Massed Compute

POSITION PAPER

The Heterogeneity Tax

MCP as the Post-Kubernetes Control Plane for Heterogeneous GPU Infrastructure

Steve McKay

SVP Technology, Massed Compute

Abstract

Heterogeneous GPU infrastructure rewards per-device-class tuning, yet platform teams still deploy through portable Kubernetes manifests and golden container images that run everywhere but optimize nowhere. Per-workload tuning is expressible in YAML and GitOps, but sustaining it as hardware generations, frameworks, and traffic shapes drift exceeds practical human capacity. This position paper argues that an MCP-connected agent control plane should replace or avoid adopting Kubernetes as the user-facing operational layer for accelerated compute: not because the MCP protocol is an orchestrator, but because GPU fleets need a performance-first adaptation loop where composable tool servers and policy replace declarative reconcile as the primary control law. The resulting performance gap is formalized as the heterogeneity tax; the paper surveys what the Kubernetes ecosystem already automates, compares automation parity across three stacks, and synthesizes published evidence without primary benchmarks. Three linked hypotheses organize the argument: a real performance gap on mixed fleets (**H1**), a policy-guarded closed adaptation loop as the alternative control law (**H2**), and unsustainable human-scale sustain through Kubernetes-shaped operations (**H3**). The shift from spec conformance to runtime optimization is an architectural hypothesis supported by secondary analysis; Table 2 scenarios define the head-to-head evaluation that quantifies it for accelerated infrastructure.

1. Introduction

Kubernetes has become the standard orchestration layer for CPU and GPU workloads alike, including many new GPU clusters and cloud accelerator pools. Platform teams use it to run microservices, batch jobs, and accelerated compute (server-side work that requires GPUs or similar hardware for acceptable performance): inference and model serving, distributed training, rendering, video processing, and GPU-backed simulation. That dominance makes sense when one portable pod specification can describe the work. GPU workloads rarely stay that simple. They differ by device class and generation, driver and CUDA stack, memory footprint, batching behavior, and sensitivity to node topology. The same cluster often mixes those profiles while users still interact through the same Kubernetes objects. This paper argues that the Model Context Protocol (MCP) should replace or avoid adopting Kubernetes as the user-facing control plane for heterogeneous GPU infrastructure, because workload diversity on accelerators is the normal operating condition, not an exception Kubernetes can absorb with more YAML.

Figure 1 previews the replacement architecture. An MCP host sits above composable tool servers that drive node runtimes directly; the Kubernetes API is absent from the control path users and agents rely on to get work done. MCP is not a hypervisor. It is the operational contract: an agent or workflow calls tools to provision and release GPU capacity, attach storage, deploy an accelerator-aware runtime, observe utilization and latency, tune batch size or image choice, and roll back. Kubernetes forces every GPU job through static manifests reconciled toward a fixed spec. MCP lets each workload class execute its own sequence: spin up an inference pool, tear down a training gang, retune a rendering pipeline, select a device class, verify, repeat.

Heterogeneity is handled by varying actions, not by stretching one portable pod template across every accelerator class and business workflow.

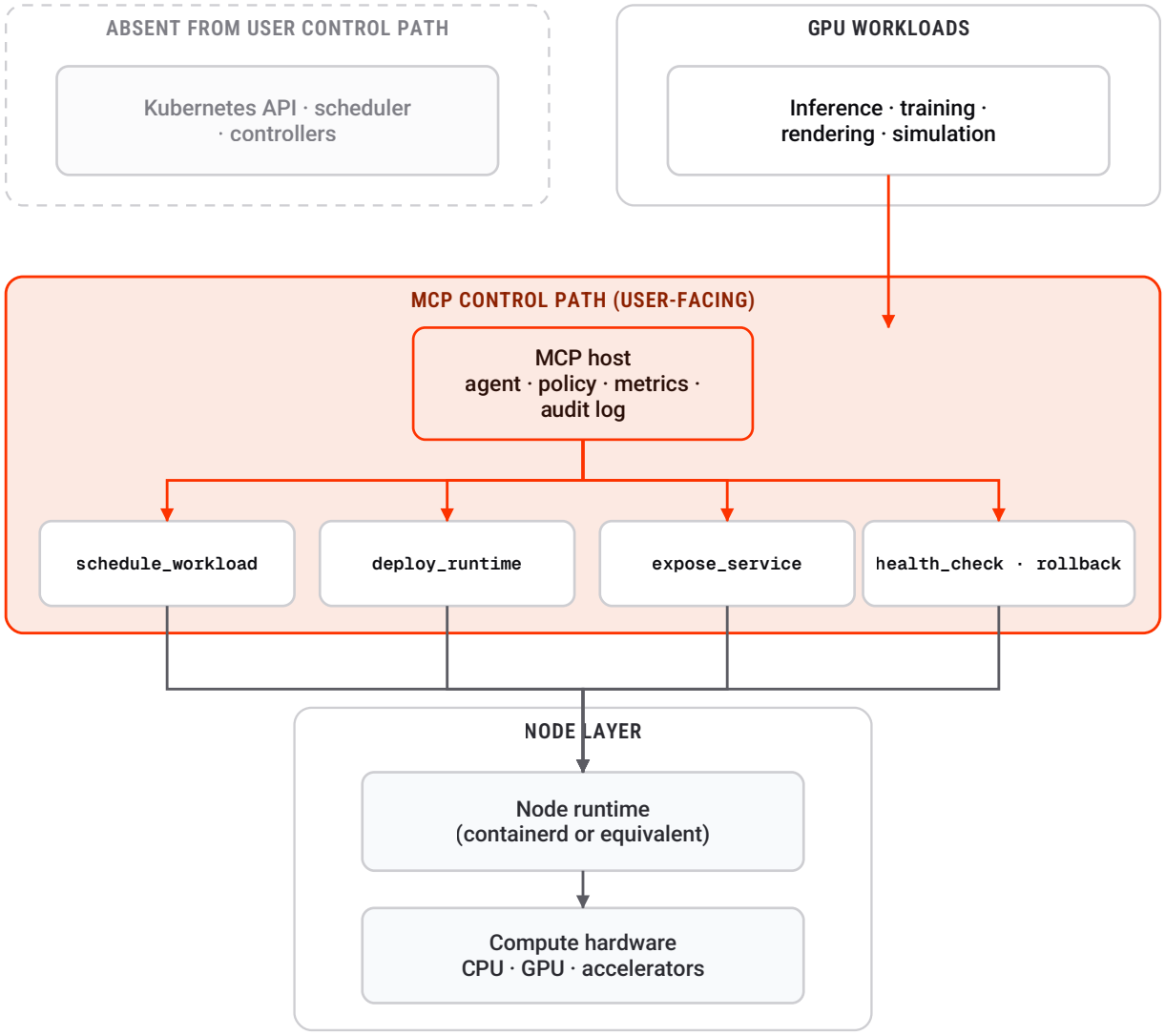

Figure 1. *Replacement architecture for heterogeneous GPU infrastructure. Users and agents interact through an MCP host and composable tool servers. Node runtimes and hardware remain; the Kubernetes API, scheduler, and controllers are not on the user control path.*

The cost of forcing GPU work through that template is the **heterogeneity tax**: the gap between a portable baseline (one golden image, one Deployment-shaped spec) and a deployment tuned for each accelerator class and job purpose. The tax is well documented on GPU fleets. Iteration-level scheduling for LLM serving has reported order-of-magnitude throughput gains over static batching on the same devices [1]. Framework and runtime choices produce double-digit spreads on identical GPUs [2]. Teams on heterogeneous accelerator clusters running Kubernetes have reported large tail-latency improvements only after layering topology alignment, fractional partitioning, and placement policy well beyond a single Deployment [3]. Those results illustrate what Kubernetes leaves on the table when configuration stays static while GPU work keeps changing.

Three hypotheses tie the argument together—heterogeneity tax (**H1**), closed adaptation loop (**H2**), sustain burden (**H3**)—tested analytically in Section 6. MCP replacement follows when the tax is real, the adaptation loop is plausible, and the sustain burden exceeds human scale. Kubernetes is built for the portable baseline, not for continuous performance tuning under drift. Controllers reconcile desired configuration with running state, which suits workloads where a fixed spec is enough. Heterogeneous fleets need a different loop: acquire the right device class, apply the matching stack, measure throughput or time-to-solution, adjust batch settings or image, move placement, release. Static GitOps locks operators into spec conformance while platform teams manually close the performance loop through node pools, operators, and pull requests. MCP is designed for the second loop as the primary control law.

Volcano, Kueue, GPU operators, and serving autoscalers such as KServe with KEDA [4]–[6] patch individual pain points on Kubernetes: queueing, gang scheduling, driver install, replica scaling. They do not fix the structural mismatch. Each new GPU workload scenario still tends to add Custom Resource Definitions (CRDs), webhooks, and charts [7]. Kubernetes accumulates integrations; MCP extends tool servers at lower marginal cost. Where accelerator types, frameworks, and workflows multiply, the MCP path scales intent; the Kubernetes path scales configuration debt.

The MCP model separates two timescales that heterogeneous GPU fleets require (Figure 5, Section 5.4). The inner loop optimizes a running job: observe metrics, change runtime parameters, images, or placement, confirm or revert. The outer loop adds capability as new GPU generations, regions, or workload types appear, without freezing every user on a fleet-wide manifest change. Kubernetes closes on spec conformance; MCP closes on business requirements and GPU workload outcomes through policy-driven monitor-decide-act loops. That difference is why MCP is treated not as a companion to Kubernetes but as the successor operational layer for accelerated compute.

The case is developed for anyone who provides or consumes GPU compute today, within high-performance computing (HPC) clusters that mix CPU and accelerator resources. Evidence draws on published GPU studies, automation-parity analysis across three stacks, and operational complexity reasoning [8]. For heterogeneous GPU infrastructure, MCP is a better fit between how work is described and how capacity is configured.

