



Invisible Infrastructure at Scale

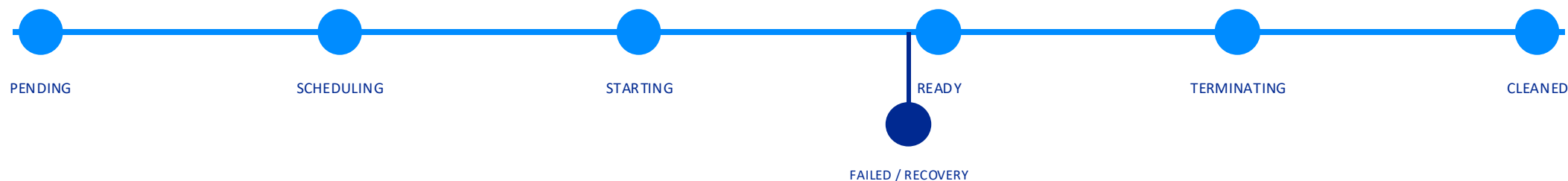
Operating 5,000 daily notebook servers across multiple Kubernetes clusters

Karthik Ravi and Karthik Banala

PayPal Cosmos.AI Platform

Managing Notebook Server Lifecycles at Scale

5,000+ daily notebook servers move through independent states across multiple Kubernetes clusters



VARIABLE INTENT

CPU, memory, accelerator, image, storage, and runtime requirements differ for every server.

VARIABLE CONTEXT

The Kubernetes cluster, resource objects, endpoint, and observed state are cluster-specific.

VARIABLE OUTCOME

Startup can succeed, stall, fail, recover, or leave stale resources that still require cleanup.

One launch request creates an operational responsibility that lasts until cleanup is verified.

Cosmos Platform

■ Deep dive ahead

AI TOOLS & EXPERIENCES

AI Assistants

Coding Tools

Harness Tools & Components

AI Marketplace

AGENTIC AI PLATFORM

Agent Lifecycle & Components

Agent Frameworks

Skills & Plugins

MCP

AI GATEWAY Unified model access · routing · guardrails · rate limits · cost & usage telemetry

