Towards a Systems Foundation for Agentic Cloud Management

Minghao Li[1], Ziqian Liu[1], Ziyu Mao[1], Daqian Ding[1], Yu Kang[2], Qingwei Lin[2], Tianyin Xu[3], Yiming Qiu[1]
[1]The University of Hong Kong [2]Microsoft [3]University of Illinois Urbana-Champaign

Abstract

Agentic cloud management is emerging as a practice to automate laborious operations, minimize toil, and improve responsiveness. Despite the rapid development of autonomous management agents, we argue that the fundamental missing piece is a systems foundation to enable safe, effective operations across agents and between agents and human operators. In this paper, we advocate for the need of such a systems foundation and share our efforts on developing CLOUDWEAVER, an agentic management substrate that works across existing cloud-user interfaces and future agent-native interfaces. Specifically, we discuss how CLOUDWEAVER (1) scopes the context of individual agent sessions with local views of cloud resources and (2) coordinates concurrent management operations on shared cloud resources. CLOUDWEAVER offers strong safety guarantees and attributable feedback in the presence of conflicting intents, while preserving concurrency between independent operations. We validate CLOUDWEAVER using a representative Azure API workload.

1 Introduction

Cloud infrastructures form the backbone of modern IT, but are notoriously hard to manage due to the fundamental split between the cloud providers and their tenants [34]. The tenants manage cloud resources indirectly through multi-modal management interfaces across both low-level management actions (e.g., SDK or CLI [25]) and higher-level Infrastructure-as-Code (IaC) workflows [8, 10]. The practice enables tenants to control complex cloud resources without owning the underlying datacenter infrastructures.

However, this abstraction breaks at the boundary between scoped user-management intent and shared cloud infrastructure. A tenant often means parties with distinct resource domains, operational roles, and ownerships [17]: one team may repair network topology, another auto-scale compute capacity, and another configure security policies. Each party acts through a scoped stream of observations and operations, without a complete and authoritative view of the tenant's shared state. Once these operations reach the provider backend, their scope is lost: they become individual API calls whose effects unfold over shared resources [18]. The caller retains the intent but not the feasibility of intent execution, while the provider sees the shared state but not the context that relates execution to the intent. Human operators bridge this gap through workflow orchestration, change windows, and manual recovery, leaving coordination as external conventions rather than a guarantee of the cloud-user interface.

Agentic cloud management now faces a fundamental choice: should AI agents be expected to reconstruct the missing coordination between management intent and cloud execution through reasoning, or should the cloud management stack provide it as a systems abstraction? We argue for the latter. Better AI may formulate stronger plans and recover from more failures, but they still reason from the context and feedback exposed by the system [21, 23, 39, 43]. Asking probabilistic AI agents to reconcile conflicting intents by inferring session scope, concurrent effects, and execution outcomes from incomplete state and ambiguous feedback only compounds uncertainty. Coordination should therefore shift away from external best practices and be built directly into the cloud management abstraction itself.

In this paper, we advocate for a systems foundation to enable agentic cloud management. Specifically, we envision a transparent *coordination contract* beneath existing cloud-user interfaces and future agent-native interfaces, which offers the execution semantics required for reliable closed-loop agent control. The contract binds each session's context and actions to authoritative shared state, commits their effects through semantic transactions, and returns attributable feedback that renews the context of affected sessions. Realizing this vision would bridge the gap between scoped intent and shared state, making their correspondence a first-class systems invariant.

To obtain coherent reasoning context, the contract needs to enforce strong isolation across sessions. This is nontrivial because each session-scoped view is a live execution status that must be constructed online, not from a snapshot [42]. Furthermore, to be transparent to interface choices, coordination must sit at the boundary of provider APIs, where intents arrive as low-level operation streams. The session view must then play two important roles: (1) before execution, it checks admissibility: each incoming operation must be checked against the session's current context to decide whether it can extend the view with valid resources, dependencies, and ownership, and (2) after admission, it tracks feedback: the same view must track pending, committed, failed, and shared effects so that the session can trust the observation. If either role is wrong, the session may reason about dependencies that never existed or act on resources it does not own.

Safe coordination of concurrent operations is another challenge. A locally valid intent may not be globally executable

as operations admitted by different sessions may still conflict with each other. The interface therefore needs an authoritative global view: a single source of truth that records resources, dependencies, and operations across all sessions. At first glance, concurrency control over this view may seem as simple as enforcing mutual exclusion on each cloud resource. In reality, it must reason about semantic effects: deleting a resource may invalidate another session's dependency; updating a parent resource may block child-resource creation; and quota reservations may make otherwise independent intents incompatible. These effects determine which operations can run together, which must wait, and which should be denied before reaching the provider. Thus, concurrency requires a unified mechanism that makes global admission decisions with semantic-aware execution policies.