Contributions

1. **Problem formalization.** Definition of the heterogeneity tax and why it persists for GPU and mixed CPU/GPU workloads on Kubernetes-shaped platforms, including with a full ecosystem of schedulers and operators.
2. **Replacement architecture.** Mapping of Kubernetes control functions to MCP tool servers and agent policy, with dual-timescale adaptation for heterogeneous GPU workloads (Figure 1).
3. **Secondary evidence and sustain-burden analysis.** Synthesis of published GPU benchmarks, automation-effort comparison across Kubernetes native, Kubernetes plus ecosystem, and MCP agent stacks, sustain-burden modeling, and what primary evaluation would still be required.

Scope

This is a position paper, not a production benchmark report. The argument is for replacing or not adopting the Kubernetes operational layer on GPU infrastructure, not for eliminating containers, hypervisors, or node

runtimes underneath. The same thesis applies at build time on greenfield accelerator pools (Section 8.2) and as migration off installed Kubernetes (Section 8.3). Quantitative claims about how much of the heterogeneity tax an agent loop closes remain **H2** hypotheses informed by autotuning precedent [9] until primary evaluation is available. Section 8 discusses greenfield adoption, brownfield objections, tenancy, auditability, and remaining gaps.

Organization

The argument maps to those hypotheses: Section 3 develops **H1**; Section 4 **H3**; Section 5 **H2** and the replacement architecture; Section 6 tests all three. Section 2 aligns vocabulary for reconciliation and MCP. Section 7 situates related work between evidence and objections. Section 8 covers objections, open problems, and deployment paths. Section 9 concludes. References are listed in Section 10.

2. Background

Two control models frame the paper: Kubernetes reconciliation and an MCP-connected agent loop. Familiarity with clusters is assumed; this is not a Kubernetes tutorial. Section 3 names and breaks down the heterogeneity tax on mixed GPU fleets.

2.1 Kubernetes Reconciliation Model

Large-scale cluster managers converged on the same pattern: users declare desired state; a control plane drives observed state toward it. Borg at Google established the template that Kubernetes later productized for a portable API [10]. The user submits manifests (Deployments, Jobs, Services, and custom resources). Controllers watch those objects and emit changes until running workloads match the declaration. The scheduler assigns pods to nodes subject to resource requests, affinity rules, and quotas. The kubelet and container runtime (typically containerd or an equivalent CRI backend) start containers on the chosen node. Networking and storage attach through parallel controller paths.

The closed variable in this loop is **spec conformance**: does what is running match what was declared? Controllers retry, backoff, and surface conditions when they do not. That objective fits long-lived CPU microservices on homogeneous hardware, where the right outcome is stable replicas behind a known image and resource limit. It also fits batch jobs when the declaration already encodes the placement and resource shape the job needs.

Reconcile optimizes what was declared, not whether running work still meets **business requirements** (latency tiers, throughput floors, cost caps, deadlines, priority) after traffic, hardware, or workload mix drifts. Observability can surface those gaps; unless a human edits the manifest, requirement miss is not the control error reconcile closes on.

Accelerated workloads still flow through the same loop. Device plugins and extended resources expose GPUs or fractional partitions into the pod spec; Volcano and Kueue extend scheduling policy. None of that changes the control law. The desired state remains a fixed YAML document until a human or pipeline edits it. Performance tuning encoded as environment variables, image tags, or affinity rules is just more fields in the

same reconcile target. Section 4 surveys how much automation the ecosystem adds on top; the baseline model is unchanged.

2.2 GPU and Accelerated Workload Sensitivity

GPU-accelerated inference, training, rendering, and simulation share a trait that portable pod specs under-specify: effective throughput and time-to-solution depend on choices made at several layers, and those choices interact.

Application runtime parameters include batch size, concurrency, precision, and framework-specific scheduler settings. Published serving systems report large spreads when these differ on the same device class [1], [2]. Container and runtime stack choices fix driver generation, CUDA version, and library builds. An image that boots on every node in a mixed fleet is rarely optimal on any one device class. Topology and placement determine whether multi-device work can use fast interconnect paths; collective libraries assume particular PCIe and network layouts.

Kubernetes can represent all three layers in manifests: ConfigMaps for parameters, image fields for stacks, affinity and topology spread for placement. The platform does not automatically search that space when traffic shifts or a new accelerator generation arrives. Operators or humans change the declaration, reconcile runs again, and the fleet moves to a new fixed point. Autotuning services in machine learning show that closed loops over runtime parameters can run at scale when objectives and guardrails are explicit [9]; Kubernetes itself does not provide that loop for infrastructure configuration. Section 3 decomposes the resulting performance gap into layers P1–P4; the decomposition applies beyond inference to other accelerated and mixed CPU/GPU jobs on shared clusters.

2.3 MCP and Agentic Control Planes

The Model Context Protocol (MCP) defines how a host discovers and invokes tools exposed by servers, with optional resources for read-only context [11]. MCP is a wire format and lifecycle contract, not a scheduler or optimizer. It standardizes list-tools and call-tool interactions so new capabilities ship as servers rather than as bespoke APIs per integration.

In the replacement architecture outlined here, the host connects to composable tool servers that perform infrastructure actions: schedule work on nodes, deploy a container with a chosen runtime stack, attach storage, expose endpoints, read metrics, roll back a bad change. An agent (which may be LLM-driven or a deterministic policy engine) sits in the host and decides which tools to call in what order. Policy constrains the allowlist, argument bounds, and approval gates, and encodes **business requirements** as objectives the loop must satisfy. Metrics tools supply performance monitoring for the observe step.

The closed variable differs from reconciliation. The agent asks whether workload outcomes meet business requirements on the hardware that is present now, not whether running state matches a checked-in spec. The inner loop evaluates each tool action against those requirements continuously. Tools mutate the environment directly through the servers; there is no requirement that every change flow through a single portable Deployment document first. That separation matters for heterogeneous GPU fleets, where the right sequence of actions varies by workload class and drift event.

MCP does not replace node runtimes, drivers, or physical hardware. It replaces Kubernetes as the user-facing operational layer for the workload class this paper targets: the path through which intent becomes provisioned capacity and tuned runtime behavior. Figure 1 (Section 1) sketches that stack. Section 5 maps Kubernetes control functions to tool servers and describes inner and outer adaptation loops. Section 3 quantifies the performance gap that motivates changing the control law in the first place.

3. Problem: The Heterogeneity Tax

HPC clusters rarely serve a single purpose. The same infrastructure may run CPU-bound simulation, GPU-accelerated training, model serving, rendering, and ad hoc analysis for different groups. The resulting gap is the **heterogeneity tax**; Section 3.2 decomposes it. GPU-accelerated serving is cited heavily where published numbers exist; the decomposition applies to other accelerated and mixed CPU/GPU workloads that depend on tuned runtimes, images, and placement.

Section 2 covers reconciliation and the MCP host-tool model; that vocabulary is used throughout.

3.1 Definition

The **heterogeneity tax** is the gap between two deployment states:

1. a **portable baseline**: one golden image, one Deployment-shaped spec, one GitOps pipeline;
2. a **per-platform tuned deployment** that meets workload goals on each device class, software stack, and topology in the fleet.

The tax is not one number. It is several deficits that compound when hardware, libraries, and traffic drift. Teams pay it as lost performance or missed time targets (run the portable baseline) or as operational labor (maintain per-workload configs, node pools, and glue code). Kubernetes allows the second path on both CPU and accelerated nodes. Sections 4 and 6 argue that it does not scale under heterogeneity and drift without changing how control works.

Hypothesis **H1** (Section 1) holds that portable defaults often underperform tuned configs on mixed accelerator fleets. The citations in Section 1 and Table 3 (Section 6) support that claim for GPU-serving workloads; the structure generalizes to other tuned HPC jobs.

3.2 Decomposition: P1–P4

Blaming the whole tax on containers or on Kubernetes alone mixes causes. Four layers capture it. A single drift event may need changes at more than one.

P1: Container and runtime stack. The image fixes framework version, libraries, and driver assumptions on accelerated nodes (and toolchain choices on CPU-heavy jobs). A build tuned for one device generation may underuse another or fail compatibility checks. Fractional accelerator partitioning adds device plugin settings, extended resource requests, and partition profiles that must match the image [12]. Portable images converge on the lowest common denominator that boots everywhere, not the stack that performs best for any one workload class.

P2: Application runtime parameters. Throughput and time-to-solution depend on parameters the application controls: batch sizes, concurrency, precision, solver settings, or scheduling granularity. In GPU-accelerated inference, Orca reported a 36.9× throughput gain over a static-batch baseline at matched latency [1]; vLLM's continuous batching shows similar sensitivity [13]; framework comparisons on identical accelerators often show double-digit spreads [2]. Analogous tuning exists in HPC batch jobs and rendering pipelines. None of it comes from a portable pod spec alone; it requires tuned settings and retuning when the problem changes.

P3: Topology and placement. Distributed and accelerated work cares how CPU, memory, accelerators, and interconnect align. Multi-node jobs depend on network and PCIe layout for collectives [14]. On heterogeneous accelerated clusters running Kubernetes, practitioners have reported large tail-latency improvements after topology-aware kubelet config, fractional partitioning, and spread placement [3]. CPU-only MPI-style workloads show similar placement sensitivity in classical cluster operations. Bad placement rarely fails the pod; it fails the job's performance target.

P4: Scheduler interference. Some loss is not portability at all: queueing, gang scheduling, preemption, and noisy neighbors on shared clusters. Volcano and Kueue handle admission and placement policy for batch-oriented shapes [4], [5]; they do not retune application parameters when workload characteristics drift. P4 matters even when P1–P3 are handled, but it does not remove the cost of portable defaults at those layers.