LLM PLATFORM

Training & Evaluation

Model Repository

Inferencing

MULTI-MODAL PLATFORM

Doc AI · Voice AI

VENDOR MANAGED AI

Agent Platforms

Model Providers

AI INFRASTRUCTURE

GPU Clusters

Cloud & on-prem

Fleet Management

Capacity & health

Scheduling

Kueue · quotas

Storage & Network

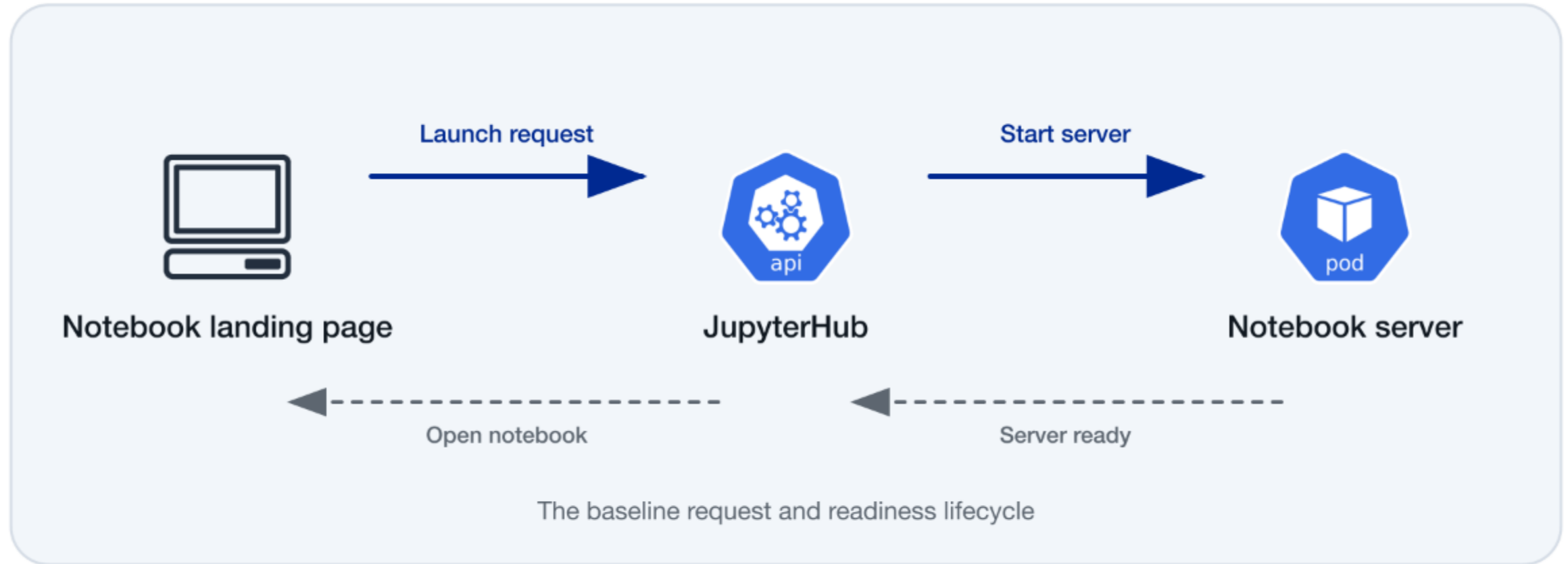
High-throughput fabric

Workbench

JupyterHub

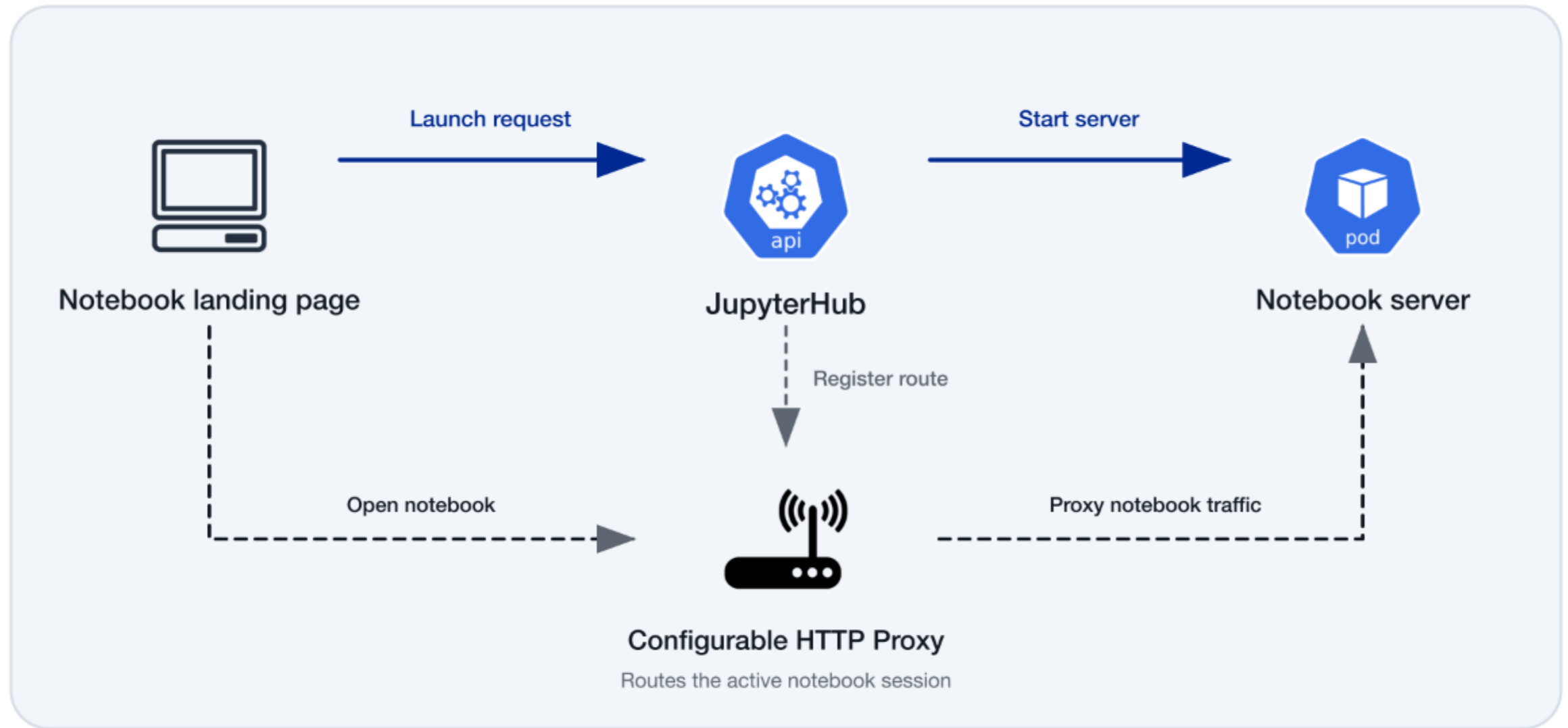
SECURITY · GOVERNANCE · OBSERVABILITY

Notebook Creation and User Traffic Follow Different Paths