To address these challenges, we propose to build coordination atop three architectural pillars. First, the system must preserve management intent as it is lowered into provider API calls. It interprets each session's API stream as semantic deltas over cloud resources, dependencies, lifecycle state, and capacity, thereby recovering the structure that conventional interfaces discard at the provider boundary. Second, session-local and global views are coupled by a bidirectional projection: every local object and pending effect has a global counterpart, while changes in authoritative shared state are reflected back into the session context. This projection makes the correspondence between reasoning and execution a continuously maintained invariant rather than a condition repaired after drifts. Third, concurrency control follows the semantics of each semantic delta. The transaction layer acquires structural rights for topology and lifecycle effects and reserves consumable rights for capacity and quota, allowing independent transitions to proceed while delaying those whose effects truly conflict. Together, these principles preserve intent across interface lowering, ground every session in shared execution, and commit only globally valid cloud-state transitions.

We are building a substrate to realize the envisioned systems foundation, named CloudWeaver. We discuss its core design with preliminary results, and discuss how it could enable future agentic cloud management endeavors.

2 Background

Cloud Infrastructure Management. The management is exposed through a variety of cloud-user interfaces. Provider APIs define the authoritative operations over cloud resources. Most users invoke APIs indirectly: SDKs package them into libraries [20], CLIs expose them as commands [35], and web consoles support ClickOps with visual context [41]. Above them, Infrastructure-as-Code (IaC) has become the dominant structured approach [32, 34]. IaC systems such as Terraform [10] and Pulumi [8] let users specify desired cloud state, compare it against known state, construct an execution plan, and schedule the corresponding API operations. Despite their different abstractions, these interfaces ultimately converge at the same boundary: they translate tenant management intent into provider operations over shared cloud state.

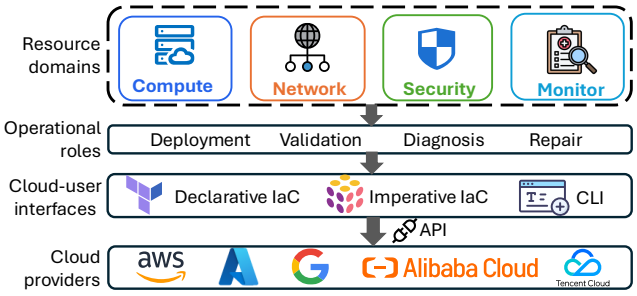

Figure 1: Modern cloud management stack.

This stack has been increasingly used by AI agents. Platforms are exposing infrastructure state, operation context, and actions to agents. Providers such as Azure, AWS, and GCP now offer agentic services for operating and troubleshooting cloud systems [1, 3, 5], while IaC ecosystems are introducing MCP interfaces that connect LLMs to infrastructure metadata and configuration workflows [7, 9]. Recent AI benchmarks are also moving towards system operations and infrastructure management [24, 28, 29, 38]. Together, these efforts point to a new paradigm in which agents become active participants in the observe–plan–execute loop, acting over the same provider state as human operators [22, 27, 30, 36, 40, 43].

Figure 1 depicts the landscape of modern cloud management stack. Within a tenant, responsibility is commonly divided across resource domains (e.g., compute, network, security, and monitoring) and across operational roles (e.g., deployment, diagnosis, and repair). Each party operates from a scoped management context, while its actions ultimately affect infrastructure shared with other parties. As agents join these workflows, cloud management increasingly consists of multiple human operators and agent sessions independently observing, planning, and acting over the shared provider state.

State of the Art & Limitations. Existing cloud management lacks a coordination system—management operations are directly submitted to the provider and rely on the cloud backend to reject invalid ones. In other words, it treats provider errors as a coordination mechanism, but provider errors are late feedback [26, 37]. By the time an operation fails, it has already been interpreted by complex provider internals, and the caller must infer whether the cause was stale context, a hidden dependency, exhausted quota, an in-flight provider conflict, etc. The ambiguity is painful for humans and toxic for agents whose next action is conditioned on the feedback.

The opposite extreme is to impose a single global lock on all management operations. However, most operations do not conflict, and existing IaC systems are explicitly designed to

exploit concurrency among independent changes [33]. Terraform [10], for example, constructs a dependency graph and executes non-dependent operations in parallel. Serialize-all is therefore not an appropriate abstraction either.