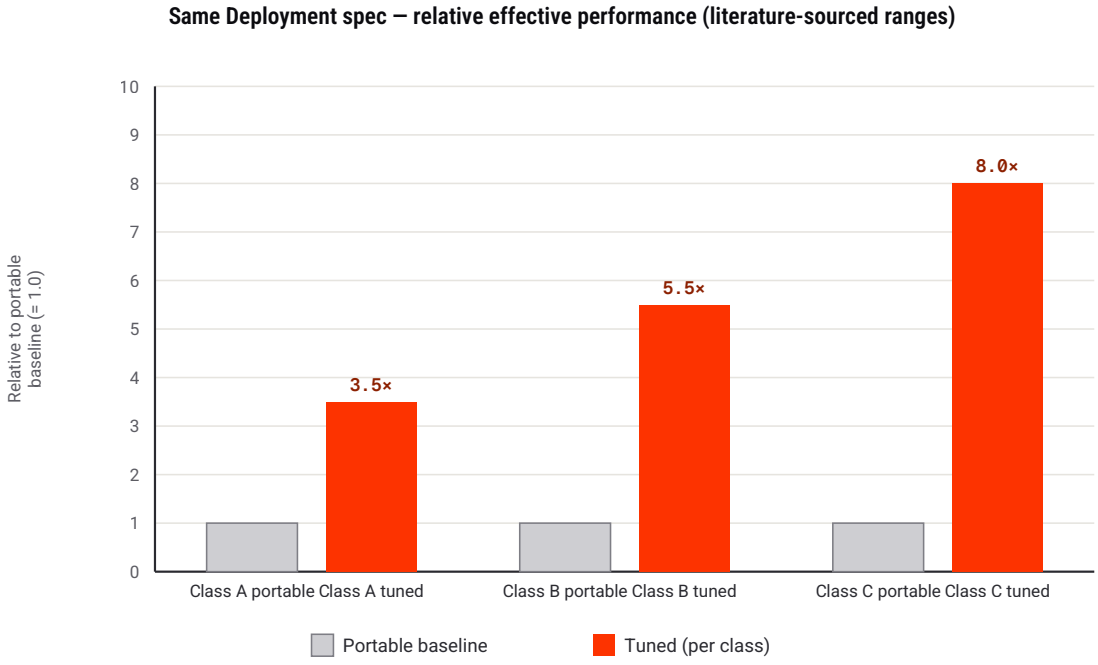

Figure 2. *Same nominal Deployment spec, different effective performance across three generic device classes (A, B, C) in a mixed fleet. Gray bars: portable baseline (= 1.0). Orange bars: illustrative tuned gains from runtime batching on class A [1], [13], framework choice on class B [2], and topology-aware stack deployment on class C [3]. Not measurements from this paper.*

3.3 Golden Image Practice

One image for the fleet is reasonable operations: one scan surface, one patch path, one compliance review. Teams are not wrong to ship the same container to every pool, CPU or accelerated. The practice still pushes

tuning toward the lowest common denominator, because every per-workload exception forks the pipeline (more manifests, more tests, more on-call notes).

Mature shops run multiple node pools and image tags per device class. That proves tuning is possible; it does not prove it scales. Each new hardware generation or workload family grows the configuration cross-product analyzed in Section 4. Labor can reduce the tax; it cannot erase it if drift outruns edit bandwidth.

3.4 Workload Examples

In-pod vs cross-node (accelerated compute). vLLM's continuous batcher handles part of P2 inside the container without Kubernetes [13]. That helps until optimal settings diverge across device classes or the fix requires a different runtime image. Then someone must decide per pool: patch env vars, redeploy a different image, or move work to better-connected nodes.

Healthy pods, missed targets. A job's input distribution shifts (traffic shape for serving, mesh resolution for simulation, frame size for rendering), or business requirements change (SLA tier, budget cap, job priority, deadline). Resource pressure rises, metrics degrade, utilization still looks acceptable, and reconcile reports healthy pods. Requirement miss is invisible to the control law: the heterogeneity tax shows up as lost value and steady underperformance, which Kubernetes does not treat as a control error (Section 5.4 contrasts the MCP path).

Section 4 examines why Kubernetes operations, including a full HPC ecosystem install, struggle to close this loop at human scale. Section 5 outlines the MCP-based alternative.

4. Limits of Human-Scale Kubernetes Operations

The heterogeneity tax can be reduced on Kubernetes for both CPU and accelerated workloads. The open question is whether reduction is sustainable when HPC fleets mix device classes and workload purposes, without a human in the GitOps path on every drift event. The analysis below first catalogs what the ecosystem already automates, then why configuration burden still grows, then how three stacks compare on the same operational outcomes.

4.1 What the Ecosystem Already Automates

Treating Kubernetes as "YAML plus the default scheduler" is a weak baseline. Production HPC and ML platforms add substantial automation. Table 2a lists common components and what each actually does.

Table 2a. Kubernetes HPC / accelerated-compute ecosystem (selected components).

Category	Examples	Typical automation
Batch / gang scheduling	Volcano, Kueue	Queueing, gang scheduling, fair-share, topology-aware batch placement [4], [5]
GPU / accelerator	GPU Operator, device plugins, fractional partitioning, Node Feature Discovery	Driver install, accelerator expose, device labels [12]
Model serving	KServe, KEDA	Replica autoscaling on runtime metrics [6]
Node provisioning	Karpenter, Cluster Autoscaler	Device-class-specific node classes
Observability	Prometheus, DCGM exporter	Metrics and alerts (observe only)
GitOps / policy	Argo CD, Flux, OPA/Kyverno	Declarative deploy, admission
In-container tuning	vLLM, Triton	L1 batching inside the pod [13]

Volcano and Kueue are strong at queueing, gang scheduling, and quota admission on CPU and GPU clusters. KServe with KEDA scales replica count from serving metrics. Domain runtimes handle in-container tuning for specific frameworks. No stock install ships a full cross-layer performance loop: detect regression, pick among runtime, image, placement, and resource changes, verify, and repeat on drift without custom glue. That gap is what Section 5 targets.

4.2 Configuration Explosion and Toil

Section 3's layers each add knobs: image tag, driver and library pins, runtime env vars, CPU and accelerator affinity, topology manager policy, partition profiles, queue definitions, NetworkPolicy, autoscaling rules, and Prometheus recording rules. Drift adds events: new device class, region, model, or input shape.

SRE literature defines **toil** as manual, repetitive work that grows with the service and crowds out engineering time if unchecked [8], [15]. Per-workload tuning on Kubernetes, even when expressible in YAML, looks like toil: drift wants a PR, review, merge, and deploy. Burden scales roughly with device classes × configuration dimensions × drift frequency (formalized in Section 6.3).

Hypothesis **H3** (Section 1) is not that operators are incompetent. It is that the integration surface exceeds what a platform team can maintain.

4.3 The Cross-Layer Gap

Ecosystem tools each cover part of the problem. What remains is coordination across layers when something drifts:

1. Metrics show a throughput drop after a traffic-shape change.
2. An engineer picks among L1 batch tweak, L2 image rebuild, topology move, or replica scale, and which pools get which change.
3. GitOps applies the fix (hours to days, plus review).

4. Nothing automatically checks recovery on every device class.

Argo CD and KServe support rollback after a human identifies a bad change. Perf-triggered rollback needs custom PrometheusRule-to-webhook wiring not present out of the box. Prometheus, KServe, and KEDA can signal business requirement pressure (latency, queue depth, cost drivers); closing the loop on requirement miss still needs custom glue or human GitOps, as Figure 3 contrasts. The bottleneck is usually not pod scheduling; it is latency and attention in the performance edit loop.

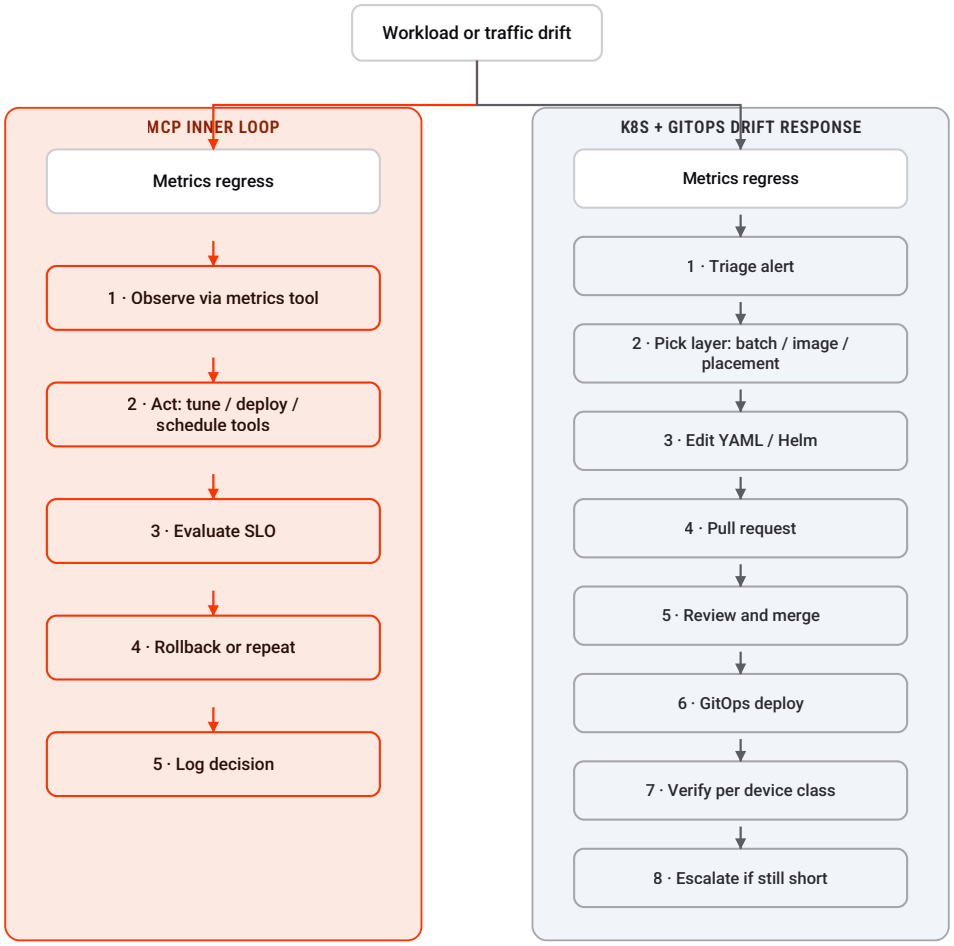

Figure 3. *Illustrative step counts for a drift response: K8s + GitOps (human gates) vs MCP inner loop (Section 5.4). Analytical comparison, not a measured deployment.*

4.4 Automation Parity Preview

Table 2 compares three stacks on the same outcomes: what it costs to build, integrate, sustain, and extend automation. The comparison frames both a greenfield build-time stack choice and the sustain burden on installed Kubernetes fleets. Table 2 primarily tests **H3** (sustain burden); Section 6.2 develops the comparison. Table 2b defines stack shorthand used in Table 2; the effort key below defines L / M / H grades.