A ready server still needs a stable route for browser and WebSocket traffic

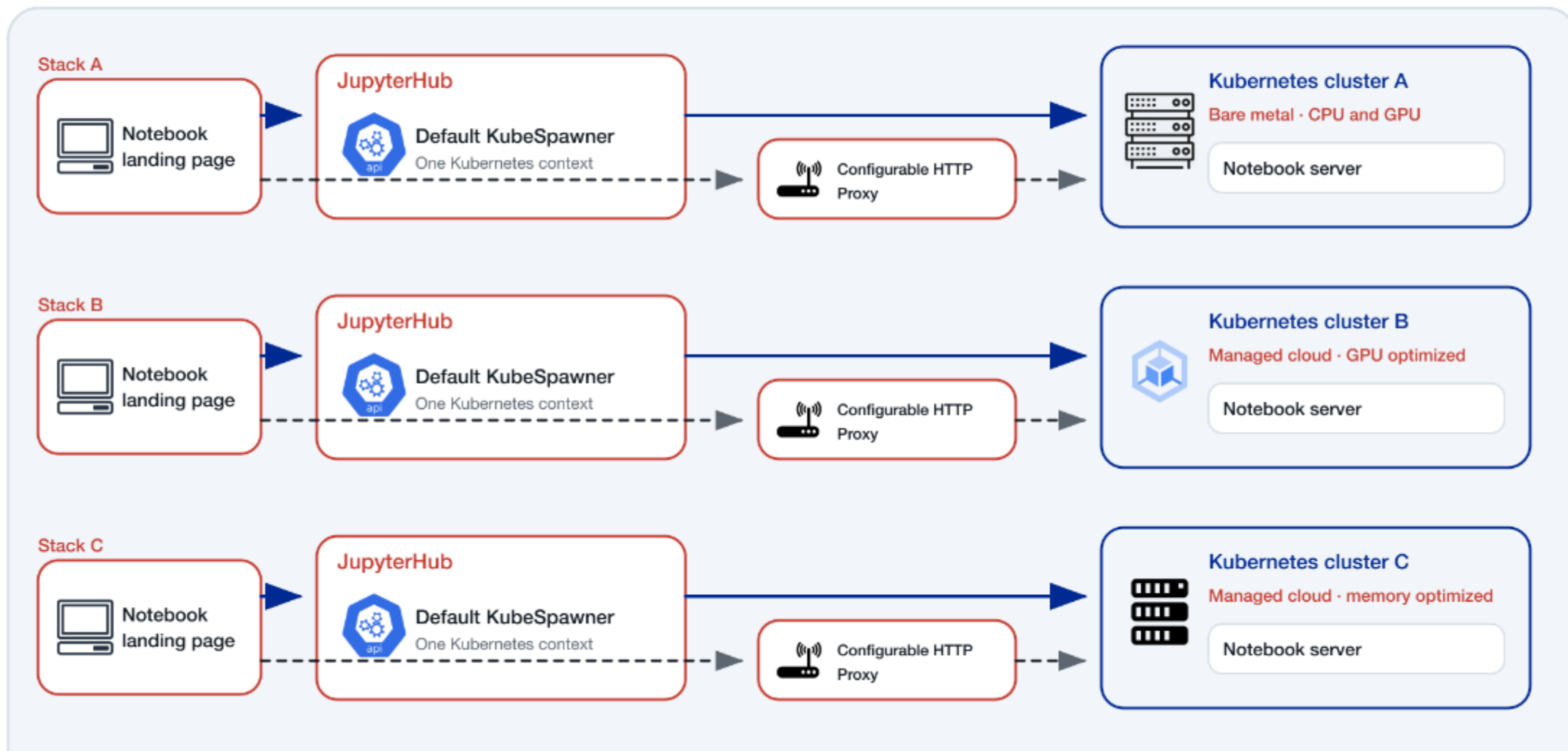
Compute Capacity Extends Beyond One Cluster



The proxy improves access, but placement still assumes one Kubernetes context

Each Cluster Adds Another Operational Stack

Every new cluster repeats the entry point, control plane, proxy, and maintenance path



The cluster is only the first item on the receipt

Duplicating the stack creates recurring operational costs long after provisioning.