While IaC exposes parallelism, it is confined to the state within a workflow; concurrent changes issued by other workflows remain outside its view. Even within this boundary, IaC does not fully capture execution conflicts [11, 12]. IaC provider plugins may rely on coarse, resource-specific mutexes to serialize operations; such locks approximate conflicts, without fine-grained coordination. IaC drift-reconciliation systems [2, 6, 15] like NSync [42] propagate out-of-band changes back into the IaC program, but only after its view has diverged from the cloud. The limitation is therefore not drift itself, but a workflow-scoped execution model that is incomplete within its boundary and blind beyond it.

Modern cloud orchestration platforms often partition infrastructure into independently managed units and workflows, with HCP workspaces [13] and Spacelift stacks [14] as two common examples. These boundaries are useful for organizing state, ownership, and execution, but they do not determine whether concurrently issued changes can safely coexist over shared provider state. Conflicts may cross partitions through shared dependencies or capacity, while distinct intents within the same partition may still overlap during execution. This mismatch becomes especially problematic for agentic management, where sessions are often created around short-lived intents rather than durable administrative boundaries.

3 Towards a Systems Foundation

3.1 A Motivating Example

Figure 2 presents a scenario involving two agents operating on the same VM. Agent A intends to permanently destroy the VM, while Agent B concurrently intends to replace it through a destroy–then–recreate workflow. If Agent B's destroy executes first, Agent A's subsequent destroy fails because the VM no longer exists. Agent B then recreates the VM, while Agent A continues retrying until it destroys the replacement, causing Agent B's intent to be lost. Conversely, if Agent A's destroy executes first, Agent B's destroy fails. Agent B then observes that the VM has already been removed and proceeds to recreate it, causing Agent A's intent to be lost.

This case exposes two distinct limitations. First, the final outcome depends on operation timing and retry order. Without concurrency control over active intents, neither agent nor the system determines which intent should prevail; that decision is effectively delegated to incidental API interleavings and independent retry loops. This makes execution nondeterministic from the agents' perspective. Second, the feedback available to agents lacks intent attribution. The first rejection reveals only an operational conflict, not that another session

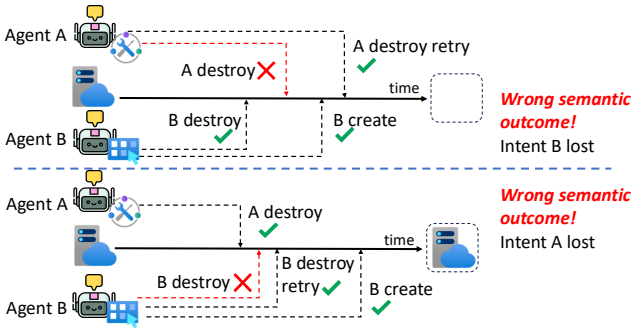

Figure 2: A motivating intent conflict example.

is pursuing an incompatible intent. If agents had learned the conflicting intent with the first rejection, they could have revised their plans instead of blindly retrying. What is missing is a system-level abstraction that connects session-scoped management intent to authoritative shared state throughout planning, execution, and feedback.

3.2 Design Principles

We view coordination as a common foundation for agentic cloud management. By grounding each intent in session-scoped context and coordinating concurrent operations over shared infrastructure, this foundation provides reliable execution semantics upon which new agent-native interfaces, validation and recovery mechanisms, agent architectures, and other management capabilities can be built. We distill this foundation into two design principles as shown below.

Session-scoped cloud views. Cloud-management agents operate on session-scoped intents, but realize those intents over resources and constraints shared across sessions. Therefore, a session boundary must be defined as a policy-controlled projection over reconciled global state, rather than as a private physical copy. This projection may mix private resources and explicitly shared resources, while the underlying global infrastructure should remain hidden. If shared global state is exposed directly, the session abstraction breaks: agents may observe unrelated resources, cleanup artifacts, failures, or transient states; observations become unstable under concurrency; and the system cannot enforce session-level ownership, visibility, or access permissions. Raw global state also lacks the session-level topology, dependencies and recent operational context needed for planning and verification. Leakage appears during execution: one session may fail because another consumes shared capacity, triggers rate limits, or mutates shared parents. Such effects may look intent-relevant to the agent but are actually uncontrolled cross-session interference.

Principle 1. Agentic cloud management must maintain session-scoped local views grounded in global state, with bounded observation and action space.

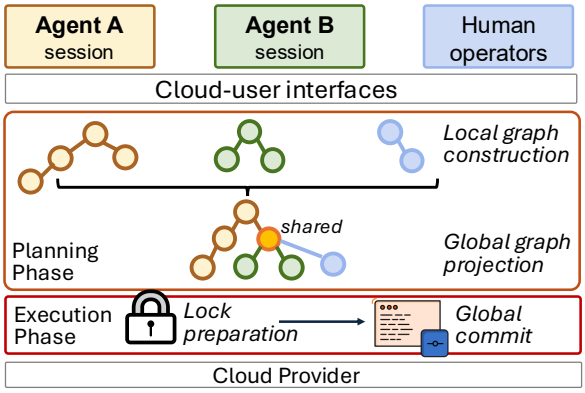

Figure 3: CLOUDWEAVER overview.