Table 2b. Stack shorthand and composition.

Shorthand	Composition
K8s native	Deployments, Services, HPA, affinity, ConfigMaps, GitOps
K8s + ecosystem	K8s native plus Volcano and/or Kueue, GPU Operator, NFD, KServe/KEDA, Karpenter, Argo CD, Prometheus/DCGM, OPA/Kyverno
MCP agent	MCP host, orchestration tool servers, policy guardrails (Kubernetes not in the control path)

Effort key (L / M / H).

Grade	Drift response	Integration / build	Stack sustain
L	Mostly automated once configured; few human gates per drift event	One-time setup; minimal glue	Few moving parts; patch path is narrow
M	Periodic human triage or policy approval	Multiple components wired; some custom glue	Ongoing tool-server or policy updates; moderate on-call surface
H	Manual YAML/PR cycle per drift; scales with device classes	New CRD, operator, or webhook per scenario	Many integrated subsystems; upgrade and config churn (typical K8s + ecosystem: ~8–15 components [7])

Table 2. Automation parity by stack. Shaded cells: rose = H, amber = M–H (effort key above).

Scenario	Outcome	K8s native	K8s + ecosystem	MCP agent
1. Runtime tuning on drift	Restore throughput after workload shift	**H**	**M–H**	**M**
2. Optimal image / stack per device class	Right container stack per accelerator generation	**H**	**M–H**	**M**
3. Topology-aware placement	Work lands on interconnect-friendly accelerator set	**H**	**M**	**M**
4. Rollback after bad tune	Revert to last known good	**M**	**M**	**L–M**
5. Add new automation capability	New metric-to-action path	**H**	**H**	**M**
6. Onboard new device class	Fleet accepts new accelerator generation	**H**	**M–H**	**M**
7. Sustain integrated stack (cross-cutting)	Keep automation patched, integrated, and operable across upgrades and drift	**M**	**H**	**M**

Row 7 adds a cross-cutting sustain scenario that rows 1–6 under-specify: K8s + ecosystem lowers effort on individual scenarios but raises sustain burden across Volcano, Kueue, GPU Operator, KServe, GitOps, observability, and custom glue. Each new scenario often adds CRDs, Helm values, or webhooks [7]. On the MCP path, agents scaffold and register new tools under policy; human operators approve allowlist and high-blast-radius changes (Section 5.5). The claim is lower marginal extension cost (row 5) and a narrower integration surface than a full K8s + ecosystem install, not zero operations.

Volcano and Kueue cover queueing and placement policy; domain runtimes cover in-container L1 tuning where applicable. MCP agent coverage in rows 1–6 is an architectural target developed in Section 5 and validated analytically in Section 6, not a deployed result.

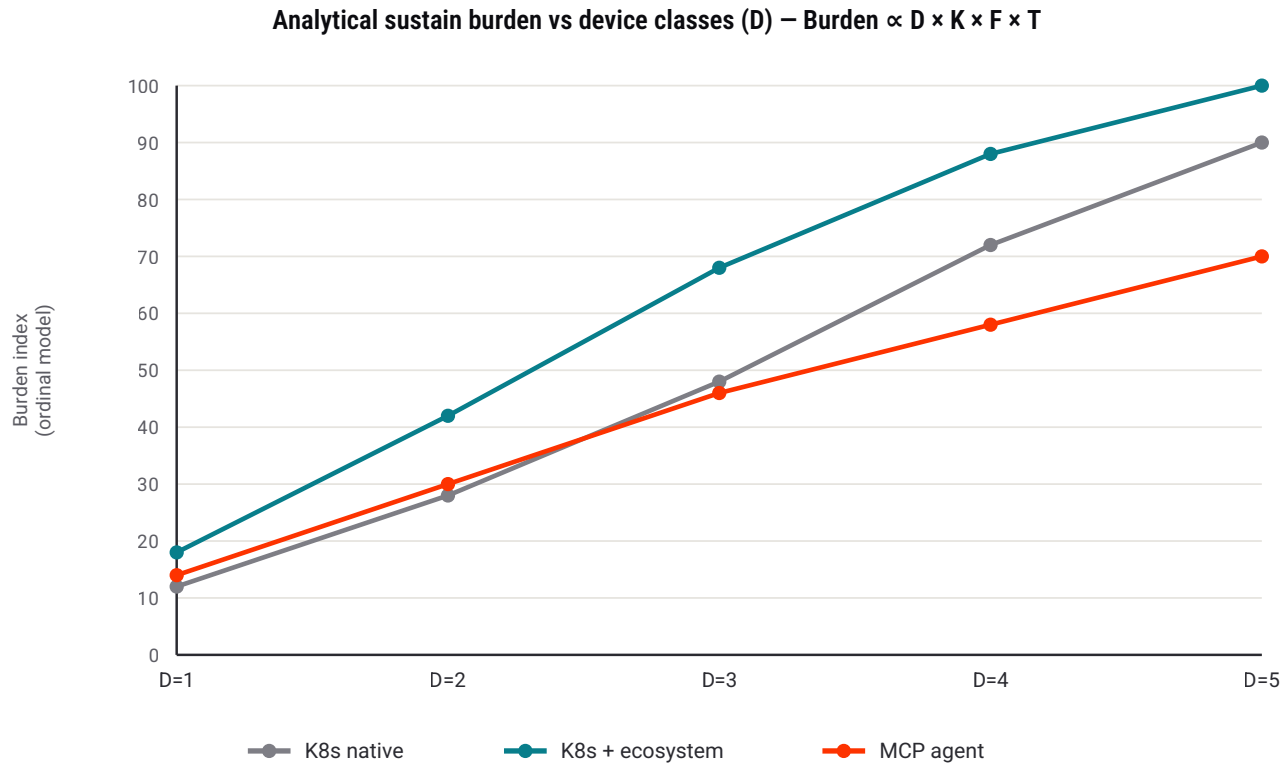

Figure 4. *Analytical sustain burden vs device-class count D from Section 6.3 (Burden ∝ D × K × F × T). Lines: K8s native (slate), K8s + ecosystem (teal), MCP agent (orange). Ordinal model, not measured person-hours.*

Figure 4 varies **D**, the count of distinct device classes in the fleet, from 1 to 5 on the x-axis while holding the other terms fixed. K is independent configuration dimensions per workload (image, runtime env, affinity, queue policy, autoscaling rules, and similar fields from Section 3's P1–P4). F is drift events per planning horizon (new hardware, traffic shape, model, or framework release). T is tenant or workload families sharing the platform. The curves are an ordinal slice of Burden ∝ D × K × F × T, not fitted person-hours; the steeper K8s + ecosystem line reflects stack sustain (Table 2 row 7). Section 6.3 formalizes the model and expands Table 2 with evidence ties.

5. MCP as Replacement Control Plane

Section 4 left a residual problem: even with Volcano, Kueue, KServe, and GitOps, mixed HPC and accelerated-compute fleets still need a cross-layer performance loop that humans cannot keep up with. The replacement architecture for H2 is an MCP-connected agent (policy, metrics, tool servers) as the operational control plane. Kubernetes drops out of the control path; node runtimes and hardware stay.

5.1 Why Replace, Not Augment

Another operator or scheduler plugin on Kubernetes still runs the same control law: reconcile desired configuration with observed state [10]. That works when hardware and workload objectives stay stable enough that YAML equivalence suffices. For varied HPC and mixed fleets under drift, the goal is domain-specific performance and cost, not YAML equivalence.

Kubernetes routes intent through a portable pod spec, scheduler, and CRI. Portability is the point. Mixed simulation, rendering, and GPU-accelerated serving need simultaneous tuning at application, container, and placement layers. Adding performance loops on top does not change the platform's objective; it adds more surfaces to maintain. The argument is for replacing the operational layer for this workload class, not bolting a sidecar that patches Deployments after reconcile.

Greenfield pools need not adopt a Kubernetes operational layer at all (Section 8.2). The mismatch between spec reconcile and runtime optimization grows with fleet heterogeneity and drift (Sections 3–4).

5.2 Kubernetes Function → MCP Tool Mapping

Replacement does not mean cloning every Kubernetes API. It means exposing the functions operators actually use (schedule, deploy, expose, store, secure, roll out) as tools the agent calls on node runtimes such as containerd, with policy on the tool allowlist instead of only on pod fields.

Table 1. Kubernetes control-plane functions mapped to MCP tool servers (illustrative names).

Kubernetes function	Typical K8s surface	MCP replacement tool(s)	Notes
Schedule / place	Scheduler, Volcano, Kueue, affinity	`schedule_workload`	Uses live metrics and topology, not static labels alone
Deploy / run	Deployment, GPU Operator, device plugins	`deploy_runtime`	Per-device-class image and runtime stack via containerd on target nodes
Expose / route	Service, Ingress, Gateway API	`expose_service`	Registers endpoints after health checks pass
Store / secrets	PV, Secret, CSI	`mount_volume`, `rotate_secret`	Same storage backends; different call path
Rollout / health	Deployment rollout, probes	`health_check`, `rollback`	Agent drives transitions; rollback is explicit
Policy / tenancy	RBAC, NetworkPolicy, OPA/Kyverno	Policy-as-code on tool allowlist	Open problem; see §5.7 and §8

Each tool lives on an MCP server with a schema the host can invoke [11]. The host runs the adaptation loop; servers perform infrastructure side effects. New placement logic can ship as a tool or server without a cluster-wide CRD. Section 6 compares extension cost against the Kubernetes ecosystem (Table 2).

5.3 Why MCP Versus a Generic Agent and Ad Hoc APIs

A team could point an LLM at bespoke REST endpoints. MCP's value is a standard host-tool contract (list tools, call tool, typed arguments) so primitives accumulate as composable servers instead of a growing internal API per scenario.

