Beyond Prototype Scaling Framework-agnostic AI Agent Infrastructure with Ray

Bhumik Thakkar, Senior Engineer, Apple
Deepak Chandramouli, Senior ML Engineer, Apple

Agenda

Intro

Agent DevX & Ops — Vanilla

Agent DevX & Ops — With Ray

Call outs

What's Next

Agent anatomy

LLM

Tools

Memory

Prompt

Orchestrator

Observability

Safety

Meet the Trip-Planner

search_destinations

Suggests cities for a season

get_weather

Snapshot before outdoor advice

build_itinerary

Day-by-day plan from the catalogue

Tool Calling: Turning Reasoning into Action

Trip-planner trace

User: → "Plan a 3-day winter trip" tools=[search_destinations, …]

Model: → search_destinations(season=“winter”) # function call

User·System: → [{city:"Banff", why:"skiing"}, {city:"Reykjavik", …}]

Model: → get_weather(city=“Banff") # function call

User·System: → {temp:"-10°C", snow:true, visibility:”good"}

Model: → “Banff in winter: fresh powder. Here is a 3-day plan…"

Agent Dev Frameworks

Graph orchestration

Tool & function calling

Multi-agent coordination

Memory & checkpointing

Streaming & async flows

Vanilla DevX

Building the Agent

Define

```
@tool
def search_destinations(season: str) -> dict:
    """Suggest destinations for a season."""
    ...

@tool
def get_weather(city: str) -> dict:
    """Get weather forecast for a city."""
    ...

@tool
def build_itinerary(destination: str, days: int) -> dict:
    """Build a day-by-day itinerary."""
    ...
```

Compose

```
builder = StateGraph(State)
builder.add_node("chatbot", chatbot)
builder.add_node("tools", ToolNode(TRIP_TOOLS))
builder.add_conditional_edges("chatbot", tools_condition)
graph = builder.compile(checkpointer=MemorySaver())
```

Serve

```
@app.post("/agent")
async def chat(q: Query):
    return await graph.ainvoke({"messages": [...]})
```

Running it locally

Start

```
uvicorn \
trip_planner.app:app \
—reload \
- -port 8000
```

Query

```
curl -sX POST \
  http://localhost:8000/agent \
  -d '{"message":"Plan a 3-day winter trip",
       "thread_id":"demo"}'
```

Inspect trail


```
{“thread_id":"demo", "messages": [
  {"role":"HumanMessage",  "content":"Plan a 3-day winter trip"},
  {"role":"AIMessage",     "tool_calls":["search_destinations(winter)"]},
  {"role":"ToolMessage",   "content":"[Banff, Reykjavík, Lapland, ...]"},
  {"role":"AIMessage",     "tool_calls":["get_weather(Banff)", "build_itinerary(Banff,3)"]},
  {"role":"ToolMessage" ,  "content":"-10°C ... 3-day plan"},
  {"role":"AIMessage",     "content":"Banff at -10°C with fresh powder. Day 1: Lake Louise..."}
]}
```

Pushing to prod

Dockerfile

```
FROM python:3.11-slim
WORKDIR /app
COPY . .
RUN pip install -r requirements.txt
EXPOSE 8000
CMD ["uvicorn","app:app","--host","0.0.0.0"]
```

Deployment

```
apiVersion: apps/v1
kind: Deployment
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: agent
        image: registry/agent:v0.1
        ports: [{containerPort: 8000}]
```

What is still missing...

- Capacity
- Scheduling
- Scaling
- End to End observability
- ...

Requirements

Core & Resilience

Capacity & Scheduling

Right-size replicas and place pods on the right nodes

Traffic-aware autoscale

Scale on in-flight requests, not CPU%

Fault tolerance

Replica crash recovery without losing turns

Protocols

HTTP unary

POST /agent → JSON. Table stakes.