Coordination under semantic constraints. Cloud coordination must reason about the semantic scope and effects of operations rather than resource identity alone. Many conflicts arise from identifier, lifecycle, and configuration semantics, such as scoped uniqueness, in-flight resource mutations, and compatibility constraints across related resources. For example, if one agent is updating a VM, another agent should not concurrently update the same VM, resize its attached disks, or attach or detach NICs, because these operations interact through the VM's lifecycle and attachment state. Other conflicts are capacity-related, including user-defined limits such as shared-disk attachment limits and subnet usable-IP capacity, as well as provider-defined limits such as regional VM-family vCPU quota or NAT/public-IP outbound-port quota. Capacity constraints are also semantic rather than simple counters. For example, a shared data disk with `maxShares=2` exposes two attachment slots, but this capacity is scoped to that particular disk and is consumed only by operations that attach the same disk. Coordination must therefore identify semantic conflicts before provider execution, preventing delayed failures, leaked resources, and cross-session interference while preserving parallelism among compatible operations.

Principle 2. Agentic cloud management must enforce semantic constraints before provider execution, preventing unsafe side effects while preserving safe concurrency.

4 CLOUDWEAVER

We are developing CLOUDWEAVER, a system substrate that offers safe and efficient agentic cloud management. Figure 3 gives an overview of CLOUDWEAVER. CLOUDWEAVER separates planning from execution to realize the two principles discussed in §3.2. For Principle 1, CLOUDWEAVER's planning phase maintains local and global views of shared cloud state—local views expose each agent to an isolated and session-scoped environment, while the global view captures underlying shared resources and cross-session dependencies.

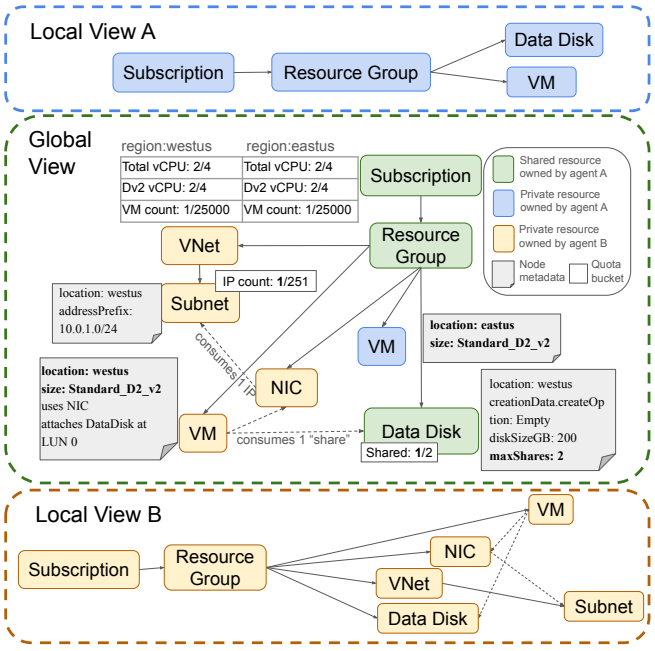

Figure 4: Global and local views of shared resources.

For Principle 2, CLOUDWEAVER's execution phase coordinates operations before provider submission using structural and capacity-aware synchronization. In this section, we present the two key components of CLOUDWEAVER.

4.1 Session Management for Agent Planning

To support the planning phase of AI agents, CLOUDWEAVER maintains shared cloud state in two representations: a global graph and local graphs. The global graph $G^{\mathrm{g}} = (V^{\mathrm{g}}, E^{\mathrm{g}})$ records the reconciled shared state across sessions. Each vertex in V^{g} represents a cloud resource such as a VM, a NIC, a subnet, or a disk, while each edge in E^{g} captures semantic relations such as containment, attachment, routing, reference, or dependency. Vertices are annotated with provider identifiers, lifecycle states, ownership and visibility labels, and intent history. Intent history helps agents identify why and how the resource transformed to the current state as well as who initiated the operations. In general, the global graph serves as the coordination substrate that records what physically exists, what is shared or reserved, and what constraints must be considered before provider execution.