MCP addresses composability first: a standard host-tool contract so primitives accumulate as servers instead of a growing internal API per scenario. Agent safety depends on the policy layer around that contract (typed tool schemas, allowlists, argument bounds, approval gates, and audit logging on the host; Section 5.7), not on the wire format alone. The pattern mirrors the Language Server Protocol (LSP) for editor plugins: one contract let many editors share language servers instead of bespoke integrations per IDE, without LSP replacing IDE sandboxing. The optimizer is the agent plus policy plus metrics, not the protocol alone. Fixed autotuners, RL schedulers on Kubernetes, and workflow engines solve pieces of the problem inside reconcile; they do not change the closed variable from spec match to runtime performance.

5.4 Inner Loop: Workload Optimization

The inner loop monitors performance, decides, and acts against business requirements encoded in host policy:

1. **Observe:** metrics tools report throughput, latency, utilization (CPU or accelerator), queue depth, topology, and cost signals.
2. **Decide:** the agent compares observed state to policy objectives and selects the next action within allowlist and approval bounds.
3. **Act:** call tools (e.g., `deploy_runtime` with new env, `schedule_workload` for topology-friendly placement).
4. **Evaluate:** confirm business requirements (service level objectives, cost ceilings, priority rules) are met or rollback.
5. **Repeat** until requirements are met, bounds block action, or rollback fires.

This substitutes for Kubernetes reconciliation. Reconcile asks whether running state matches the spec; the inner loop asks whether business requirements are met on this hardware now, including service level objectives (latency, throughput, and related workload targets) and cost targets. Section 3.4 gives a concrete case where pods stay healthy while requirements fail. The inner loop is the architectural basis for **H2** (Section 6.1). Production autotuning systems such as Vizier show that closed loops can run at scale with clear objectives and guardrails [9]. Whether the same holds for full HPC infra control is an open question; Section 6 treats it analytically.

Most actions should stay at L1 runtime parameters (§5.6). L2 and L3 are rarer and gated by policy.

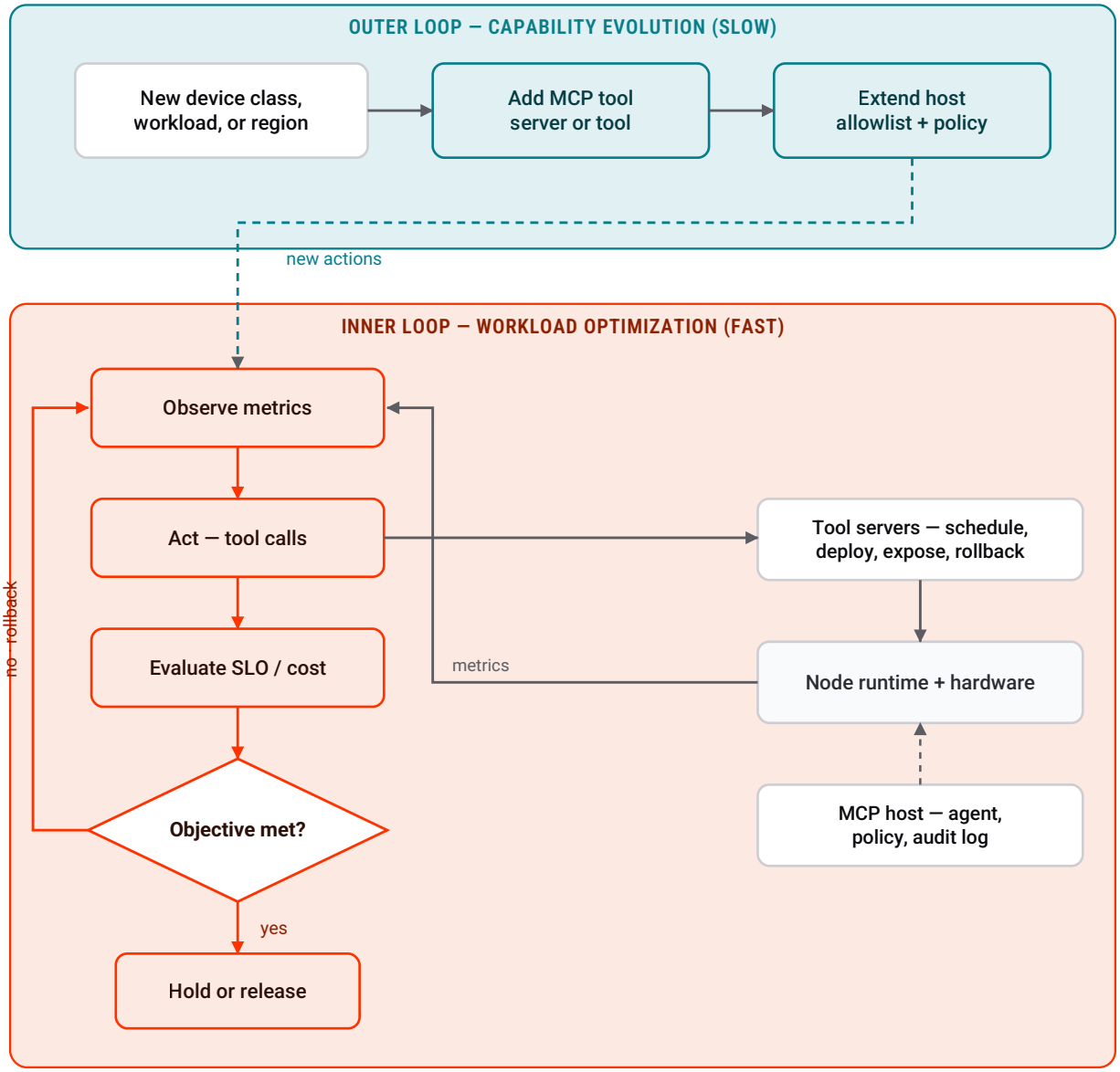

Figure 5. *MCP dual-timescale control. Inner loop (orange): observe via metrics tools, decide and act via tool servers, evaluate against business requirements in policy, repeat or rollback. Outer loop (teal): new capability ships as tool servers and allowlist extensions that feed the inner loop. Figure 3 (Section 4.3) compares inner-loop steps to GitOps; Figure 6 (Section 5.8) adds the Kubernetes reconcile timeline.*

5.5 Outer Loop: Capability Evolution

Figure 5 (outer band) runs on platform evolution timescales: agents extend the platform by registering new MCP tool servers (telemetry, canaries, device-class hooks) and updating the allowlist under policy, without redesigning orchestration for each case.

On Kubernetes, a new automation path often means a new operator, CRD, chart, and GitOps wiring (Section 4). On the MCP path it means a new server or tool. Agents and coding assistants can scaffold, test, and publish those tools; human operators set allowlist boundaries and approve L2/L3 or fleet-wide additions (Sections 5.6–5.7). The bet is lower marginal cost per new primitive than another operator stack. MCP spec and

community servers can co-evolve with agent capability, as CRDs did for Kubernetes, but oriented toward performance actions rather than portable spec fields [11].

5.6 Tuning Tiers L1 / L2 / L3

Actions are grouped by blast radius:

- **L1 (runtime):** batch size, concurrency, solver or framework parameters, routing weights. Fast to apply and revert; most drift events should stop here.
- **L2 (container / stack):** retag or rebuild image, change libraries or drivers visible to the workload, redeploy via `deploy_runtime`.
- **L3 (node / kernel):** driver pin, partition profile, kernel tunables. Privileged, rare, requires approval.

Policy maps agent identity and scenario to allowed tiers. Section 8 covers tenancy when L2/L3 cross boundaries without Kubernetes namespaces.

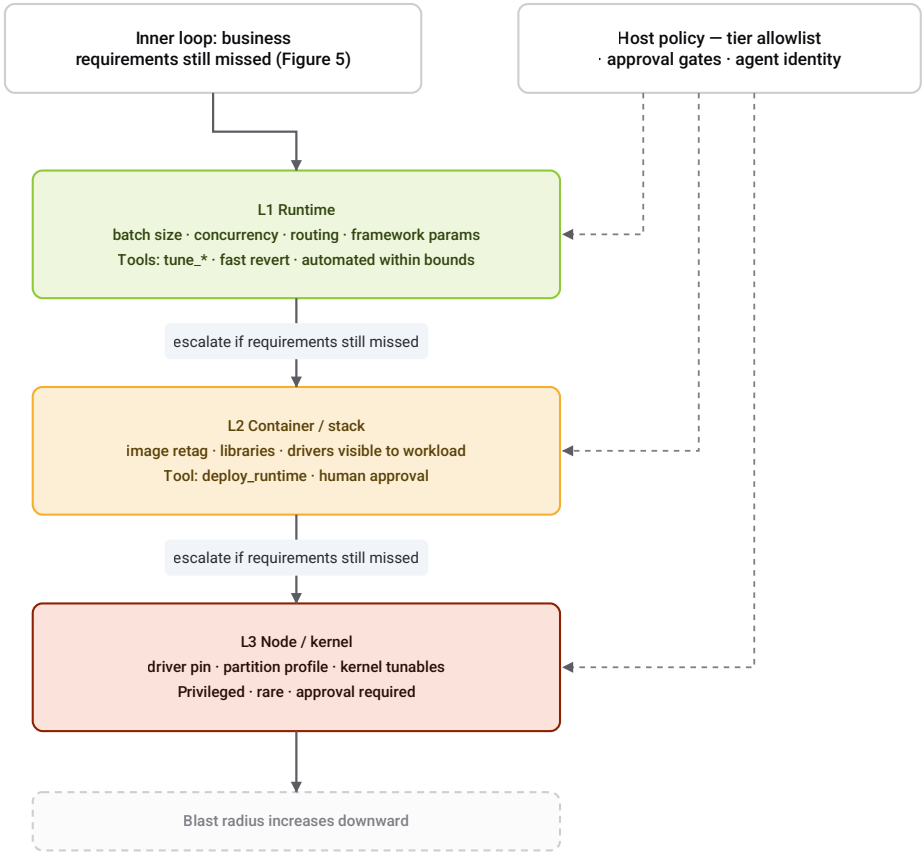

Figure 7. *Tuning tiers for MCP inner-loop actions. Blast radius increases from L1 through L3; host policy gates escalation and approval (Section 5.7).*

5.7 Safety and Failure Modes

Kubernetes can fail quietly: spec satisfied, SLO missed. Agents add wrong actions: bad tool arguments, injection through telemetry, tuning oscillation, or an L2 change that helps one device class and hurts another.

