by_<group>: { … } }
- AWS / Kubernetes: wrapped per role in { byRole: { "<roleArn>": envelope } }
- no fields → JSON-stringified raw payload (the safety net)



Projection Interface

Options ride alongside `params` in each request and are stripped before the API call; the projector walks the raw response and returns a compact envelope.

REQUEST OPTIONS

`fields?`	`string[]`	dotted paths; "[]" walks arrays, "[0]" indexes · up to 50
`where?`	`{ label: value }`	equals, in-list or glob "*prod*" · case-insensitive · up to 20
`groupBy?`	`string[]`	per-group results for count, sum, max and min
`mode?`	`"rows" | "count" | "sum" | "max" | "min"`	CSV rows (default), or just totals — the cheapest each numeric field, overall and per group
`limit?`	`number`	row cap after filtering; off by default
`offset?`	`number`	rows to skip before limit — page with offset + limit

ENVELOPE RETURNED TO THE LLM

`{ shape, total, returned?, truncated?, offset?, fields?, data }`

- csv → header row + one row per match, columns named after the fields
- count → { total, by_<field>: { value: n } } — a few hundred bytes
- sum / max / min → { rows, <mode>: { <field>: n }, by_<group>: { … } }
- AWS / Kubernetes: wrapped per role in { byRole: { "<roleArn>": envelope } }
- no fields → JSON-stringified raw payload (the safety net)



Projection Interface

Options ride alongside `params` in each request and are stripped before the API call; the projector walks the raw response and returns a compact envelope.

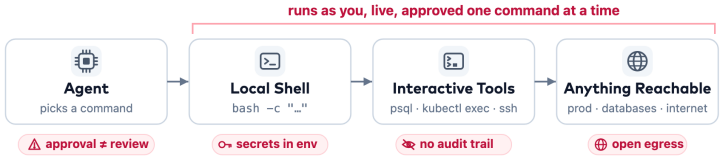
REQUEST OPTIONS

`fields?`	`string[]`	dotted paths; "[]" walks arrays, "[0]" indexes · up to 50
`where?`	`{ label: value }`	equals, in-list or glob "*prod*" · case-insensitive · up to 20
`groupBy?`	`string[]`	per-group results for count, sum, max and min
`mode?`	`"rows" | "count" | "sum" | "max" | "min"`	CSV rows (default), or just totals — the cheapest
`limit?`	`number`	each numeric field, overall and per group
`offset?`	`number`	row cap after filtering; off by default
		rows to skip before limit — page with offset + limit

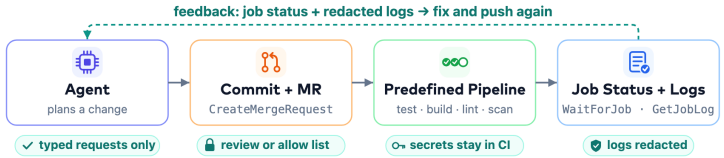
ENVELOPE RETURNED TO THE LLM

`{ shape, total, returned?, truncated?, offset?, fields?, data }`

- csv → header row + one row per match, columns named after the fields
- count → { total, by_<field>: { value: n } } — a few hundred bytes
- sum / max / min → { rows, <mode>: { <field>: n }, by_<group>: { … } }
- AWS / Kubernetes: wrapped per role in { byRole: { "<roleArn>": envelope } }
- no fields → JSON-stringified raw payload (the safety net)



- **Direct execution:** commands run live on real systems; approving each one is not reviewing the change.
- **Secrets in reach:** env vars, `~/.aws`, kubeconfig and tokens are one `env` or `cat` away.
- **Interactive blind spots:** what happens inside psql, kubectl exec or ssh leaves no audit trail.
- **Network is hard to fence:** curl, package installs and DNS from a shell can reach any host.
- **Injection becomes execution:** instructions hidden in tool output can turn into commands.



- **Merge requests, not commands:** every change is a commit on an MR — reviewed, versioned, revertible.
- **Predefined pipelines execute:** the MR runs the repo's own tests, builds, linters and security checks.
- **CI is the feedback loop:** the agent pulls job status and redacted job logs, then pushes a fix.
- **No shell, no secrets:** the model emits typed requests; credentials stay with the server and CI.
- **Fenced and auditable:** every call is recorded, writes need review or an allow list, only people merge.

Traditional Harness

The agent drives a shell: it filters output with text tools and runs commands itself

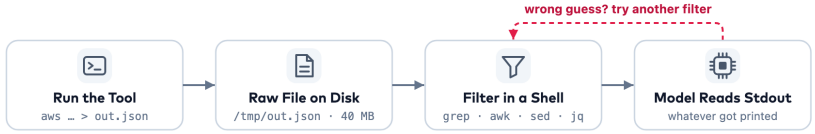
vs

This Harness

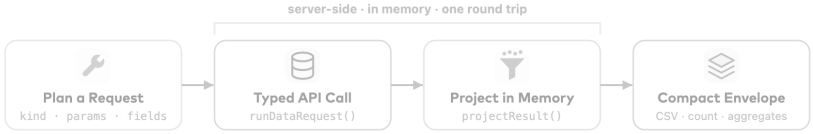
The agent sends typed requests: the server projects results and pipelines do the executing

1 Reducing Tool Output

how a large API response becomes the few facts the model needs



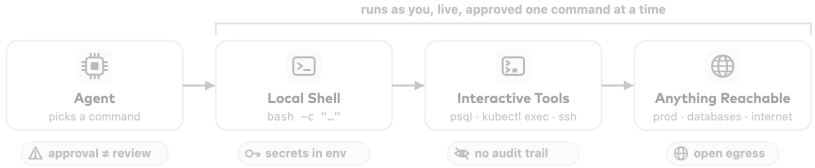
- **A round trip per filter:** each grep, awk or jq pass is another shell command and another LLM turn.
- **Raw data on disk:** full payloads, secrets included, sit in files the next cat pulls into context.
- **Text tools guess at structure:** nested JSON, CSV quoting and multi-line logs break the pipe.
- **Unbounded output:** forget to pipe through head and thousands of lines flood the context window.