YOUR ORDER: 1 NEW CLUSTER

1 × Kubernetes cluster	EXPECTED
1 × Landing page	+1
1 × JupyterHub	+1
1 × Configurable HTTP Proxy	+1
1 × Configuration and credentials	+1
1 × Monitoring and upgrade path	+1

TOTAL

ANOTHER PLATFORM TO OPERATE

THE RECURRING CHARGES

DEPLOY

Deploy and upgrade independently

DRIFT

Keep configuration consistent across stacks

ROUTE

Maintain another user entry and routing path

RECOVER

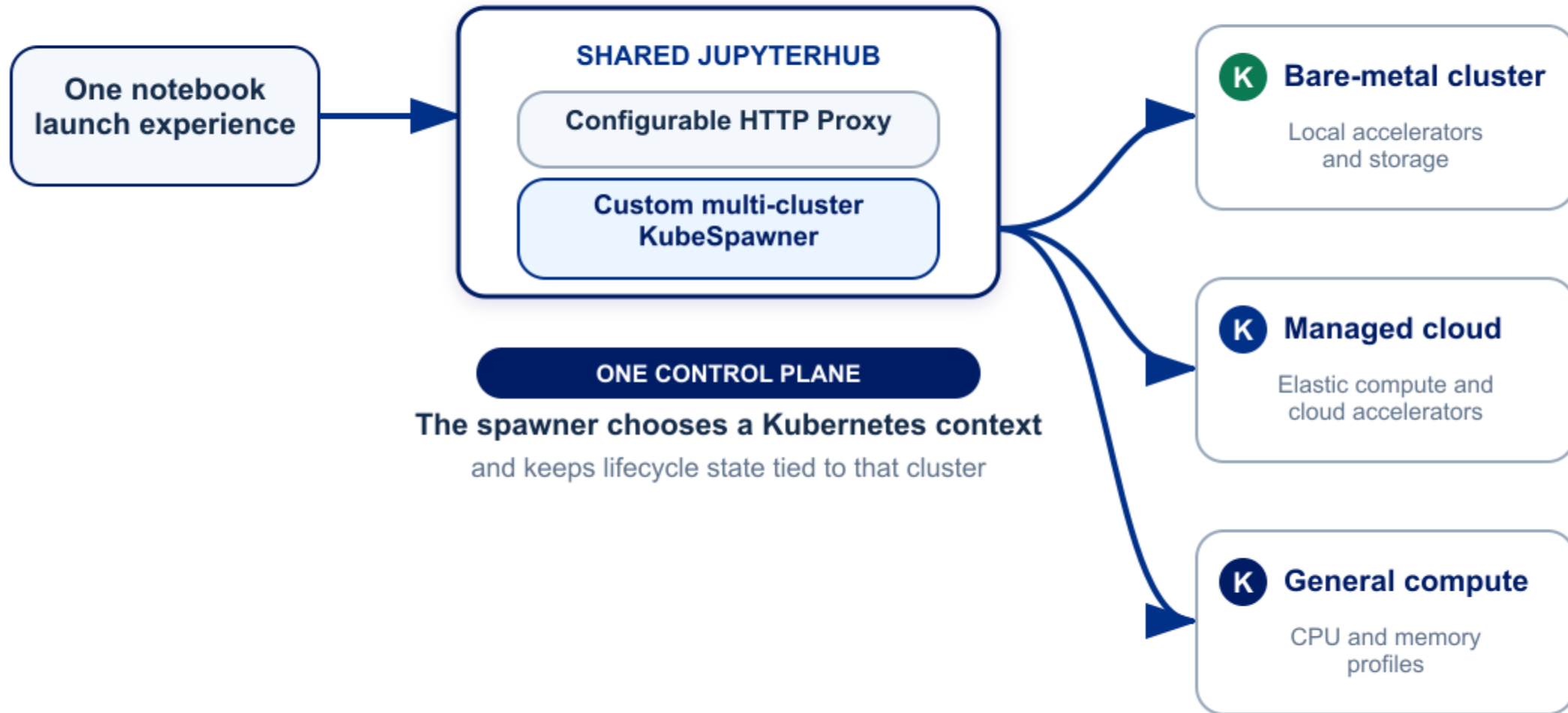
Diagnose, recover, and clean up separately