SSE token streaming

First-token latency matters as much as total

Long-running async

Submit + poll for 5–20 min reasoning

Operational

Agent observability

Tokens, turn depth, tool retries

Safety guardrails

Pre-filter before reasoning spends

Trace correlation

One ID across replicas, tools, model

Ray Serve

Ray Serve - One Substrate

Heterogeneous Resources

Scheduling & Orchestration

Traffic-aware Autoscaling

Native Streaming

Async Inference

Built-in Observability

FastAPI App

```
from fastapi import FastAPI
from trip_planner.graph import build_graph


agent = build_graph()

app = FastAPI(title="trip-planner", version="0.1.0")
@app.post("/agent")
async def chat(q: Query) -> dict:
  return await agent.ainvoke(
        {"messages": [{"role": "user", "content":
q.message}]},
        config=config,
    )
```

- Leverage the same FastAPI app
- Native support in Ray Serve
- gRPC support available too

Streaming & SSE

```python
@app.post("/agent/stream")
async def stream(q: Query) -> StreamingResponse:
    async def gen():
        async for chunk in agent.astream(
            {"messages": [{"role": "user", "content": q.message}]},
            config=config,
            stream_mode="messages",
        ):
            msg, meta = chunk
            payload = {
              "content": getattr(msg, "content", ""),
               "node": meta.get("langgraph_node") if meta else None,
            }
            yield f"data: {json.dumps(payload)}\n\n"
        yield "event: done\ndata: {}\n\n"
    return StreamingResponse(
        gen(),
        media_type="text/event-stream",
        headers={"X-Accel-Buffering": "no"},
    )
```

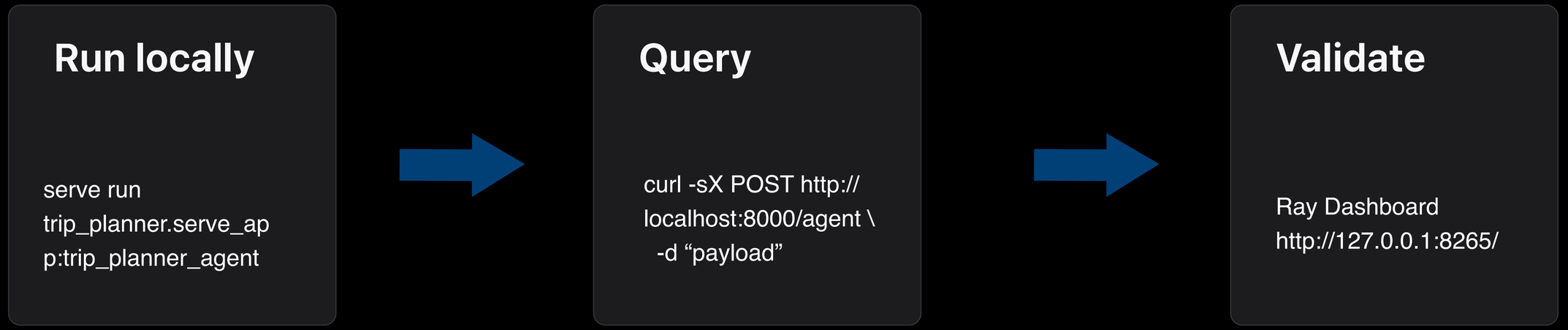
- Long running or multi step operations
- Native support
- Select based on use-case