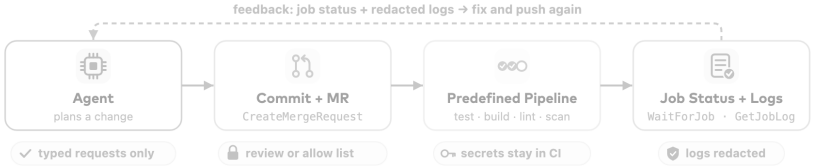
- **Declared up front:** fields, where, groupBy, mode, limit and offset ride along with the request.
- **Reduced server-side:** the response is projected in memory, not dumped to a file and read back.
- **Structure-aware and exact:** paths walk arrays; counts, sums, max and min are computed, not guessed.
- **Bounded by design:** capped fields and filters, explicit paging, and a truncated flag when rows remain.

2 Executing Actions

how the agent changes things, and how it learns whether the change worked



- **Direct execution:** commands run live on real systems; approving each one is not reviewing the change.
- **Secrets in reach:** env vars, ~/.aws, kubeconfig and tokens are one env or cat away.
- **Interactive blind spots:** what happens inside psql, kubectl exec or ssh leaves no audit trail.
- **Network is hard to fence:** curl, package installs and DNS from a shell can reach any host.
- **Injection becomes execution:** instructions hidden in tool output can turn into commands.



- **Merge requests, not commands:** every change is a commit on an MR — reviewed, versioned, revertible.
- **Predefined pipelines execute:** the MR runs the repo's own tests, builds, linters and security checks.
- **CI is the feedback loop:** the agent pulls job status and redacted job logs, then pushes a fix.
- **No shell, no secrets:** the model emits typed requests; credentials stay with the server and CI.
- **Fenced and auditable:** every call is recorded, writes need review or an allow list, only people merge.

TAKEAWAY **A shell harness** trusts the model with a terminal and your credentials. **This harness** keeps it on typed requests and lets reviewed, predefined pipelines do the executing.

KubeCon CloudNativeCon North America 2026

Traditional Harness

The agent drives a shell: it filters output with text tools and runs commands itself

vs

This Harness

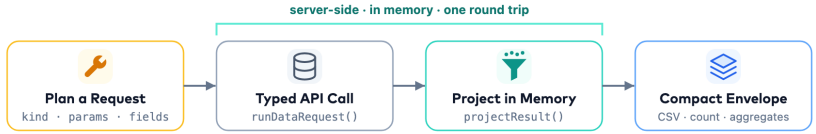
The agent sends typed requests: the server projects results and pipelines do the executing

1 Reducing Tool Output

how a large API response becomes the few facts the model needs



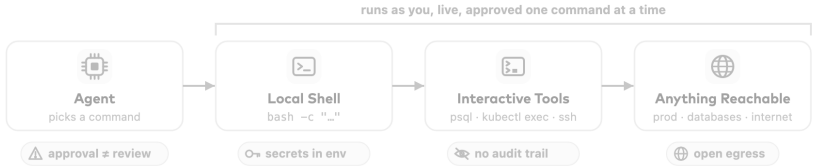
- **A round trip per filter:** each grep, awk or jq pass is another shell command and another LLM turn.
- **Raw data on disk:** full payloads, secrets included, sit in files the next cat pulls into context.
- **Text tools guess at structure:** nested JSON, CSV quoting and multi-line logs break the pipe.
- **Unbounded output:** forget to pipe through head and thousands of lines flood the context window.



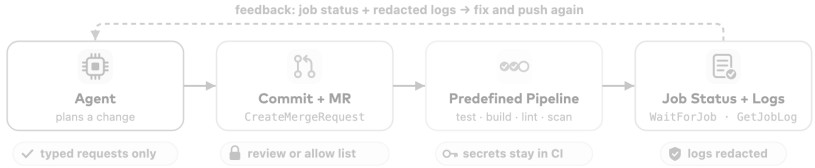
- **Declared up front:** fields, where, groupBy, mode, limit and offset ride along with the request.
- **Reduced server-side:** the response is projected in memory, not dumped to a file and read back.
- **Structure-aware and exact:** paths walk arrays; counts, sums, max and min are computed, not guessed.
- **Bounded by design:** capped fields and filters, explicit paging, and a truncated flag when rows remain.

2 Executing Actions

how the agent changes things, and how it learns whether the change worked



- **Direct execution:** commands run live on real systems; approving each one is not reviewing the change.
- **Secrets in reach:** env vars, `~/.aws`, kubeconfig and tokens are one `env` or `cat` away.
- **Interactive blind spots:** what happens inside psql, kubectl exec or ssh leaves no audit trail.
- **Network is hard to fence:** curl, package installs and DNS from a shell can reach any host.
- **Injection becomes execution:** instructions hidden in tool output can turn into commands.



- **Merge requests, not commands:** every change is a commit on an MR — reviewed, versioned, revertible.
- **Predefined pipelines execute:** the MR runs the repo's own tests, builds, linters and security checks.
- **CI is the feedback loop:** the agent pulls job status and redacted job logs, then pushes a fix.
- **No shell, no secrets:** the model emits typed requests; credentials stay with the server and CI.
- **Fenced and auditable:** every call is recorded, writes need review or an allow list, only people merge.