AUTO-RENEWS FOREVER

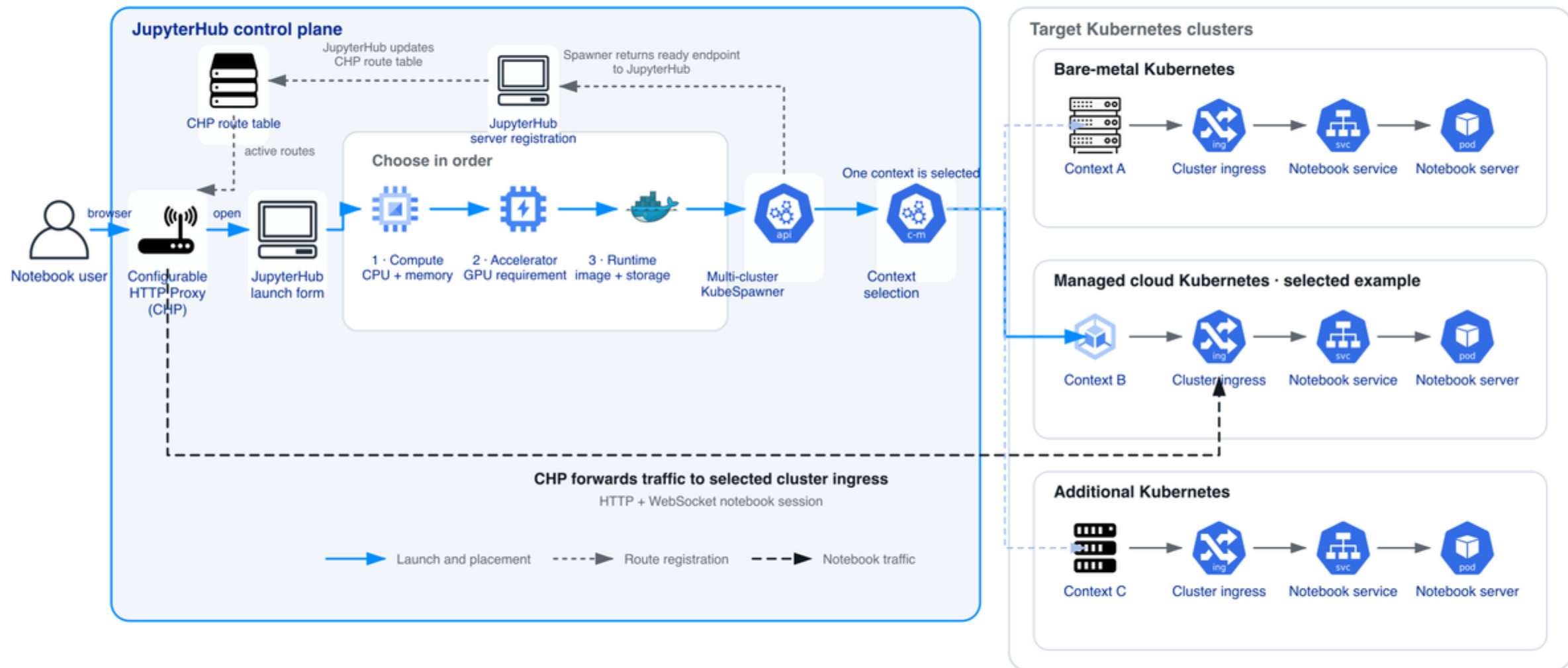
The cluster was the inexpensive line item.

We needed clusters to scale without scaling JupyterHub operational stacks.

One JupyterHub Across Multiple Kubernetes Clusters



Launch, Place, Register, and Route



Optimization 1: Persist cluster context with each notebook server

A server name identifies the workload; durable spawner state identifies where its lifecycle belongs.

PROBLEM AT SCALE After a Hub restart, a server key alone cannot tell us which Kubernetes context owns that notebook server.