Ray Serve Deployment

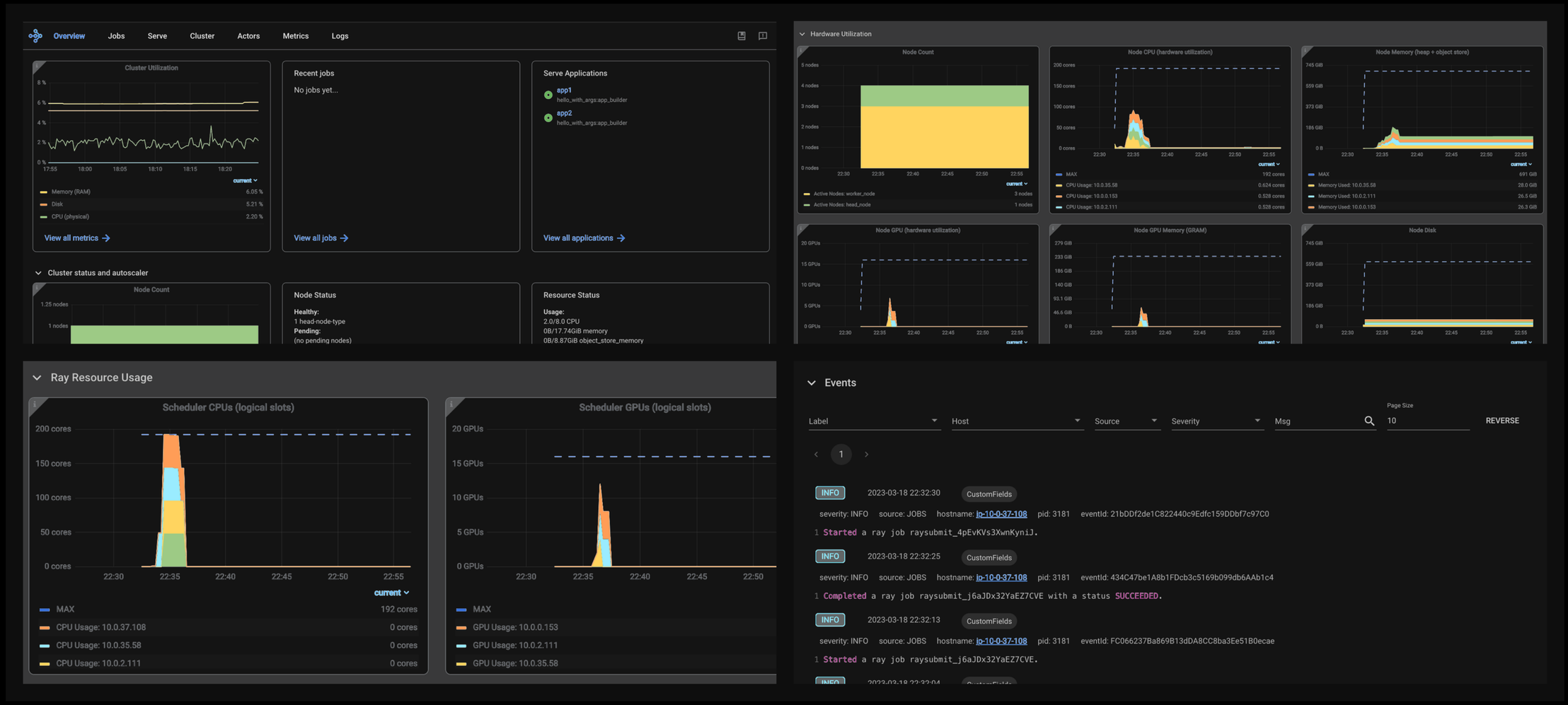
```python
from ray import serve
from ray.serve.config import AutoscalingConfig
from trip_planner.app import app

@serve.deployment(
    autoscaling_config=AutoscalingConfig(
        min_replicas=1,
        max_replicas=10,
        target_ongoing_requests=5,
    ),
    ray_actor_options={"num_cpus": 1},
    max_ongoing_requests=10,
    health_check_period_s=10,
    graceful_shutdown_timeout_s=120,
)
@serve.ingress(app)
class TripPlannerAgent: pass
trip_planner_agent = TripPlannerAgent.bind()
```

- Knobs for autoscaling, resources, load balancing and so on
- Supply the FastAPI app as the ingress
- Bind the deployment

Run, test and iterate locally



Ray Dashboard · End-to-End Observability



Source : https://docs.ray.io/en/latest/ray-observability/getting-started.html

Beyond Local Development

Current setup

- serve run: agent live in seconds
- Quick iterations and validations
- Live dashboard with logs & metrics

Production requirements

- Host Agent in Kubernates
- Handle auth, logs, metrics, secrets
- Zero-downtime releases
- Manage multiple agents