TAKEAWAY **A shell harness** trusts the model with a terminal and your credentials. **This harness** keeps it on typed requests and lets reviewed, predefined pipelines do the executing.

KubeCon CloudNativeCon North America 2026

Part 3: Decisions

Where RAG Breaks Down...

- "How many", "list all" or other numeric questions fail as RAG only returns a top few in its result set ("How many EC2s do we have running?" fails).
- Multi-step questions fail, answers that need a lookup to find the answer fail (pull the pods from staging clusters).
- Permissions around data access get a bit muddled and difficult to manage.
- A bad answer can be ingestion, chunking, embedding, ranking and a lot of other issues.



Why RAG Fails on Infra Data

- Vectorizing logs and metrics in a RAG system is expensive (each vectorization + LLM cost).
- Delays in data in RAG pipelines cause data to become irrelevant (15-minute pipeline is too late for our use case).
- Hallucinations in RAG due to pulling wrong data chunk, ranking irrelevant content to high, fragmented data.
- RAG is great for policy docs, not great for streaming data.



Projections instead of RAG

- With projections, large datasets became much smaller result sets to operate on.
- Remember this: “How many EC2s are running?” calls ec2.DescribeInstances and returns 40MB of JSON.
- Projections on it can count and group it reducing 40MB to 800 bytes.
- RAG based solutions fail to answer these questions.



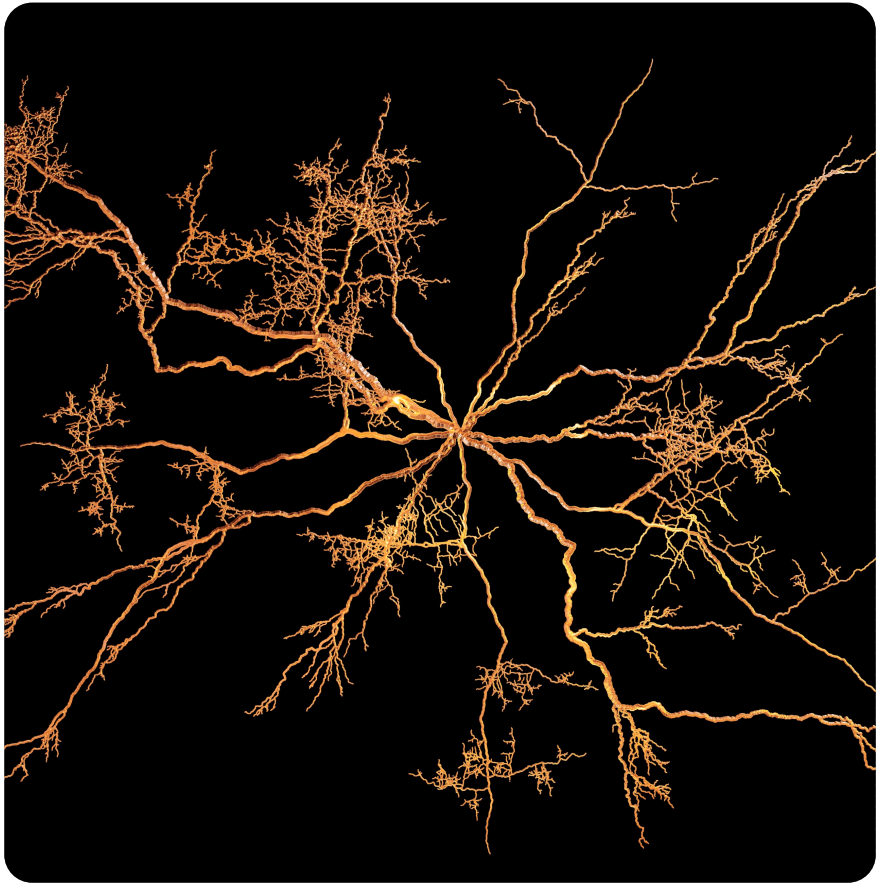
Ditch MCPs, adopt tools

- Rely on built-in functions, not MCP.
- Very transparent what the agent has (and WILL have) access to.
- Ability to filter out data from API calls.
- Special post processing filters/options so agents can remove data.
- Don't send specs to agents, functions match the native API allowing agents to use existing training. Keeps the context window low!
- Keys will never be revealed.



Other Security Principles

- Allow list for website URL requests.
- Allow list for sending emails.
- Use function calls to keep clear access privileges & audit logs, keep secrets app side.
- Require all write to go through merge requests.
- Require approval & review of instructions going into agents.
- Rate limit outgoing calls.