For each session i, CLOUDWEAVER exposes a local graph G_i^{l}, a policy-controlled projection of G^{g}. The local view contains private and explicitly shared resources visible to the agent. Local resources and relations are mapped from their global counterparts by CLOUDWEAVER, allowing the same shared global resource to appear in multiple sessions without revealing the underlying physical sharing. Figure 4 illustrates this projection. The global graph records both topology and

Table 1: MGL modes for cloud transactions.

Mode	Meaning
X	Exclusive access to a node and its descendants.
S	Shared access to a node and its descendants.
IX	Intention to acquire exclusive locks below this node.
IS	Intention to acquire shared locks below this node.
NL	No lock.

constraints: a VM in *westus* consumes regional vCPU quota, its NIC consumes subnet address capacity, and a shared disk consumes one *maxShares* slot. The projection policy exposes shared resources in the relevant local views while keeping private resources visible only to their owner sessions.

At runtime, the local view is constructed and maintained through two separate paths. First, the API parser processes each incoming request and matches each to a provider plugin rule. It evaluates the rule against the current G_i^{l}, and performs a preflight condition check on whether the request is valid under the session's visible resources, dependencies, guards, and capacity constraints. If valid, the parser derives a local graph delta ΔG_i^{l}, including resource updates, semantic edge modifications, extracted metadata, and constraint effects. The projection layer then resolves local resource identifiers from the delta graph to produce a global graph delta ΔG^{g}, which exposes the affected resources, relations, and constraints to the execution phase for coordinated provider execution. After execution completes, the cloud provider responses and reconciliation results are fed back to update both G^{g} and each affected G_i^{l}. This feedback path captures asynchronous completion, failures, and drift to ensure that later admission checks are performed against a safe and up-to-date local view rather than a stale session snapshot.

4.2 Semantic Transaction for Agent Execution

A CLOUDWEAVER transaction is an internal execution unit derived from an intercepted cloud operation. It binds the issuing session and its management intent to the projected global delta of the operation ΔG^{g}, the structural lock requirements, the affected capacity constraints, and the execution status. The transaction allows CLOUDWEAVER to determine whether the operation can be safely admitted before forwarding the corresponding request to the provider. To coordinate these transactions, CLOUDWEAVER uses Multi-Granularity Locks (MGL) to coordinate operations across hierarchical resource dependencies, and Escrow Locks to reserve capacity without serializing independent transactions.

MGL protects the resource topology encoded in ΔG^{g}, including containment, attachment, and dependency relations. For each transaction T, the runtime derives a structural lock set L_T over the touched nodes and edges: modified resources receive exclusive locks, while their ancestors receive intention

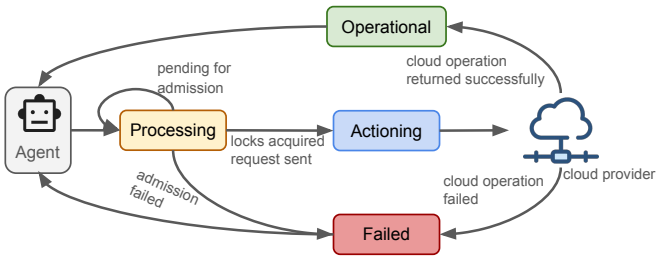

Figure 5: Transaction state transitions.

locks. For example, creating a VM may acquire IX locks on its parent region, VNet, and subnet, and X locks on the new VM and NIC; deleting the subnet would require an X lock on the subnet and therefore conflict with this child-resource creation. Table 1 summarizes the MGL modes.

Escrow locks protect capacity constraints such as regional quota, subnet IP capacity, and shared-disk attachment slots. Each capacity bucket tracks reserved capacity bounds, and an operation can be admitted only if its effect keeps the bucket feasible. This allows concurrent operations to consume disjoint capacity requests while preventing overbooking.

Execution proceeds in two steps to avoid holding expensive locks for a long duration. Figure 5 shows the lifecycle of a transaction. In the planning phase, an intercepted request is parsed into a transaction descriptor containing a lock plan derived from a snapshot of the global graph with status *Processing*. This computation runs outside the global critical section, allowing independent sessions to plan concurrently. In the commit phase, the runtime revalidates the descriptor against the current global graph, acquires the required MGLs, and reserves escrow capacity. If admission fails, the request is not sent to the provider and is rejected, queued, or retried according to policy. Alongside the decision, CLOUDWEAVER returns an attributable conflict report identifying the conflicting resources and constraints together with the intent of the transaction currently blocking admission. If admission succeeds, the admitted delta is installed in G^{g} with status *Actioning*, local–global mappings are recorded, and the API request is forwarded. Upon completion, the runtime reconciles the provider response with the graph: a successful operation transitions the affected resources to *Operational*, whereas an unsuccessful operation marks them as *Failed* and releases or rolls back the associated reservations.