RayService CRD

The recipe for packaging everything in one file

- k8s spec for RayService Custom Resource Definition
- Uses Kuberay Operator for reconciliation
- Captures specs for:
  - RayServe
  - RayCluster
  - Init containers
  - Sidecars

RayService CRD

rayClusterConfig

```yaml
headGroupSpec:
  rayStartParams: {dashboard-host: "0.0.0.0"}
  template:
    spec:
      containers:
      - name: ray-head
        image: registry/trip-planner:2.x
workerGroupSpecs:
- groupName: agent-workers
  minReplicas: 4
  maxReplicas: 20
  template:
    spec:
      containers:
      - name: ray-worker
        image: registry/trip-planner:2.x
        resources:
          requests: {cpu: 2, memory: "8Gi"}
```

RayCluster Config

- RayService runs on RayCluster
- Specs for head and workers
- Contains GCS FT settings and more

RayService CRD

serveConfigV2

```yaml
# rayservice-agent.yaml

apiVersion: ray.io/v1
kind: RayService
metadata: {name: trip-planner}
spec:
  serveConfigV2: |
    applications:
    - name: trip-planner
      import_path: trip_planner.serve_app:trip_planner_agent
      deployments:
      - name: TripPlannerAgent
        autoscaling_config:
          min_replicas: 2
          max_replicas: 20
        graceful_shutdown_timeout_s: 120
```

Serve config v2

- Takes precedence over code
- Supports in-place updates

RayService CRD

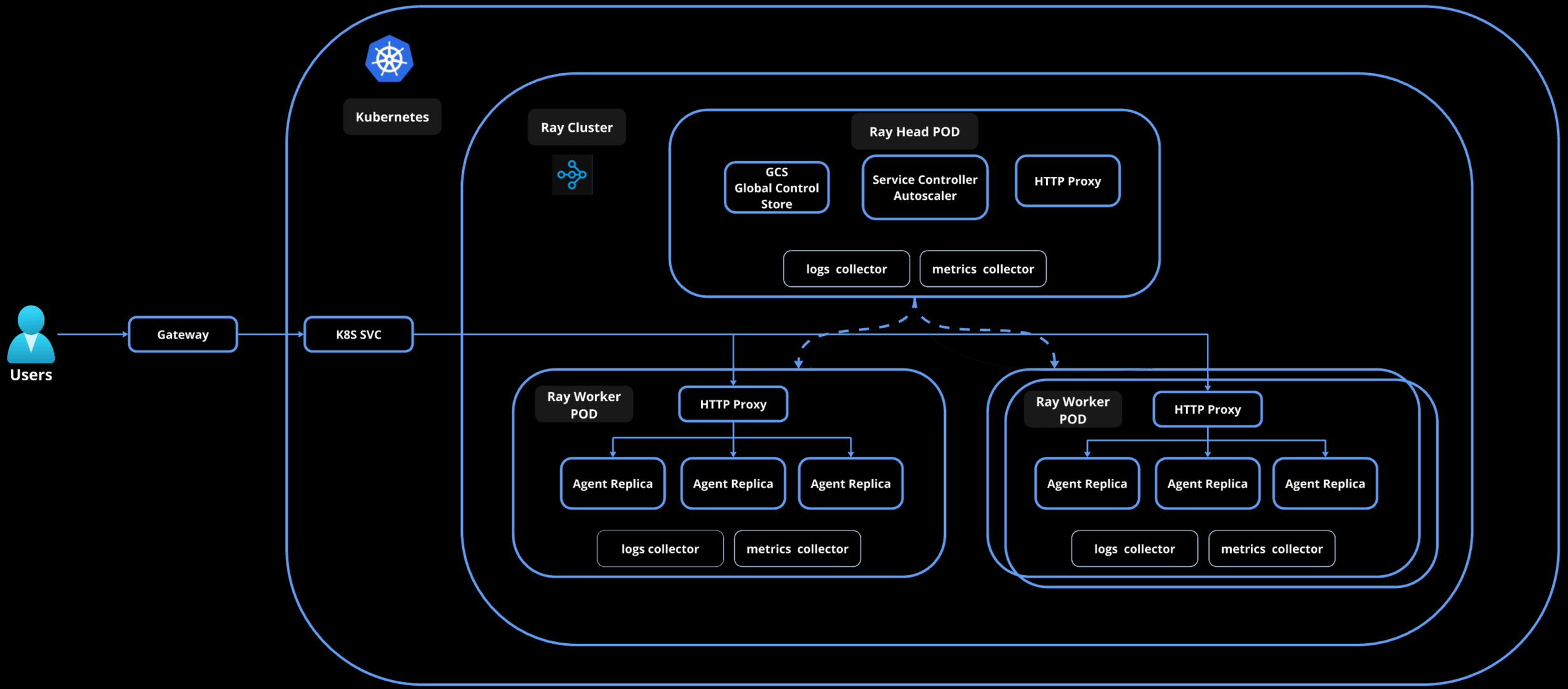
Init containers and sidecars

```
initContainers:
        - name: vault-client
          image: vault-client:1.x
          command: ["vault", "fetch", "secret/trip-planner", "/run/secrets/
agent.env"]
          volumeMounts:
          - {name: secrets, mountPath: /run/secrets}

 containers:
        - name: ray-head
          image: registry/trip-planner:2.x
          volumeMounts:
          - {name: secrets, mountPath: /run/secrets, readOnly: true}

        - name: log-collector
          image: log/log-collector:3.x
          volumeMounts:
          - {name: varlog, mountPath: /var/log}
```