Extra Guardrail Security

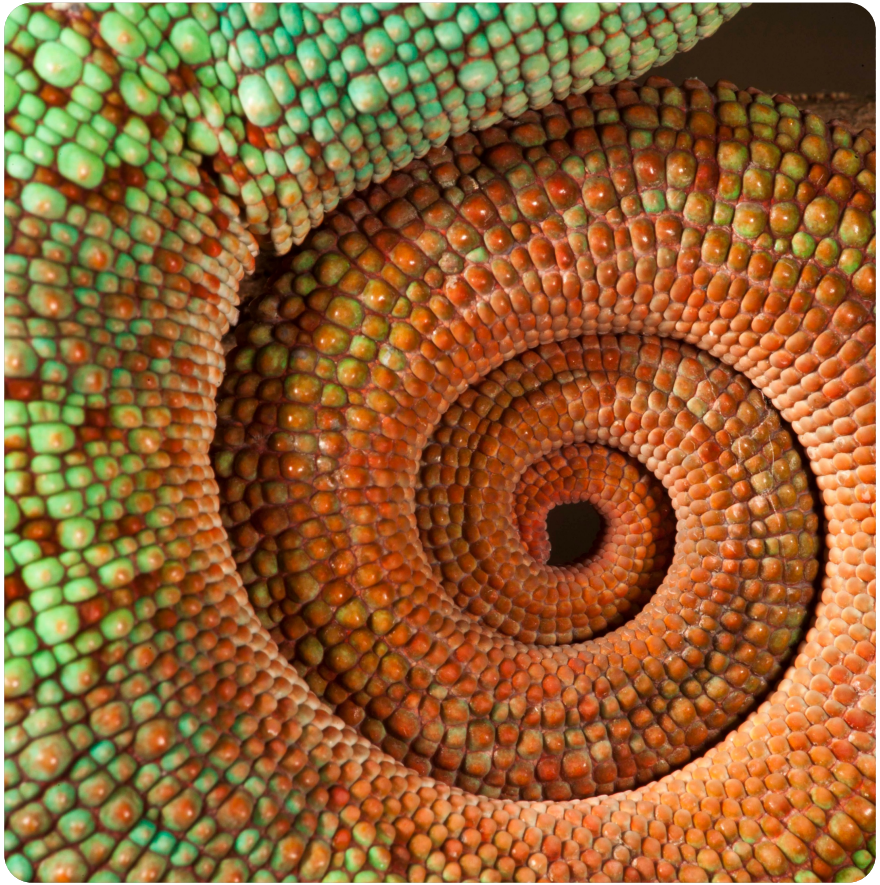
- Ensured instructions and information are clearly marked going to an LLM.
- Force information to go through tight guardrails for prompt injection.
- Rely on AWS Guardrails built-in profanity, hate, sexual content filters.
- Custom filters in bedrock guardrails for secret filtering (PEM regex, AWS access key regex, etc.).
- Log everything bedrock receives and its responses in an encrypted S3.



Part 4: Results

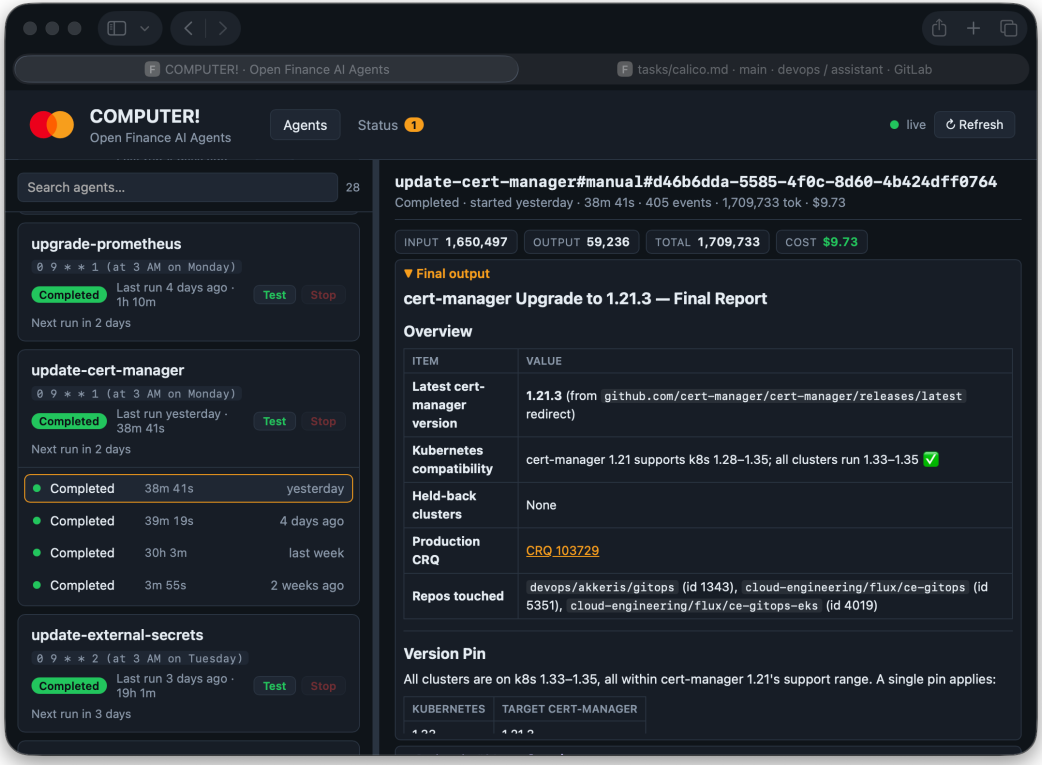
What We Built: COMPUTER!