5 Preliminary Validation

We validate CLOUDWEAVER using an Azure trace, showing that CLOUDWEAVER preserves concurrency among independent operations while preventing semantic conflicts.

Experimental workload trace. The workload contains six Azure API operations from three sessions. Session A creates a shared resource group and VNet, while Session C creates

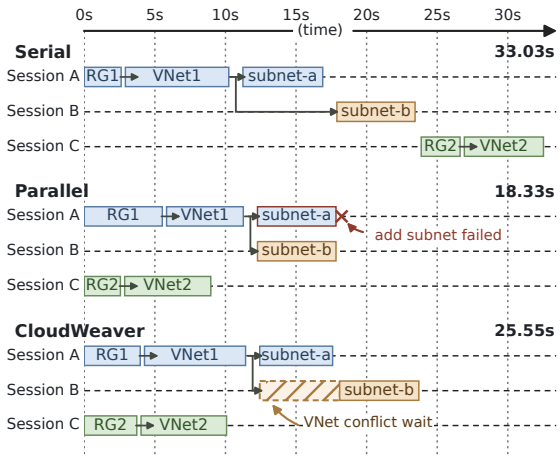

Figure 6: Timeline of the preliminary validation trace.

an independent resource group and VNet. This independent branch represents safe parallelism that should not be serialized with A's work. After creating the shared VNet, Sessions A and B add two distinct subnets to it. The API sequence treats these requests as independent because they target different subnet resources, allowing them to become ready at the same time. However, both operations mutate the same Azure VNet parent and therefore cannot safely execute concurrently under Azure's provider semantics.

Execution behavior. We compare CLOUDWEAVER against two execution policies. The *serial* policy executes one operation at a time across sessions. This avoids provider conflicts, but also serializes independent work such as Session C's VNet creation. The *parallel* policy executes all dependency-ready operations concurrently, overlapping independent branches while issuing both subnet writes to the shared VNet at the same time. In our run, Azure accepts one request and rejects the other with `HTTP 409` and `AnotherOpInProgress`, because an operation on the same VNet parent is still in progress. This failure shows that the *parallel* policy understands explicit request ordering, but not provider-level conflict scopes, causing it to issue two distinct subnet requests concurrently even though Azure serializes both at their shared VNet parent. In contrast, CLOUDWEAVER supports semantic coordination, and admits API sequences through the transaction layer before forwarding them to Azure.

Latency comparison. Figure 6 shows that serial execution completes all six operations in 33.03s, while unconstrained parallel execution finishes in 18.33s but succeeds on only five operations. The interface completes all operations in 25.55s, reducing makespan by 7.48s (22.6%) while preserving correctness. It overlaps Session C's independent operations with Session A's shared-resource setup, while serializing conflicting subnet writes at the VNet scope. This preserves safe parallelism and avoids provider-side failures.

6 Discussion and Future Work

Scalability and safety. CLOUDWEAVER isolates provider-specific semantics in plugins, allowing each provider to define rules without modifying the coordination protocol. Developing these plugins may be largely automated by deriving cloud-operation semantics from API specifications, documentation, and traces [4, 16, 19, 31], which we leave as future work. The deployment overhead of CLOUDWEAVER is modest because the system operates only on compact resource metadata, dependency edges, lock states, and escrow counters. Its compute cost therefore depends mainly on the scope of each operation rather than the total cloud size. To ensure the safety of our semantic transaction protocol, we further validated it with TLA+, using TLC for bounded safety and progress checking and TLAPS for a general deadlock-freedom proof.

Intent-aware agentic interface. An agent-native interface built on CLOUDWEAVER should be organized around management intents rather than individual provider API calls or global updates. It would let an agent declare an objective, inspect the scope of resources and actions available to its session, submit a multi-step plan for validation, and receive structured outcomes that distinguish admissible actions, deferred actions, and conflicts requiring replanning. Rather than exposing provider-specific failures as unstructured text, the interface should return machine-consumable explanations and support explicit follow-up actions such as revise, wait, abort, or verify. Such an interface would make closed-loop cloud management a first-class interaction model, while CLOUDWEAVER supplies the underlying state and coordination needed to implement it.

Implications for RL training. CLOUDWEAVER enables scalable RL training on real cloud infrastructure by allowing many agent sessions to share warm resources while each observes a clean, session-scoped environment. Before each training batch, the system can determine which sessions can reuse existing accounts, networks, images, and pre-created resources based on task requirements, provisioning latency, quota pressure, and isolation constraints. View isolation prevents cross-session interference from corrupting trajectories and rewards, while semantic coordination preserves safe parallelism. Together, they reduce setup cost and latency, improve rollout throughput, and retain real-provider fidelity.