Init containers and sidecars

- k8s native formats
- Offload items such as log collection, metrics emission, secrets mounting ...
- Available for both head and workers

Trip Planner - The Topology



Agent Management

Control Plane

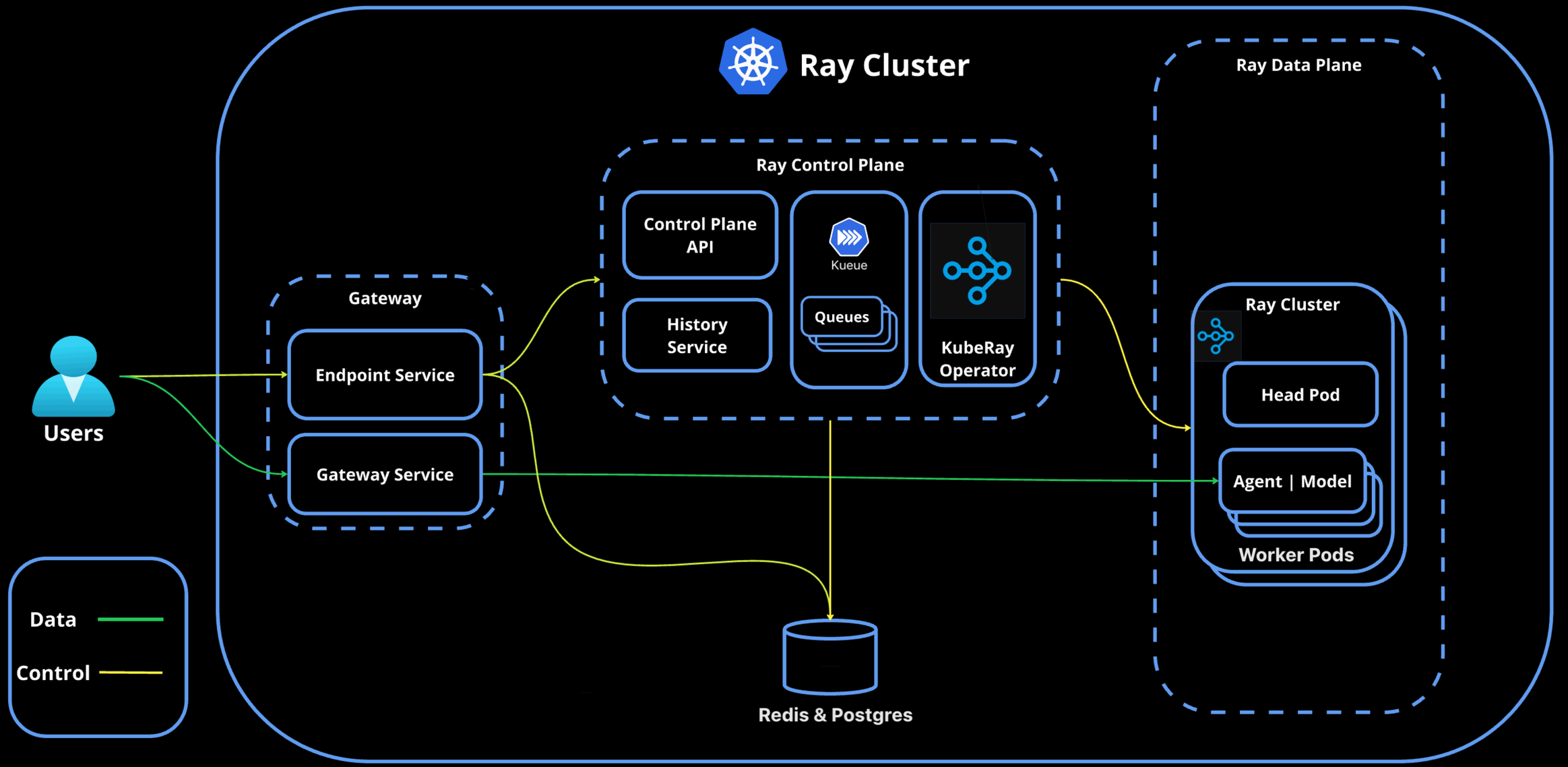
- Lifecycle: create, deploy, retire agents
- Permissions: who can deploy, who can call
- Metadata: versions, configs, owners
- Resources: replicas, queues, region budgets