NOTEBOOK-SERVER LIFECYCLE IDENTITY

SERVER KEY	user + named server
KUBERNETES CONTEXT	selected target cluster
NAMESPACE	resource ownership boundary
CREATED RESOURCES	objects to restore or delete
ENDPOINT	routable notebook location



HOW THE CONTEXT IS SAVED FOR ONE SERVER

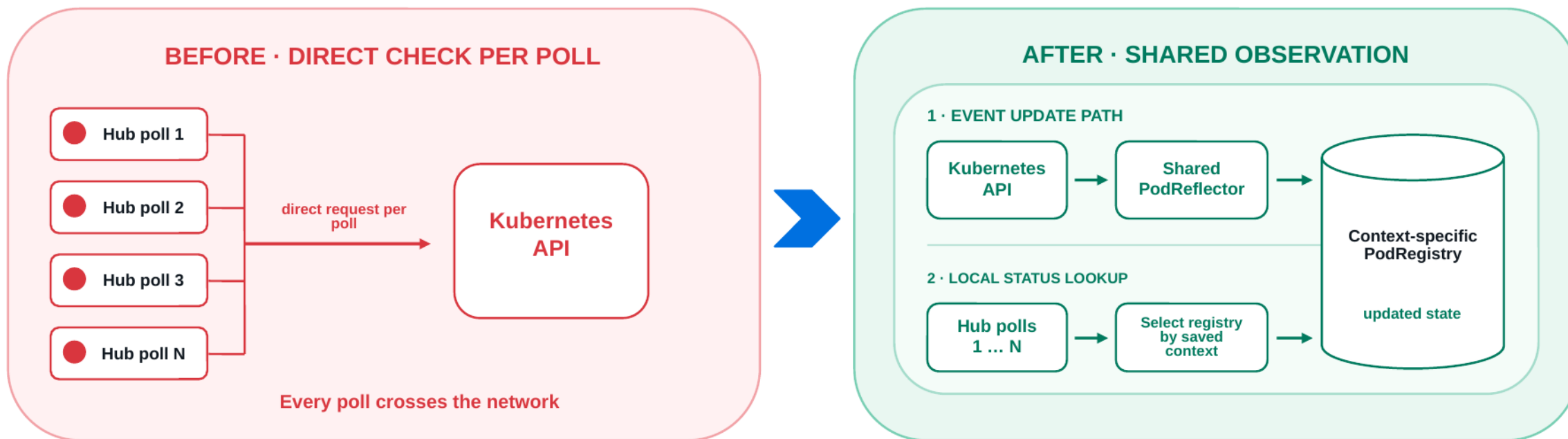
- 1 SELECT** The server receives a target Kubernetes context
`self.kubernetes_context = selected cluster`
- 2 SAVE** `get_state()` adds the context to server state
`"kubernetes_context": selected cluster` 
- 3 RESTORE** `load_state()` reinstates the same context
`self.kubernetes_context = saved value`

SEPARATE RUNTIME PATH The per-context PodRegistry cache accelerates status checks; it does not persist lifecycle ownership.

Optimization 2: Observe by cluster, not by notebook

Replace repeated per-server Kubernetes reads with one shared observation path for each context and namespace.

PROBLEM AT SCALE If every Hub poll becomes a Kubernetes request, normal lifecycle monitoring turns into sustained API pressure.