Required mitigations:

- **Rollback:** `rollback` tool and per-workload last-known-good state; no irreversible L3 without approval.
- **Bounds:** caps on concurrency, memory, replicas; objectives clipped by policy.
- **Approval:** human sign-off for L2/L3 or fleet-wide changes; L1 within bounds can run automated.
- **Audit log:** append-only record of observations, tool calls, requirement state, and outcomes (still immature compared to GitOps history; §8).

Multi-tenant isolation without Kubernetes namespaces and NetworkPolicy remains the hardest objection. MCP can still contribute at the control layer: per-tenant tool allowlists and host identity, schema validation on tool arguments before side effects, node- or pool-scoped agents with bounded blast radius, and network or storage isolation enforced at the runtime layer (CNI, volume mounts, credentials) while agents invoke only pre-authorized tools. That pattern parallels RBAC, admission control, and NetworkPolicy, but enforcement sits on tool calls rather than pod spec fields. These directions use the allowlist, policy, and audit model above; production-scale validation of tenancy on that path is open work (Section 8), not a blocker to pursuing MCP as the operational layer for accelerated fleets.

5.8 Reference Architecture

Figure 6 contrasts control timelines under GPU drift: Kubernetes spec reconcile (GitOps and controllers) versus MCP inner and outer loops from Figure 5.

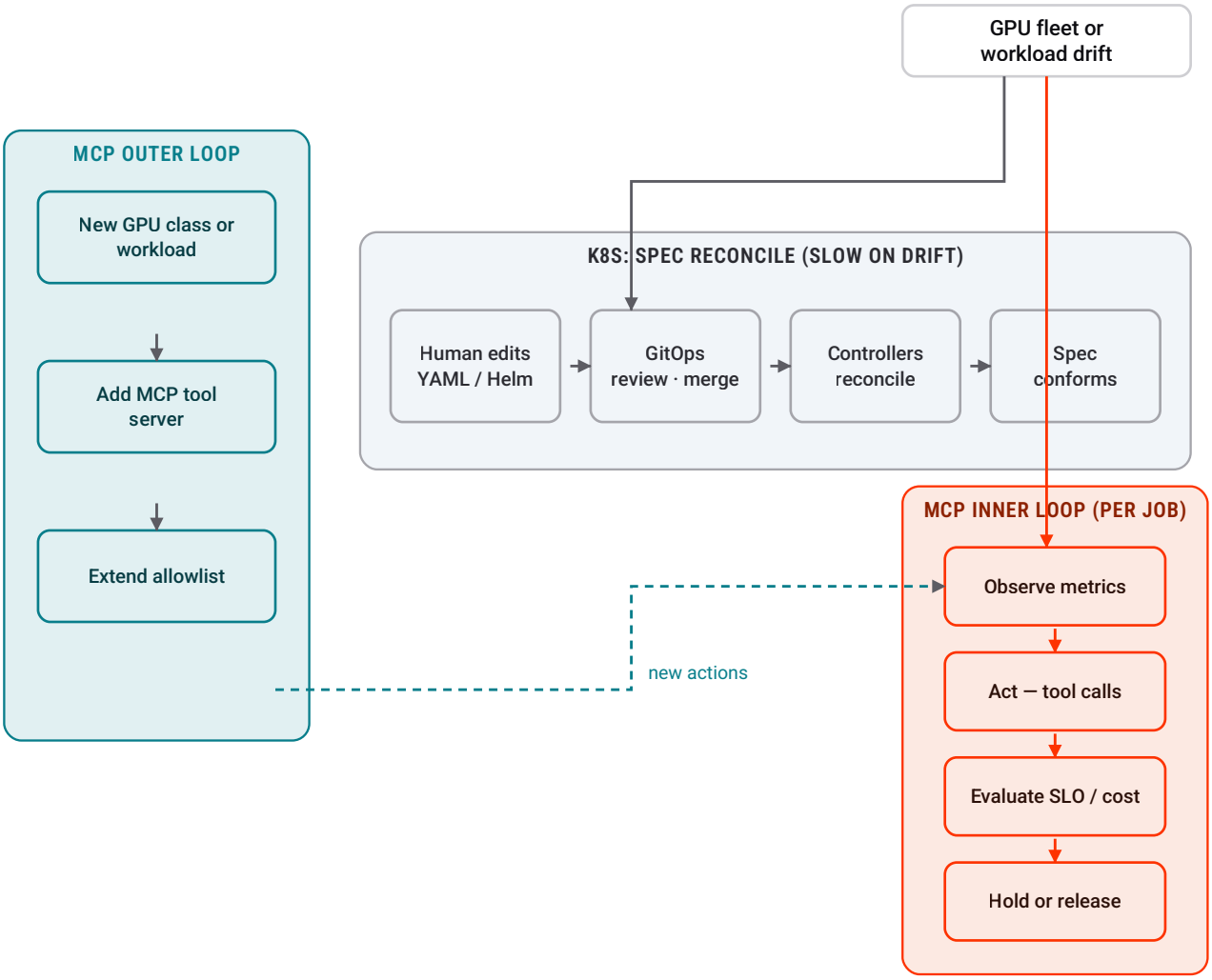

Figure 6. *Control timelines under workload drift: spec reconcile (top path) versus MCP inner and outer loops (Section 5.4–5.5).*

Figure 1 (Section 1) shows the replacement stack. The Kubernetes API server, scheduler, and controllers are outside this control path. Containers may still run under containerd; the contract is per-target optimization, not one portable pod spec for the whole fleet. Section 6 asks whether this design can match K8s + ecosystem outcomes with less integration work, and what experiments would be needed to know.

6. Evidence and Analysis

Evidence for Sections 3–5 draws on published performance studies, an automation parity comparison, an operational complexity model, and an illustrative MCP control trace. Numbers below come from cited literature or ordinal analysis unless marked illustrative.

6.1 Literature Synthesis

Three hypotheses organize the evidence (Section 1; Sections 6.2–6.4 develop each):

1. **H1 (heterogeneity tax):** Portable baselines underperform tuned deployments on mixed accelerator fleets. Developed in Section 3; Table 3 rows on batching, framework, topology, and partitioning cite published performance spreads.
2. **H2 (closed adaptation loop):** A policy-guarded monitor-decide-act loop can close part of that gap when objectives and guardrails are explicit. Proposed in Section 5.4; Vizier [9] and the illustrative trace in Section 6.4 test plausibility, not a deployed MCP stack.
3. **H3 (sustain burden):** Sustaining per-workload tuning through Kubernetes-shaped operations exceeds practical human capacity as device classes and drift events multiply. Argued in Sections 4 and 6.3; SRE toil literature [8], [15] and the burden model in Section 6.3 support ordinal sustain limits.

Table 3 maps selected sources to these claims.

Table 3. Literature synthesis: findings and claim support.

Source	Finding (published)	Supports
Orca [1]	36.9× throughput vs static-batch baseline at matched latency on same devices	H1, H2 (in-container scheduling sensitivity)
vLLM vs TGI comparison [2]	Large throughput spread on identical accelerators from framework/runtime choice	H1
MVP Factory mixed-fleet case [3]	Large tail-latency improvement only after topology, partitioning, and placement stack beyond one Deployment	H1, automation parity (integration cost on K8s)
vLLM / PagedAttention [13]	Continuous batching materially changes utilization; settings still workload- and device-sensitive	H1 (P2 partial automation in-pod)
NCCL documentation [14]	Collective performance depends on PCIe and network layout	H1 (P3)
NVIDIA MIG on Kubernetes [12]	Fractional partitioning adds device plugin, resource, and profile dimensions to portable specs	H1, automation parity
Google Vizier [9]	Black-box autotuning at scale with explicit objectives and guardrails	H2 (precedent for closed loops)
Google SRE / toil [8], [15]	Manual repetitive ops scale with service growth and crowd out engineering	H3
Borg / Kubernetes lineage [10]	Declarative reconcile optimizes spec conformance	Background for wrong closed variable
Volcano, Kueue, KServe [4]–[6]	Queueing, gang scheduling, replica scaling; not automatic cross-layer retune on drift	Partial K8s + ecosystem coverage

