Slinky: A Structural Exploration of Converged Infrastructure

Testing the Composite Pattern on Local Kubernetes Clusters

Xuanlin Zhu
Research Student

Abstract

The separation of HPC (Slurm) and Cloud-Native (Kubernetes) infrastructure is a known inefficiency in academic computing. While the concept of merging them is not new, the structural implementation remains challenging. This document reports on a local **Proof-of-Concept (PoC)** designed to test a specific hypothesis: Can the **Composite Design Pattern** provide a valid structural basis for this integration? Using the Cluster API (CAPI) on a local Docker environment, I constructed a minimal viable prototype to verify if a single node could structurally satisfy two schedulers.

1. Motivation: The Structural Hypothesis

Conversations with infrastructure engineers in May 2025 highlighted a gap between the *desire* for converged infrastructure and the *difficulty* of implementing it. The industry had already recognized this need by 2024, with Microsoft Azure and others exploring potential solutions. My contribution was not to invent the integration concept, but to propose and test a specific structural mapping:

Hypothesis: The **Cluster API (CAPI)** can serve as the "Composite" layer, masking the complexity of heterogeneous workloads (Slurm vs. Kubernetes) from the infrastructure provider.

When I suggested applying the Composite Pattern during early discussions, colleagues noted that "it's easy to say but not easy to implement." This PoC emerged from that challenge—could I design and validate the structural plumbing to make this theoretical pattern work in practice?

2. Experimental Design

To test this hypothesis without access to production cloud resources, I built a local simulation using **Cluster API Provider Docker (CAPD)**.

The experiment focused on two structural couplings:

1. **Topology Mapping:** Can we define a CAPI `MachinePool` that physically accepts both `kubelet` (K8s) and `slurmd` (HPC) processes?
2. **Event Translation:** Can a Slurm queue metric triggering a Prometheus alert successfully drive a KEDA scaler to provision a new Docker container that joins both clusters?