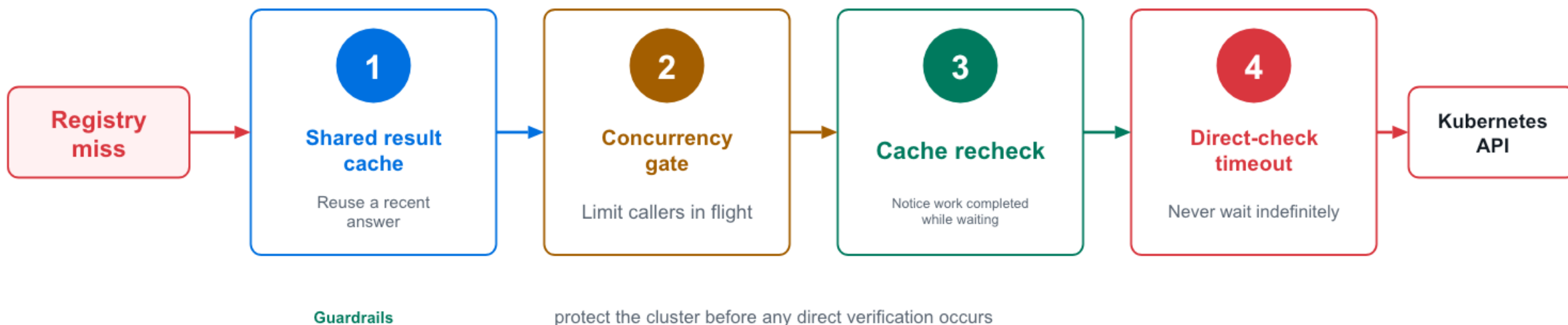
OPERATIONAL EFFECT

Many notebook checks share one continuously updated view of cluster state.

Optimization 3: Bound the exceptional path

Treat direct Kubernetes verification as an exception protected by four explicit controls.

PROBLEM AT SCALE A reliable fallback must not become a second, unbounded polling architecture.



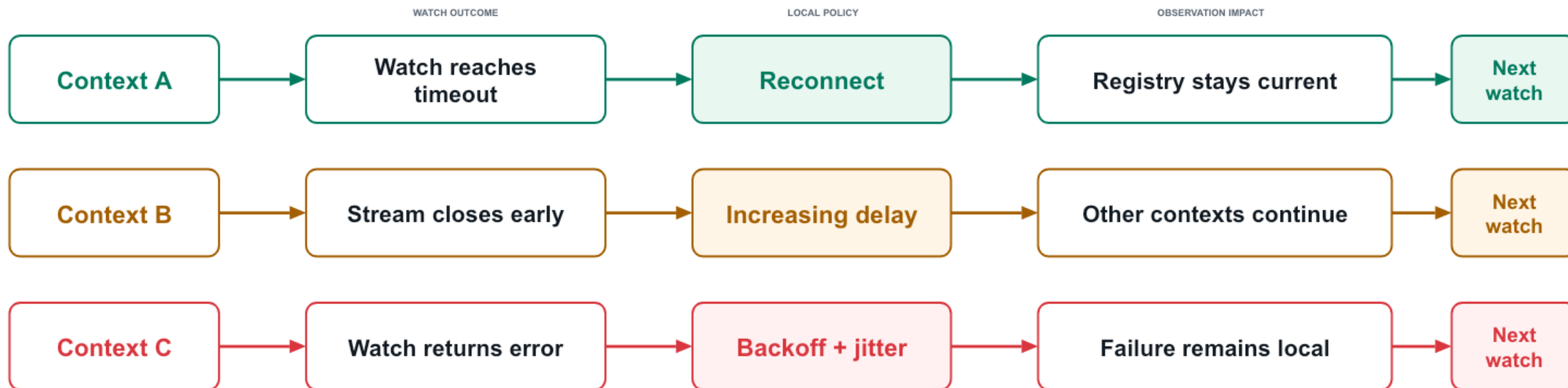
DESIGN PRINCIPLE

Optimize the expected path; constrain every escape hatch.

Optimization 4: Reconnect without creating a storm

Let each Kubernetes context own its observation health, retry state, and registry updates.

PROBLEM AT SCALE A global recovery loop can make one unstable cluster interrupt observation everywhere.



DESIGN PRINCIPLE Isolate observation and recovery by context so one cluster cannot create global uncertainty.

The reusable multi-cluster lifecycle contract

One saved context anchors every lifecycle responsibility after placement.



TAKEAWAY

The cluster context becomes the notebook's lifecycle identity—not merely its launch destination.

A reusable framework for multi-cluster lifecycle control

What the spawner design taught us

OUTCOME

One JupyterHub experience across multiple Kubernetes clusters

1

Persist ownership

Save the selected cluster context with notebook server state

2

Observe within context

Use one shared registry-backed view for each Kubernetes context

3

Bound exceptional checks

Use cached state first. Gate, recheck, and time-limit direct verification

4

Isolate recovery

Keep watch health and retry behavior within each context

REUSABLE SCOPE

Create, observe, restore, and clean up remote resources through one durable context identity.

Questions?

Invisible Infrastructure at Scale

Operating 5,000 Daily Notebook Servers Across Multiple Kubernetes

Karthik Ravi and Karthik Banala