7 Concluding Remarks

Agentic cloud management requires more than capable AI agents: it requires a systems foundation that keeps their reasoning grounded in the cloud infrastructure that they jointly change. This paper discusses design principles for such a coordination substrate, and presents CLOUDWEAVER as an initial realization. Looking forward, we envision this foundation supporting a broader evolution of agentic cloud management,

from today's copilots toward persistent, collaborative, and increasingly autonomous systems for planning, validation, recovery, and operation.

References

[1] Amazon Web Services — aboutamazon.com. https://www.aboutamazon.com/what-we-do/amazon-web-services. [Accessed 29-06-2025].

[2] Detecting and Managing Drift with Terraform. https://www.hashicorp.com/blog/detecting-and-managing-drift-with-terraform.

[3] Gemini for Google Cloud. https://cloud.google.com/products/gemini.

[4] LocalStack for AWS — localstack.cloud. https://www.localstack.cloud/localstack-for-aws. [Accessed 29-06-2025].

[5] Microsoft copilot in Azure. https://azure.microsoft.com/en-us/products/copilot.

[6] Patterns for Drift Detection with Pulumi. https://www.pulumi.com/blog/patterns-drift-detection/.

[7] [Pulumi] How Pulumi works. https://www.pulumi.com/docs/concepts/how-pulumi-works/.

[8] Pulumi: Infrastructure as code in any programming language. https://www.pulumi.com/.

[9] [Pulumi] Resource Providers. https://www.pulumi.com/docs/concepts/resources/providers/.

[10] Terraform by Hashicorp. https://www.terraform.io/.

[11] Terraform Resource Life Cycle. https://developer.hashicorp.com/terraform/tutorials/state/resource-lifecycle.

[12] Terraform Rollback. https://developers.cloudflare.com/terraform/tutorial/revert-configuration/.

[13] Terraform workspaces. https://developer.hashicorp.com/terraform/language/state/workspaces.

[14] The IaC Orchestration Platform Engineers Trust. https://spacelift.io/.

[15] Tools for Infrastructure Drift Detection. https://snyk.io/blog/tools-infrastructure-drift-detection/.

[16] Use Azurite emulator for local Azure Storage development — learn.microsoft.com. https://learn.microsoft.com/en-us/azure/storage/common/storage-use-azurite?tabs=visual-studio%2Cblob-storage. [Accessed 02-07-2025].

[17] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, and D. A. Tamburri. Devops: introducing infrastructure-as-code. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 497–498. IEEE, 2017.

[18] V. Atlidakis, P. Godefroid, and M. Polishchuk. Restler: Stateful rest api fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 748–758. IEEE, 2019.

[19] A. Bhatnagar, Y. Qiu, S. McClure, S. Ratnasamy, and A. Chen. A case for learned cloud emulators. In *Proceedings of the 24th ACM Workshop on Hot Topics in Networks*, pages 69–76, 2025.

[20] E. Bisong. An overview of google cloud platform services. *Building Machine learning and deep learning models on google cloud platform: a comprehensive guide for beginners*, pages 7–10, 2019.

[21] H. Chen, X. Qiu, K. Ren, and X. Cui. An aiops approach to data cloud based on large language models. In *Proceedings of the 2024 4th International Conference on Artificial Intelligence, Big Data and Algorithms*, pages 634–641, 2024.

[22] Y. Chen, J. Pan, J. Clark, Y. Su, N. Zheutlin, B. Bhavya, R. Arora, Y. Deng, S. Jha, and T. Xu. Stratus: A Multi-agent System for Autonomous Reliability Engineering of Modern Clouds. In *Proceedings of the 39th Annual Conference on Neural Information Processing Systems (NeurIPS'25)*, Dec. 2025.

[23] Y. Chen, M. Shetty, G. Somashekar, M. Ma, Y. Simmhan, J. Mace, C. Bansal, R. Wang, and S. Rajmohan. Aiopslab: A holistic framework to evaluate ai agents for enabling autonomous clouds, 2025.

[24] J. Clark, Y. Su, S. M. R. Pial, Y. Tian, L. Gniedziejko, H.-A. Jacobsen, Y. Chen, and T. Xu. SREGym: A Live Benchmark for AI SRE Agents with High-Fidelity Failure Scenarios. *arXiv:2605.07161*, 2026.

[25] C. Coleman, W. G. Griswold, and N. Mitchell. Do cloud developers prefer clis or web consoles? clis mostly, though it varies by task. *arXiv preprint arXiv:2209.07365*, 2022.