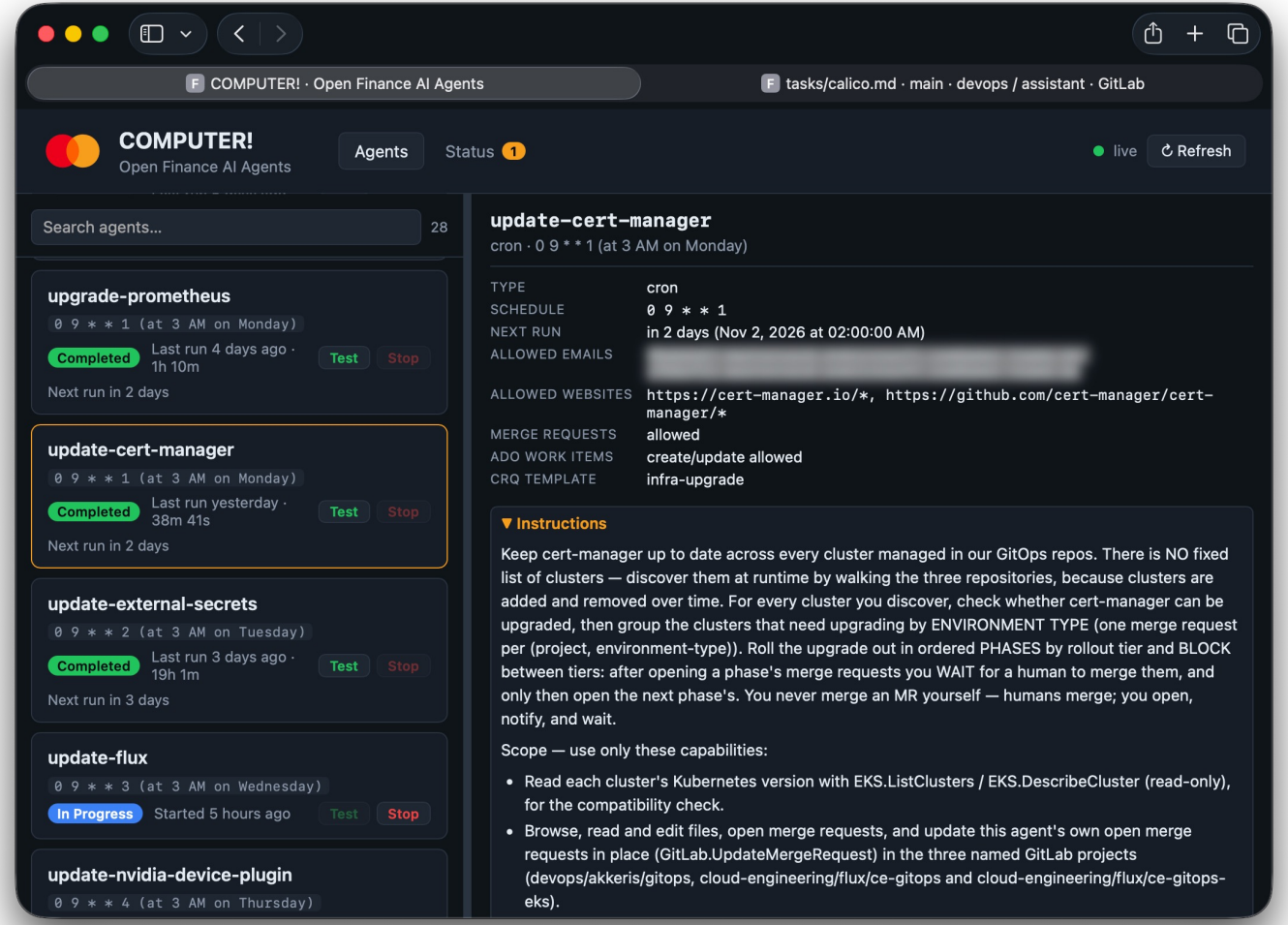
- A node.js deployable that connects with AWS Bedrock
- Is a single deployable 12-factor app with a clean security boundary
- Uses postgres to store state, store auditable history and resume tasks.
- Supports all the fancy features of context compression, sub-agents, etc.