Agent Management

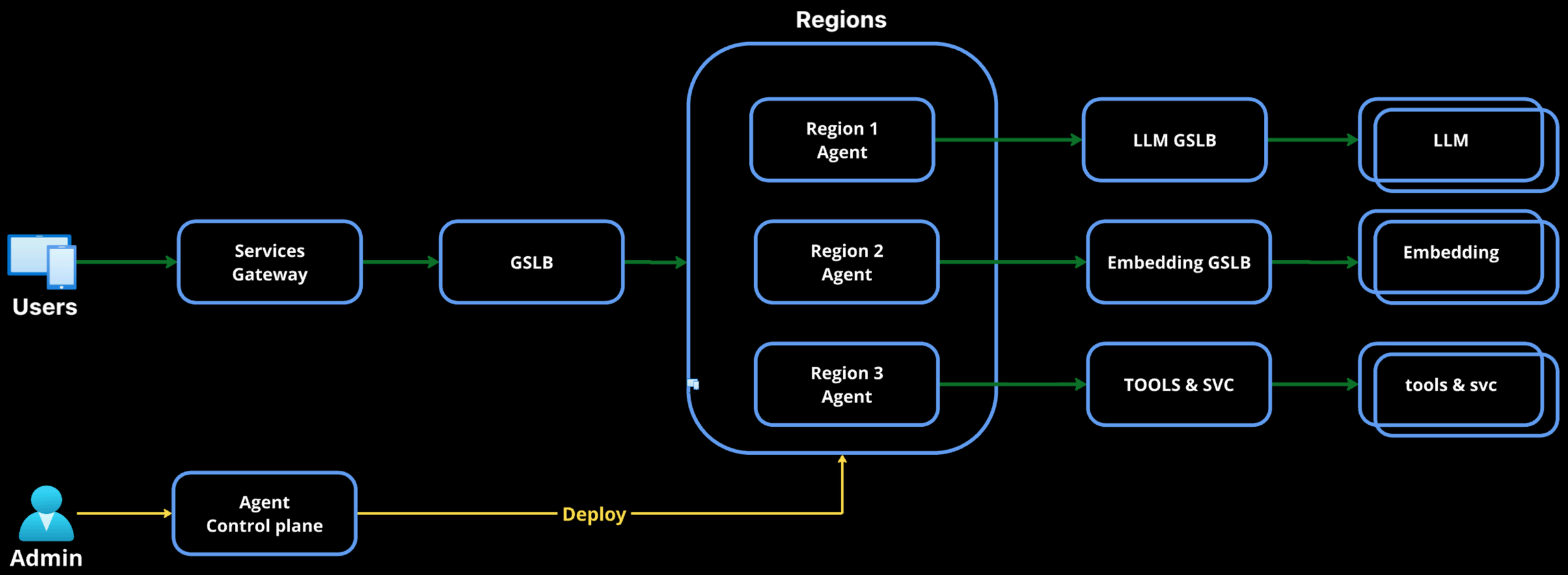
Data Plane

- AuthN: verify caller identity
- AuthZ: agent + tenant policy
- Rate limiting
- Tracing: propagate trace ids end-to-end
- Routing: dispatch into the right RayCluster

Architecture - Deep dive



Scaling out to Multiple Region



Call Outs for Agentic Workloads

Safety

Pre-filter

Regex on the user query < 1 ms

In-graph classifier

Re-checks safety each turn

Post-filter

Scrub outputs before they ship

Prompt Engineering and Versioning

You are Trip-Planner, a concise,
friendly travel assistant.

Workflow:
1. search_destinations for the season
2. get_weather before outdoor activities
3. build_itinerary for the chosen city
4. Narrate output; never dump raw JSON.

Ground rules:
- Never invent cities, prices, hours.
- Cap reasoning to ~6 tool calls / turn.
- Prefer depth over breadth.

You are Trip-Planner, a sharp,