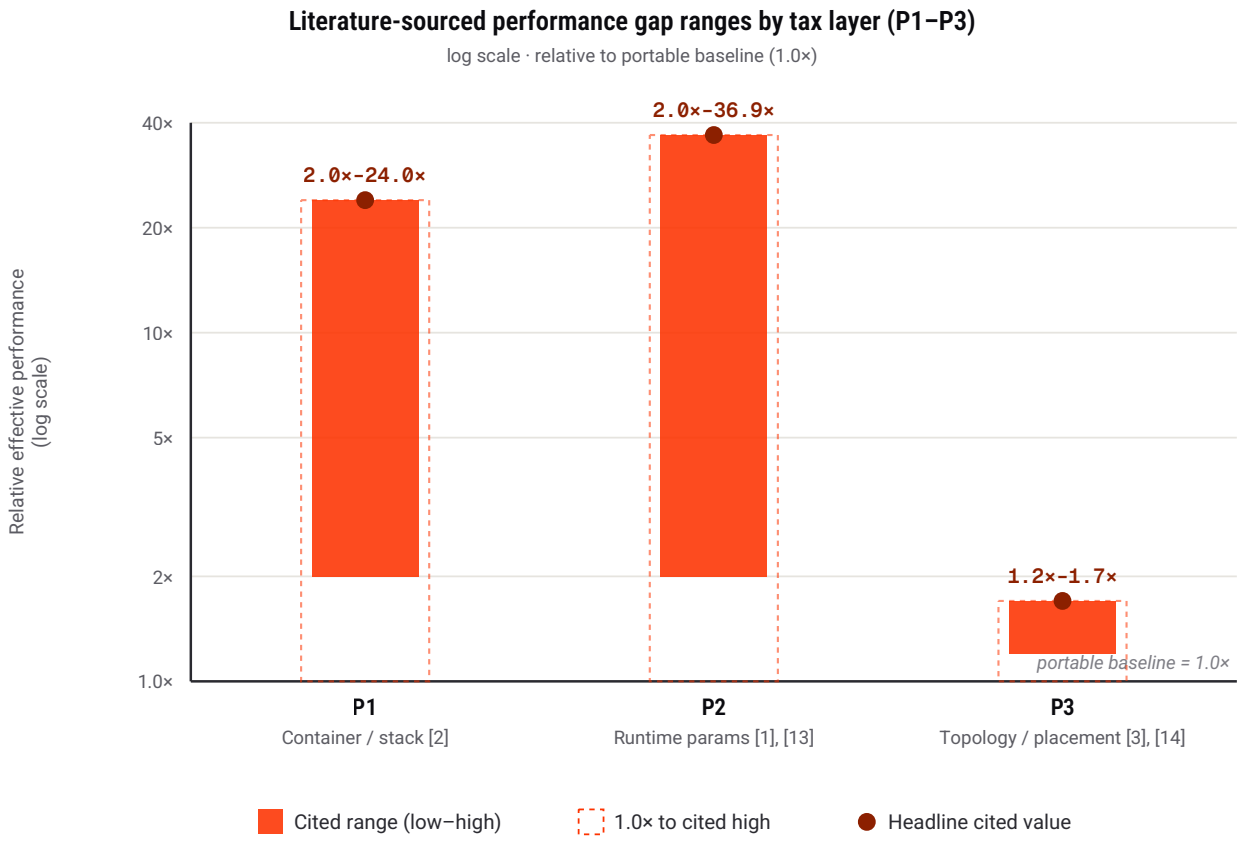

Figure 8. *Literature-sourced performance gap ranges for heterogeneity tax layers P1–P3 (Section 3.2), from Table 3 rows. Y axis starts at 0; dashed line marks portable baseline (= 1.0). Dotted outer bars: 1.0× to cited high; solid inner bands: published low–high ranges; dots: headline cited values. P1: framework and stack [2]; P2: runtime parameters [1], [13]; P3: topology and placement [3], [14]. Ranges aggregate published findings cited in Table 3.*

Two patterns recur. First, large spreads appear without changing hardware: batching, framework, and placement choices dominate [1], [2], [13]. Second, closing gaps on Kubernetes often requires stacking multiple ecosystem components [3], [4], [5], [12], which supports the automation parity argument in Section 6.2 rather than a claim that tuning is impossible on Kubernetes.

6.2 Automation Parity Analysis

The comparison method is fixed: same operational outcome across K8s native, K8s + ecosystem, and MCP agent. The ecosystem column receives full credit; row 7 (sustain integrated stack, cross-cutting) captures K8s + ecosystem maintenance load that rows 1–6 under-specify. Residual glue for cross-layer perf adaptation on drift is custom on Kubernetes (Prometheus rules, webhooks, operators) and architectural on the MCP path (Section 5.4 inner loop).

MCP agent “full” coverage in Table 2 rows 1–6 is an architectural target: achievable if tool servers and policy exist. Row 5 (add new automation capability) is where marginal extension cost differs most: new CRD/operator/Volcano config on K8s + ecosystem vs agent-scaffolded MCP tool server under policy on MCP agent [7], [11]. Row 7 marks K8s + ecosystem at H for stack sustain.

6.3 Operational Complexity Model

Section 4.2 argued informally that configuration burden grows with device classes, knobs, and drift. Section 6.3 models **H3** explicitly. Figure 4 varied **D** while holding K, F, and T fixed; sustain burden scales as:

$$\text{Burden} \propto D \times K \times F \times T$$

where **D** is distinct device classes in the fleet, **K** is independent configuration dimensions per workload (image, runtime env, affinity, queue policy, autoscaling rules, and similar fields from Section 3's P1–P4), **F** is drift events per planning horizon (new hardware, traffic shape, model, or framework release), and **T** is tenant or workload families sharing the platform.

Kubernetes ecosystem tools add terms to K (Volcano queue definitions, Kueue ResourceFlavors, GPU Operator policies, KServe annotations) even when they reduce work in one dimension. That is consistent with Table 2: K8s + ecosystem lowers effort in scheduling and device expose while leaving cross-layer perf adaptation at M–H unless teams build glue; Table 2 row 7 scores stack sustain at H for that column.

The model is not fitted to measured person-hours. It explains why mature shops run multiple node pools and golden images yet still hit sustain limits when D and F rise faster than platform team bandwidth [8], [15]. The model predicts lower burden when D, F, or heterogeneity are small; that is the boundary condition for **H3**, not a refutation of MCP on mixed accelerator fleets. Primary evaluation would track edit frequency, PR latency, and incident counts; this paper stops at ordinal and structural analysis.

6.4 Illustrative MCP Trace

The following trace tests **H2** plausibility. It is illustrative, not measured. It shows how the inner loop from Section 5.4 could run on a mixed fleet with two accelerator generations serving inference. Pool A runs a newer device class; Pool B runs an older class. One portable image and Deployment-shaped spec deploy everywhere. Traffic shifts toward longer context requests; P99 latency rises on Pool B while utilization stays flat and reconcile reports healthy pods.

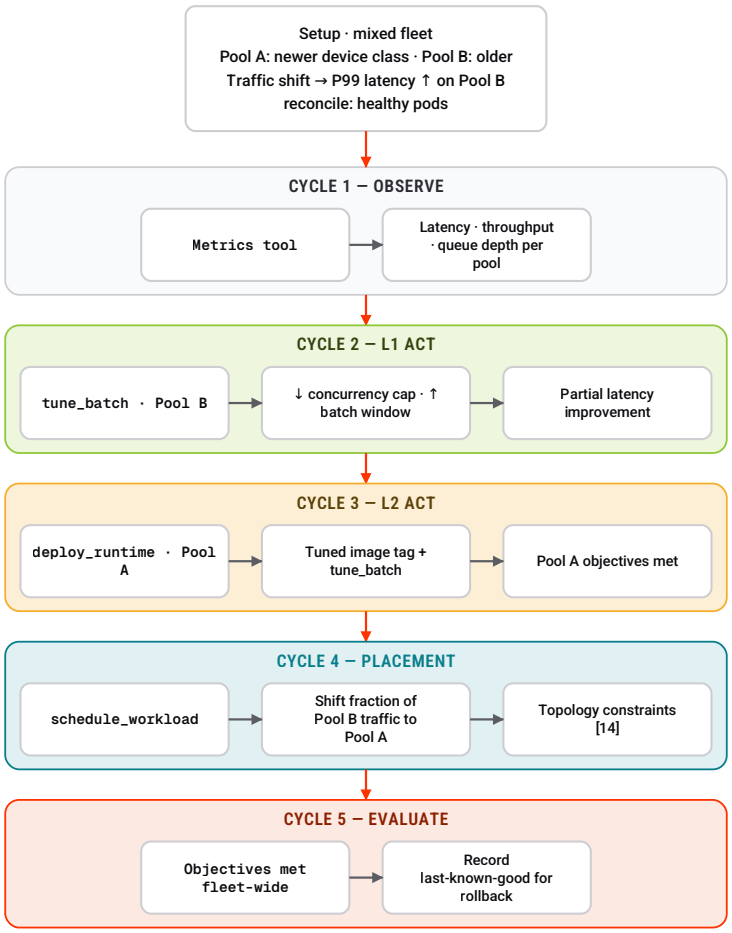

Figure 9. *Illustrative inner-loop trace for **H2**: mixed-fleet drift response across five cycles without a GitOps edit per step. Green band: L1 act; amber: L2; teal: placement; orange: evaluate.*

Pool A (newer device class) and Pool B (older class) share one portable spec until tiered tool actions restore objectives.

Figure 9 walks the escalation path from Section 5.6. The agent first observes latency, throughput, and queue depth per pool, then applies **L1** `tune_batch` on Pool B (lower concurrency cap, wider batch window). Latency improves partially, but Pool A throughput drops slightly. Policy allows **L2** escalation: `deploy_runtime` on Pool A with a device-class-tuned image tag, plus `tune_batch` on Pool A. Objectives are met on Pool A; Pool B remains above target until `schedule_workload` shifts a fraction of traffic to Pool A headroom under topology constraints [14]. A final evaluate step records last-known-good state for `rollback`; if placement had regressed Pool A, the agent would revert before trying another action. Each step is a tool call in the inner loop, not a GitOps edit.

This trace does not prove H2. It shows how cross-layer actions compose without a GitOps edit per step. Vizier-style autotuning [9] supports plausibility for L1 search; L2/L3 fleet actions remain hypotheses until deployed with policy and audit tooling (Section 8).

6.5 Limitations

Cited literature, ordinal comparison, burden modeling, and one illustrative trace support plausibility for H1, H2, and H3; this is not a production bake-off. Limits:

- **No primary MCP deployment benchmarks.** Table 2 MCP agent effort classes and the trace above are analytical. A prototype comparing drift response latency and integration artifact count against K8s + ecosystem would be the natural next step.
- **Agent cost and nondeterminism.** LLM-driven hosts add token cost, variable latency, and argument errors that deterministic autotuners avoid [9]. Policy engines or bounded tool schemas may reduce but not remove that gap.
- **Enterprise constraints.** Regulated environments may mandate declarative audit trails GitOps already provides. Agent decision logs are immature substitutes (Sections 5.7 and 8).
- **Fairness to Kubernetes.** Volcano, Kueue, and KServe solve real problems. This paper claims a residual cross-layer performance loop and marginal extension cost, not absence of Kubernetes automation.
- **Scope.** Evidence targets heterogeneous GPU and mixed HPC fleets, including build-time platform choice on greenfield pools (Section 8.2).

Section 7 places this work against schedulers, autotuning systems, and agent infrastructure prior art. Section 8 covers objections, open problems, and deployment paths.

7. Related Work

Prior work automates pieces of heterogeneous GPU operations on Kubernetes or optimizes workloads inside containers. The claim here is narrower: the control objective and operational layer should change for accelerated fleets, not that existing tools are useless. Table 4 summarizes differentiation.

Table 4. Related work vs replacement thesis.