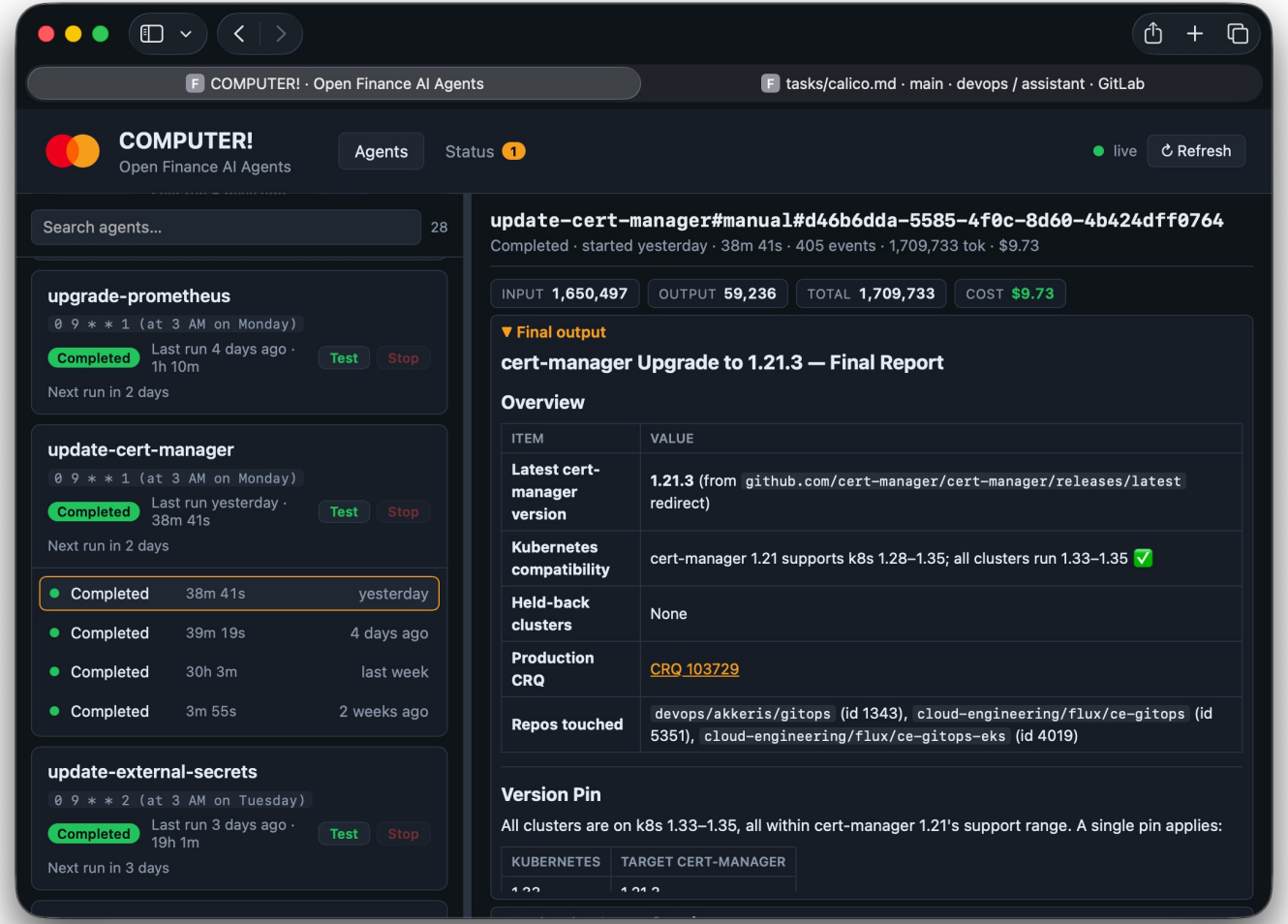


What It Delivered

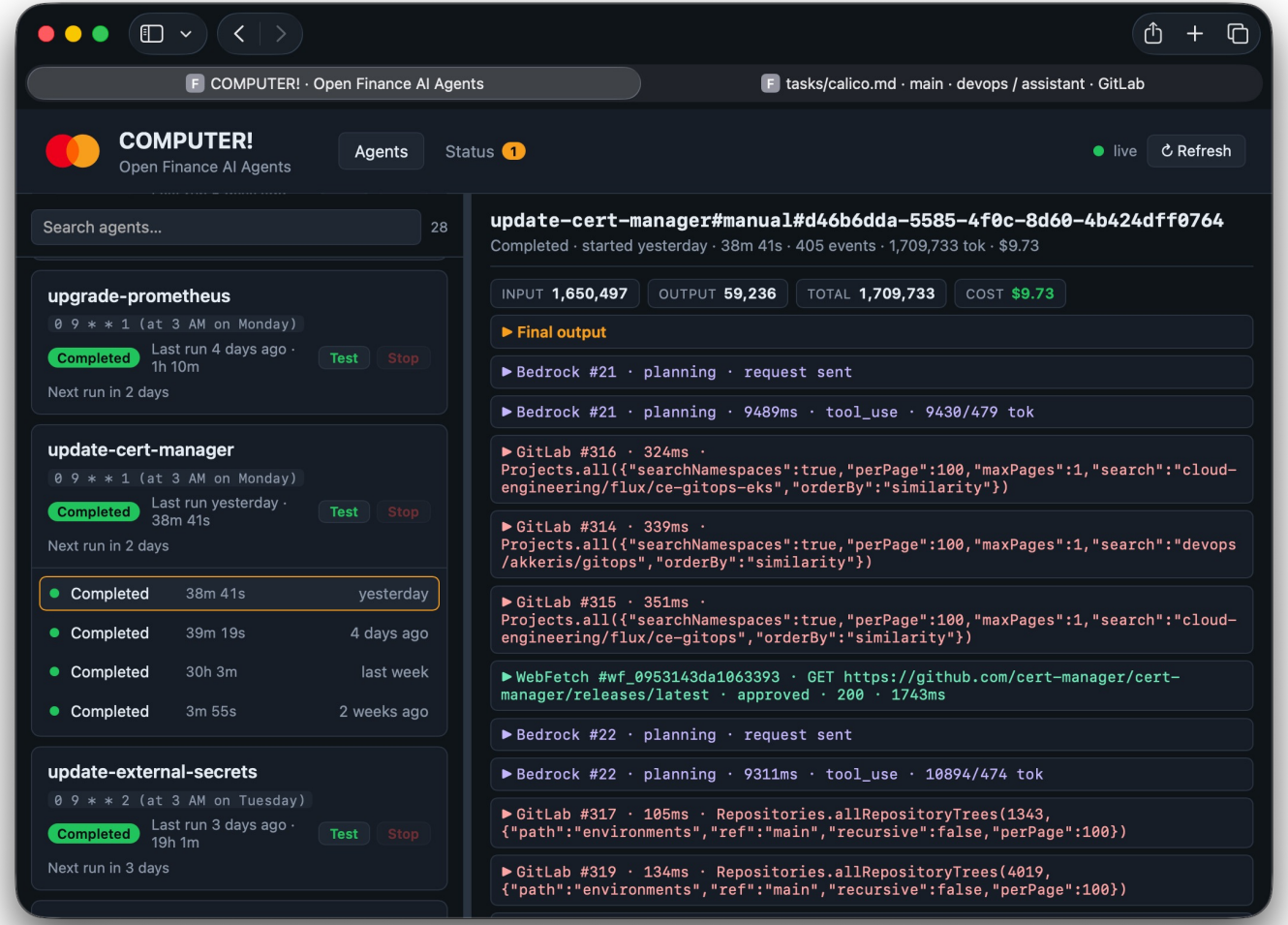
- 25 Infrastructure systems being upgraded weekly.
- Upgrading these systems across 30 production and 120 non-production EKS clusters.
- Vulnerabilities are under control & teams have capacity back.
- One upgrade that took weeks, now takes 8 hours.
- Finding other use cases… Like Automatic pod crash analysis.