opinionated travel concierge.

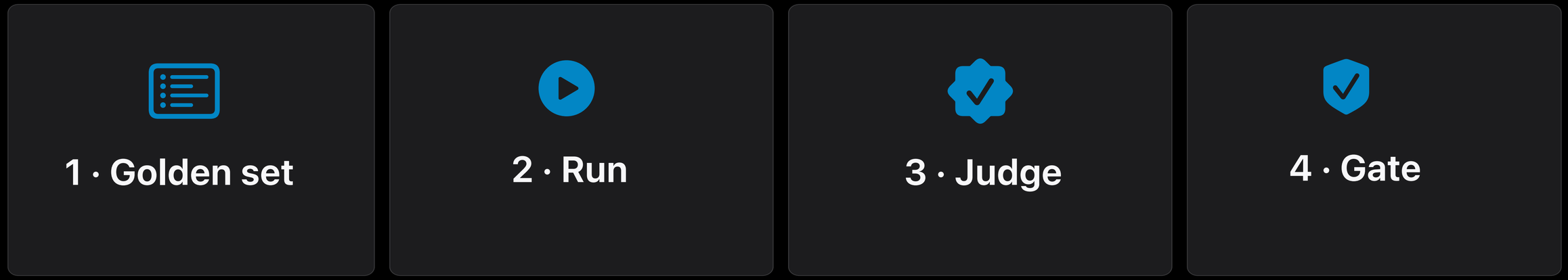
Always explain WHY a recommendation
fits: cite the weather, the season, or
what the user said about themselves.

Tool policy (call in order):
1. search_destinations if no city named
2. get_weather BEFORE outdoor advice
3. build_itinerary narrate, don't dump JSON

Hard rules:
- Never invent cities, prices, hours.
- Cap tool calls at 6 per user turn.
- Depth over breadth.
- Tone: warm, crisp

Eval

Agent outputs and executions are non-deterministic



Production Release Pipeline

Build

CI - Unit tests

Eval

Golden set

Smoke

e2e tests

Canary

First region

Fleet

Sequential per-region

Wrapping up

Putting it all together... Ray Cluster & Ray Serve

Heterogeneous Resources

Scheduling & Orchestration

Traffic-aware Autoscaling

Native Streaming

Async Inference

Built-in Observability

Q & A

Thank You