Prior art	What it automates	Gap vs this paper
Borg / Kubernetes [10]	Declarative reconcile, portable pod abstraction, multi-tenant isolation	Optimizes spec conformance; perf tuning is human-driven YAML edits
Volcano [4]	Batch/gang scheduling, queues, topology-aware scheduling plugins	Placement policy, not cross-layer perf loop on drift
Kueue [5]	Job queueing, quotas, ResourceFlavor admission by device labels	Admission and fair-share; does not retune runtime/image on drift
kube-scheduler + device plugins	Default placement, extended resources, GPU expose	Static scheduling inputs unless manifests change
GPU Operator, NFD, MIG [12]	Driver install, device labels, fractional partitioning	Reduces bring-up toil; image/runtime matrix still human-maintained
KServe + KEDA [6]	Serving autoscale on inference metrics	Scales replicas, not batch/CUDA optimality per device class
vLLM / Triton in-pod batching [13]	L1 runtime tuning inside container	Partial P2; cross-pool/image/placement still external
KubeFlow, Ray, Argo Workflows	ML pipelines, distributed training, DAG batch	Workflow orchestration inside reconcile model
Karpenter, Cluster Autoscaler	Node provisioning by class	Adds nodes; tuned configs per class still separate artifacts
GitOps (Argo CD, Flux)	Declarative deploy, rollback history	Human-gated change path; weak perf-triggered automation
Google Vizier [9]	Black-box autotuning with objectives/ guardrails	Application-level optimizer; not infra control plane
Kubernetes Operators [7]	Domain reconcile via CRDs	Each scenario adds YAML surface; marginal extension costly
MCP specification [11]	Standard host-tool wire protocol	Not an orchestrator; agent + servers form control plane here

Numerical optimizers and in-container batchers are not the target. The argument is for composable infrastructure tools plus an agent policy loop that reasons over heterogeneous fleet state at two timescales (Section 5). Schedulers and operators remain relevant as partial solutions on Kubernetes; the thesis is that sustaining cross-layer adaptation through them scales poorly compared to MCP-shaped extension.

Dual-timescale adaptation distinguishes this work from single-loop autotuning: inner perf optimization and outer capability growth via new tool servers substitute for CRD/operator sprawl. Section 8 addresses objections that amount to reimplementing the Kubernetes ecosystem wholesale (primarily where that ecosystem is already installed; Section 8.2).

8. Discussion

Objections to full replacement and remaining open problems follow, along with greenfield and brownfield deployment paths.

8.1 Objections and Open Problems

Borg and Kubernetes won for good reason. Declarative reconcile, portable pods, and namespace isolation solved multi-tenant general-purpose orchestration at scale [10]. That era assumed relatively homogeneous racks and human SRE density per service. Heterogeneous accelerator fleets with continuous perf drift shift the bottleneck from "did the spec apply?" to "did the job meet its target on this hardware?" The argument is workload-class specific, not a claim that reconcile was wrong. Teams still need to close the gaps below. For each: the concern, what MCP already offers (Section 5), and what remains unfinished:

- **Multi-tenant isolation and injection.** NetworkPolicy, RBAC, and admission control took years to mature on Kubernetes; tool allowlists are not drop-in substitutes. On MCP, per-tenant tool allowlists, argument schema validation before side effects, pool-scoped agents, and runtime isolation at CNI, volumes, and credentials cover the same ground at the tool layer (Section 5.7). Production-scale tenancy validation and strict L2/L3 approval gates remain open (Section 5.6).
- **Audit and regulatory parity.** Immutable declarative history is a compliance asset; agent decision logs are immature versus Argo CD revision history. Append-only host logs can record observations, tool calls, requirement state, and outcomes (Section 5.7). Integrating those logs with compliance retention and review workflows remains unfinished; regulated environments may block agent-driven change until parity is demonstrated.
- **Reimplementing the Kubernetes ecosystem.** Replacing ten years of CNCF projects sounds unrealistic. MCP maps functions incrementally to tools (Section 5.2), extends capability through the outer loop without a fleet-wide portable spec, and targets lower marginal extension cost (Table 2 row 5). Day-one MCP stacks still do not match full K8s + ecosystem breadth—though where no such stack is installed, this objection largely does not apply (Section 8.2).
- **Agent latency, cost, and control policy.** Reconcile loops run at millisecond scale; LLM hosts add token cost and nondeterminism [9]. Agent cycles run on drift timescales where GitOps latency already dominates (Section 4.3), using scripted or policy-engine agents for bounded L1 tuning and humans in the loop for L2/L3. The right deterministic-vs-LLM split per workload class is unresolved.
- **Safety certification.** Enterprise and regulated buyers need certified controls on high-blast-radius automation. Policy layers already specify rollback, objective bounds, human approval for L2/L3, and per-workload last-known-good state (Section 5.6). Formal certification of agent-driven orchestration does not exist.
- **Primary evaluation.** Table 2 and Section 6 claims remain ordinal without deployed evidence. The inner and outer loops in Sections 5.4–5.5 define what a prototype would exercise. A head-to-head MCP agent stack against K8s + ecosystem on drift latency, integration artifact count, and recovery success rate is still required.

8.2 Greenfield Deployment

Where no Kubernetes operational layer exists on accelerated pools, the replacement framing simplifies to adoption, not migration. New GPU clusters, cloud accelerator capacity, and HPC sites that still run batch schedulers or bare-metal provisioning face a build-time choice: adopt Kubernetes because it is the default platform pattern, or adopt MCP on the performance-first path this paper describes. Section 4 compared those stacks on the same operational outcomes; greenfield teams are not sinking cost in Volcano, Kueue, GPU operators, or chart sprawl they must later unwind. They can extend composable tool servers as device classes and workload types appear (Section 5.4 outer loop) without first standardizing on a fleet-wide portable spec.

The thesis does not require Kubernetes underneath. Containers, hypervisors, and node runtimes remain; only the user-facing reconcile layer is optional on day one.

8.3 Transitional Coexistence

Brownfield sites that already run Kubernetes face a different path: phased migration of the user-facing control plane rather than a single fleet-wide cutover. Workloads that pay the heterogeneity tax first benefit most from perf-first control during that transition. Running dual stacks during migration is realistic, not the end state this paper prefers long term.

Kubernetes as a legacy substrate under containerd is optional. The thesis targets removal from the user control path, not mandatory same-day decommission of every cluster API.

9. Conclusion

Heterogeneous GPU infrastructure exposes a structural mismatch with Kubernetes-shaped operations. Portable manifests and golden images keep fleets running; they do not keep them optimized as device classes, frameworks, and workloads drift. Section 3 named that gap the **heterogeneity tax** (**H1**), decomposed it into stack, runtime, topology, and scheduler layers, and showed that the Kubernetes ecosystem reduces but does not eliminate the sustain burden of cross-layer tuning at human scale (**H3**).

An MCP-connected agent control plane offers a different contract: composable tool servers, policy guardrails, and dual-timescale adaptation that closes on business requirements (throughput, latency, cost, priority) instead of spec conformance alone. That inner loop is the proposed response to **H2**. Mixed fleets need perf-first loops that monitor, decide, and act as the primary operational layer rather than reconcile alone toward a fixed portable spec. MCP is the integration substrate; the agent and tools are the control plane.

Sections 3–6 make the case for **H1**–**H3**: the heterogeneity tax is real on mixed fleets, human-scale sustain through Kubernetes-shaped operations breaks down under drift, and a policy-guarded MCP loop is the viable alternative control law. Section 8 takes on the real objections—tenancy, audit, safety, ecosystem breadth, and agent cost—and pairs each with MCP-side responses in Section 5 rather than reasons to stay on reconcile-first paths. For heterogeneous accelerated infrastructure governed by business requirements on throughput, latency, cost, and priority, MCP-mediated adaptation is the better operational choice than accumulating more YAML around golden images. Greenfield sites can adopt that model without the integration debt Section 4 catalogs; brownfield sites can phase migration as Section 8.3 describes.

Spec conformance deploys workloads; guarded adaptation keeps them optimized on the hardware they actually run. Reconcile-first operations excel at the first job but break down on the second once fleets mix device classes, frameworks, and shifting business requirements. In heterogeneous environments, MCP is the operational layer built for both—and the right choice for heterogeneous fleets.

10. References

[1] G.-I. Yu, J. S. Jeong, G.-W. Kim, S. Oh, B. Jung, D. Shin, and J. W. Lee, "Orca: A Distributed Serving System for Transformer-Based Generative Models," in *Proc. USENIX OSDI*, 2022. [Online]. Available: https://www.usenix.org/system/files/osdi22-yu.pdf

[2] "Comparative Analysis of LLM Inference Serving: vLLM and HuggingFace TGI," arXiv:2511.17593, 2025.

[3] "K8s GPU Inference Scheduling: Topology and MIG," MVP Factory, 2025. [Online]. Available: https://mvpfactory.io/blog/kubernetes-pod-scheduling-for-gpu-accelerated-ml-inference-topology-aware/

[4] "Volcano: Cloud Native Batch System," CNCF Volcano Project. [Online]. Available: https://volcano.sh/

[5] "Kueue: Kubernetes-native Job Queueing," Kubernetes SIG Scheduling. [Online]. Available: https://kueue.sigs.k8s.io/

[6] "KServe Autoscaling Documentation," KServe Project. [Online]. Available: https://kserve.github.io/website/docs/model-serving/generative-inference/autoscaling

[7] J. Dobies et al., *Kubernetes Operators*. O'Reilly Media, 2020.

[8] Google SRE Team, *Site Reliability Engineering*. O'Reilly Media, 2016.

[9] D. Golovin et al., "Google Vizier: A Service for Black-Box Optimization," in *Proc. ACM KDD*, 2017.

[10] A. Verma et al., "Large-scale cluster management at Google with Borg," in *Proc. ACM EuroSys*, 2015.

[11] "Model Context Protocol Specification," Anthropic, 2025. [Online]. Available: https://modelcontextprotocol.io/specification/2025-11-25

[12] NVIDIA, "MIG Support in Kubernetes." [Online]. Available: https://docs.nvidia.com/datacenter/cloud-native/kubernetes/latest/

[13] W. Kwon et al., "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *Proc. ACM SOSP*, 2023.

[14] NVIDIA, "NCCL: NVIDIA Collective Communications Library." [Online]. Available: https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/

[15] B. Beyer et al., "Invent More, Toil Less," *USENIX ;login:*, 2016.