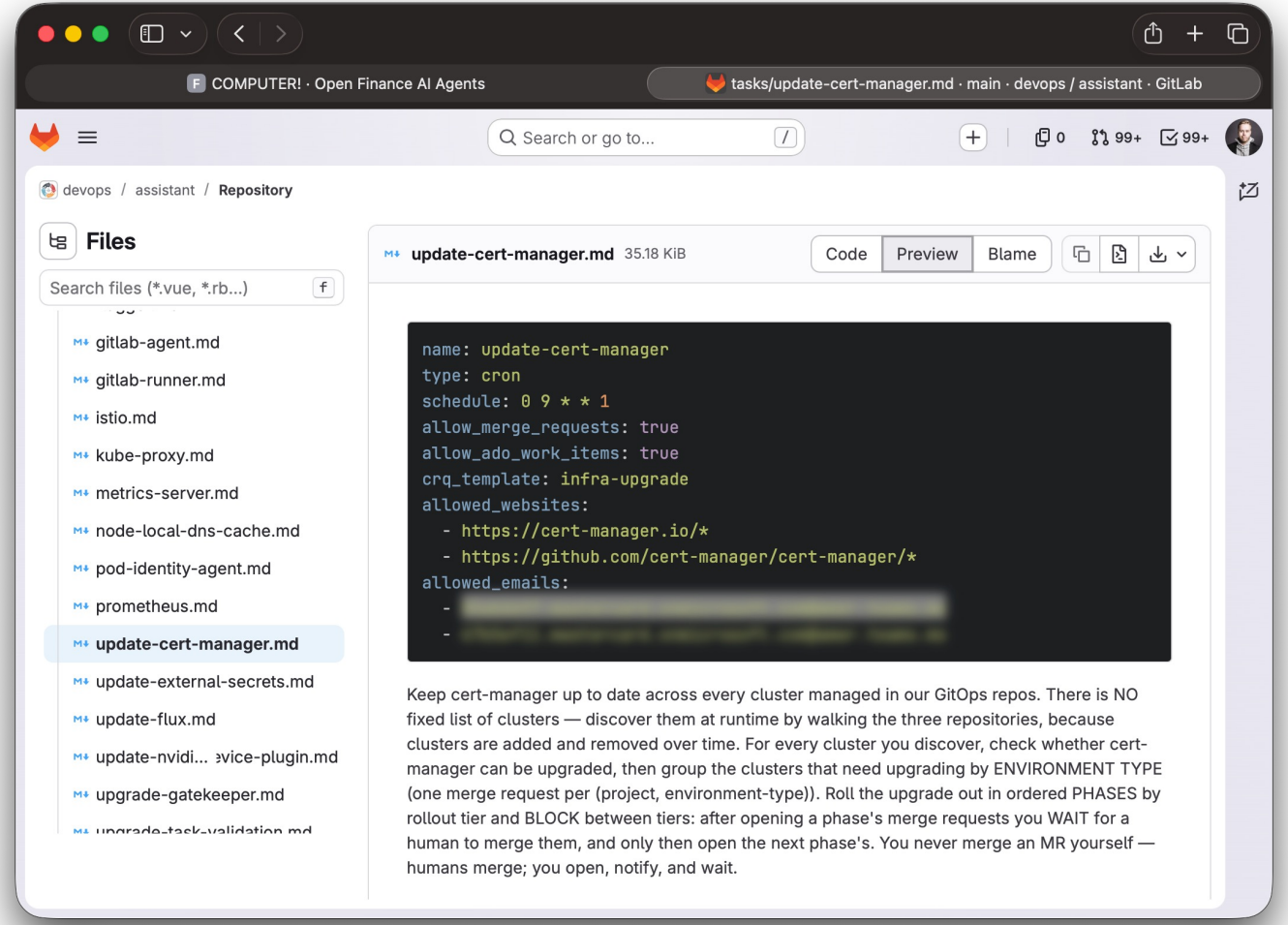

COMPUTER! · Open Finance AI Agents
tasks/calico.md · main · devops / assistant · GitLab
COMPUTER!
Open Finance AI Agents
Agents
Status 1
live
Refresh
Search agents...
28
upgrade-prometheus
0 9 * * 1 (at 3 AM on Monday)
Completed
Last run 4 days ago · 1h 10m
Test
Stop
Next run in 2 days
update-cert-manager
0 9 * * 1 (at 3 AM on Monday)
Completed
Last run yesterday · 38m 41s
Test
Stop
Next run in 2 days
update-external-secrets
0 9 * * 2 (at 3 AM on Tuesday)
Completed
Last run 3 days ago · 19h 1m
Test
Stop
Next run in 3 days
update-flux
0 9 * * 3 (at 3 AM on Wednesday)
In Progress
Started 5 hours ago
Test
Stop
update-nvidia-device-plugin
0 9 * * 4 (at 3 AM on Thursday)
update-cert-manager
cron · 0 9 * * 1 (at 3 AM on Monday)
TYPE cron
SCHEDULE 0 9 * * 1
NEXT RUN in 2 days (Nov 2, 2026 at 02:00:00 AM)
ALLOWED EMAILS
ALLOWED WEBSITES https://cert-manager.io/*, https://github.com/cert-manager/cert-manager/*
MERGE REQUESTS allowed
ADO WORK ITEMS create/update allowed
CRQ TEMPLATE infra-upgrade
▼ Instructions
Keep cert-manager up to date across every cluster managed in our GitOps repos. There is NO fixed list of clusters — discover them at runtime by walking the three repositories, because clusters are added and removed over time. For every cluster you discover, check whether cert-manager can be upgraded, then group the clusters that need upgrading by ENVIRONMENT TYPE (one merge request per (project, environment-type)). Roll the upgrade out in ordered PHASES by rollout tier and BLOCK between tiers: after opening a phase's merge requests you WAIT for a human to merge them, and only then open the next phase's. You never merge an MR yourself — humans merge; you open, notify, and wait.
Scope — use only these capabilities:
Read each cluster's Kubernetes version with EKS.ListClusters / EKS.DescribeCluster (read-only), for the compatibility check.
Browse, read and edit files, open merge requests, and update this agent's own open merge requests in place (GitLab.UpdateMergeRequest) in the three named GitLab projects (devops/akkeris/gitops, cloud-engineering/flux/ce-gitops and cloud-engineering/flux/ce-gitops-eks).