[26] J. T. Gu, X. Sun, W. Zhang, Y. Jiang, C. Wang, M. Vaziri, O. Legunsen, and T. Xu. Acto: Automatic end-to-end testing for operation correctness of cloud system management. In *Proceedings of the 29th ACM Symposium on Operating Systems Principles*, pages 96–112, 2023.

[27] H. Guo, J. Yang, J. Liu, L. Yang, L. Chai, J. Bai, J. Peng, X. Hu, C. Chen, D. Zhang, et al. Owl: A large language model for it operations. In *International Conference on Learning Representations*, volume 2024, pages 23310–23333, 2024.

[28] S. Jha, R. Arora, Y. Watanabe, T. Yanagawa, Y. Chen, J. Clark, B. Bhavya, et al. ITBench: Evaluating AI Agents across Diverse Real-World IT Automation Tasks. In *Proceedings of the 42th International Conference on Machine Learning (ICML'25)*, July 2025.

[29] P. T. J. Kon, J. Liu, Y. Qiu, W. Fan, T. He, L. Lin, H. Zhang, O. M. Park, G. S. Elengikal, Y. Kang, A. Chen, M. Chowdhury, M. Lee, and X. Wang. Iac-eval: A code generation benchmark for cloud infrastructure-as-code programs. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2024.

[30] J. Liu, C. Zhang, J. Qian, M. Ma, S. Qin, C. Bansal, Q. Lin, S. Rajmohan, and D. Zhang. Large language models can deliver accurate and interpretable time series anomaly detection. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 2*, pages 4623–4634, 2025.

[31] Moto Contributors. Moto: Mock AWS Services — Moto 5.1.7.dev documentation — docs.getmoto.org. https://docs.getmoto.org/en/latest/, 2025. [Accessed 02-07-2025].

[32] J. Peng, Y. Qiu, P. T. J. Kon, P. Zhao, Y. Huang, Z. Guo, X. Wang, and A. Chen. Automated lifting for cloud infrastructure-as-code programs. In *2025 IEEE/ACM International Workshop on Cloud Intelligence & AIOps (AIOps)*, pages 4–9. IEEE, 2025.

[33] Y. Qiu, P. T. J. Kon, R. Beckett, and A. Chen. Unearthing semantic checks for cloud infrastructure-as-code programs. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 574–589, 2024.

[34] Y. Qiu, P. T. J. Kon, J. Xing, Y. Huang, H. Liu, X. Wang, P. Huang, M. Chowdhury, and A. Chen. Simplifying cloud management with cloudless computing. In *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*, pages 95–101, 2023.

[35] J. Shah and D. Dubaria. Building modern clouds: using docker, kubernetes & google cloud platform. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0184–0189. IEEE, 2019.

[36] M. Shetty, Y. Chen, G. Somashekar, M. Ma, Y. Simmhan, X. Zhang, J. Mace, D. Vandevoorde, P. Las-Casas, S. M. Gupta, S. Nath, C. Bansal, and S. Rajmohan. Building ai agents for autonomous clouds: Challenges and design principles. In *Proceedings of 15th ACM Symposium on Cloud Computing*, 2024.

[37] X. Sun, W. Ma, J. T. Gu, Z. Ma, T. Chajed, J. Howell, A. Lattuada, O. Padon, L. Suresh, A. Szekeres, et al. Anvil: Verifying liveness of cluster management controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 649–666, 2024.

[38] Y. Xu, Y. Chen, X. Zhang, X. Lin, P. Hu, Y. Ma, S. Lu, W. Du, Z. Mao, E. Zhai, and D. Cai. Cloudeval-yaml: A practical benchmark for cloud configuration generation, 2023.

[39] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024.

[40] Y. Yang, Y. Deng, Y. Xiong, B. Li, H. Xu, and P. Cheng. Tsguard: Automated user-centric incident diagnosis for ai workloads in the cloud. *Proceedings of the ACM on Software Engineering*, 3(FSE):253–276, 2026.

[41] Z. Yang, A. Bhatnagar, Y. Qiu, T. Miao, P. T. J. Kon, Y. Xiao, Y. Huang, M. Casado, and A. Chen. Cloud infrastructure management in the age of ai agents. *arXiv preprint arXiv:2506.12270*, 2025.

[42] Z. Yang, H. Guan, V. Nicolet, B. Paulsen, J. Dodds, D. Kroening, and A. Chen. Automated cloud infrastructure-as-code reconciliation with ai agents. *arXiv preprint arXiv:2510.20211*, 2025.

[43] L. Zhang, T. Jia, M. Jia, Y. Wu, A. Liu, Y. Yang, Z. Wu, X. Hu, P. Yu, and Y. Li. A survey of aiops in the era of large language models. *ACM Computing Surveys*, 2025.