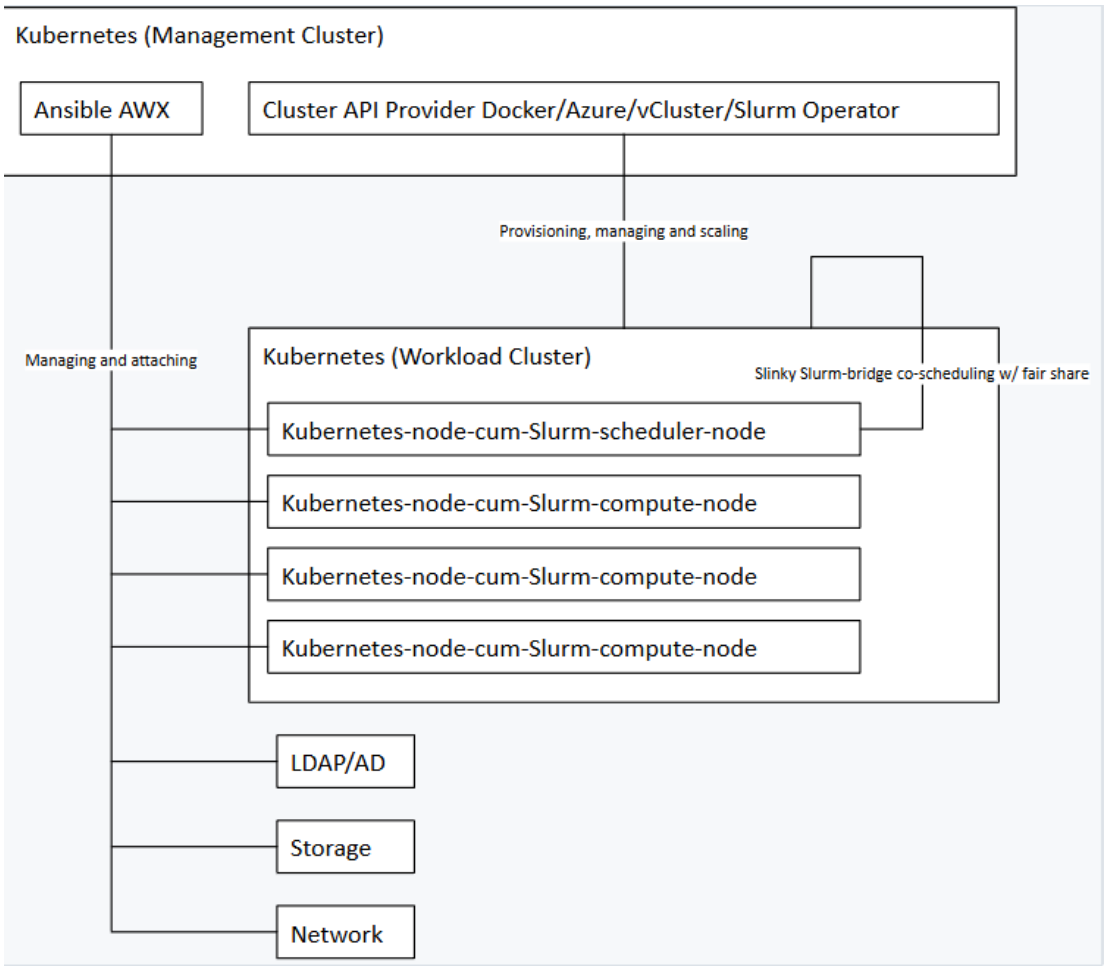

Figure 1: **Experimental Topology.** A local simulation where CAPI manages the lifecycle (top) and Ansible mimics configuration management (left). The workload cluster (bottom) demonstrates composite nodes serving both schedulers.

3. Implementation Details (Local PoC)

The implementation relied on rapid prototyping ("vibe coding") to glue existing open-source components together for validation. My specific contributions to the structural design included:

3.1. Topology Definition (CAPI Resources)

I mapped the logical roles of an HPC cluster to specific CAPI resources to test stability during scaling:

- **Control Plane (Static):** Defined as a `MachineDeployment` hosting the Slurm scheduler, pinned via `nodeAffinity` labels to prevent autoscaler termination.
- **Compute Plane (Elastic):** Defined as a `MachinePool` with autoscaling annotations, allowing nodes to scale from 0 to N based on queue pressure.

This separation was crucial for preventing the scheduler from being terminated during scale-down events—a structural detail that wasn't immediately obvious from the high-level pattern.

3.2. The "Glue" Logic (Dynamic Inventory)

To bridge Kubernetes's declarative model with Ansible's imperative configuration management, I implemented a Python script (`run.py`) that:

1. Queries the Kubernetes API to discover new CAPI-provisioned nodes
2. Parses specific labels (`node-type: compute` vs `controller`)
3. Dynamically generates an Ansible inventory for AWX to bootstrap nodes with SSSD/LDAP and shared storage

This confirmed that the required *information flow* for convergence is possible, though the production implementation would require more robust error handling and synchronization logic.

3.3. Autoscaling Triggers

I configured a KEDA `ScaledObject` to monitor the Prometheus metric for pending Slurm jobs (`slurm_partition_jobs_pending_total`). In local tests, submitting a dummy batch job successfully triggered the creation of a new simulated node that joined both the Kubernetes and Slurm clusters.

4. Limitations & Lessons Learned

While this PoC demonstrates that the structural logic is sound in a controlled environment, it does **not prove production viability**. As colleagues rightly noted: "Even if it seems to run on PoC in your local computer, it doesn't mean it can actually work."

Key limitations I discovered:

- **Local vs. Cloud:** Managing Docker containers on a laptop doesn't account for the latency, network partitioning, or race conditions of a real cloud environment (e.g., Azure).
- **Synchronization Complexity:** I learned that "dual-plane" management introduces significant timing challenges between Ansible configuration and Kubernetes readiness that my simple Python script cannot fully resolve.
- **State Management:** The prototype doesn't handle failure modes like nodes becoming unreachable during configuration, which would require more sophisticated reconciliation logic.

What I *did* prove: The Composite Pattern is worth considering—it's not fundamentally broken at the structural level. What remains unproven: Whether the coordination overhead and operational complexity make it practical at scale.

5. Conclusion

This project was a learning exercise in distributed systems and infrastructure orchestration. It validated that the Composite Pattern represents a *worthwhile direction* for investigating converged infrastructure, but successfully productizing it remains a significant engineering challenge well beyond the scope of this exploratory study.

The experience taught me as much through the gaps I discovered as through what worked: understanding why certain design decisions matter (like the MachineDeployment/MachinePool separation) only emerged through hands-on experimentation.

Acknowledgment & Availability

This feasibility study emerged from architectural discussions with infrastructure engineers at Microsoft Azure. My specific contribution was the structural design and local validation of the Cluster API integration presented here.

Note: As this work represents a research prototype developed within the context of an ongoing industry collaboration, specific source code implementation details are proprietary and not currently available for public release.