KubeCon CloudNativeCon North America 2026



KubeCon CloudNativeCon North America 2026


COMPUTER! · Open Finance AI Agents
tasks/update-cert-manager.md · main · devops / assistant · GitLab
Search or go to...
0
99+
99+
devops / assistant / Repository
Files
Search files (*.vue, *.rb...)
gitlab-agent.md
gitlab-runner.md
istio.md
kube-proxy.md
metrics-server.md
node-local-dns-cache.md
pod-identity-agent.md
prometheus.md
update-cert-manager.md
update-external-secrets.md
update-flux.md
update-nvidi... evice-plugin.md
upgrade-gatekeeper.md
update-cert-manager.md 35.18 KiB
Code
Preview
Blame
name: update-cert-manager
type: cron
schedule: 0 9 * * 1
allow_merge_requests: true
allow_ado_work_items: true
crq_template: infra-upgrade
allowed_websites:
- https://cert-manager.io/*
- https://github.com/cert-manager/cert-manager/*
allowed_emails:
Keep cert-manager up to date across every cluster managed in our GitOps repos. There is NO fixed list of clusters — discover them at runtime by walking the three repositories, because clusters are added and removed over time. For every cluster you discover, check whether cert-manager can be upgraded, then group the clusters that need upgrading by ENVIRONMENT TYPE (one merge request per (project, environment-type)). Roll the upgrade out in ordered PHASES by rollout tier and BLOCK between tiers: after opening a phase's merge requests you WAIT for a human to merge them, and only then open the next phase's. You never merge an MR yourself — humans merge; you open, notify, and wait.


DEMO TIME

Future Enhancements

- Self Improving - Ability to put in merge requests into its own repo to fix itself or its own instructions.
- Teams – Create different agent types (QA, architect, engineer) and allow them to collaborate on a problem.
- Communication – Allow different sub-agents to communicate among each other not just with the parent.



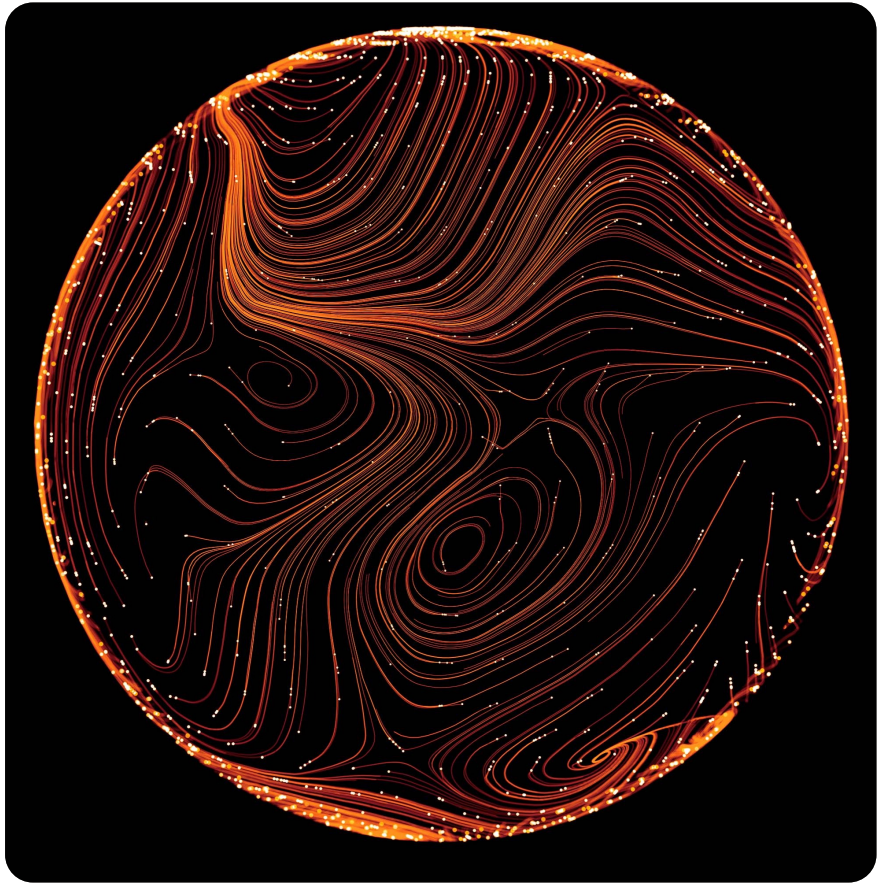
Culture Changes

- Be prepared to move fast.
- We outsource tasks, we do not outsource critical thinking.
- Don't focus on getting it done, focus on getting an agent to get it done.
- Dream big, challenge an agent to do more, it might surprise you.
- Partner with security early and often to ensure we operate with integrity.
- Always be aware of the costs.



So what?

- No shell, it's not necessary.
- Use projections over RAG.
- Tools should be functions, not MCPs.
- Use merge requests for all writes & its pipelines as feedback and to enforce what matters.



Thank You


github.com/trevorlinton


linkedin.com/in/trevor-linton


@trevorlinton.bsky.social


@trevorlinton