A contract runtime for cross-system infrastructure operations

Separating intent, policy, execution and evidence across heterogeneous environments

Jeleel Muibi · HybridOps.Tech · Version 1.2 · September 2026

Abstract

Infrastructure operations increasingly cross several control planes. One outcome may require infrastructure state, configuration automation, network authority, workload readiness, published outputs for dependent systems and a later recovery or closure path. Each participating system can be internally correct while the wider operation remains incomplete.

This paper introduces a contract runtime for that cross-system boundary. The runtime separates capability intent from environment policy and implementation, composes dependencies explicitly, evaluates readiness before dependent work advances, publishes declared outputs rather than incidental process text, and retains a structured record of the resolved operation and its verification. Bounded lifecycle actions such as rebuild, recovery and resource closure use the same contract model rather than being appended as unrelated cleanup workflows.

The runtime is intended for operations that leave a shared reconciliation plane and cross systems without one common controller, authority history or completion record. External review of v1.1 sharpened this applicability boundary.

HybridOps is the open-source implementation of the model. Its contracts are exercised across authoritative infrastructure, machine-image publication, recovery and network-lab lifecycles where the underlying providers and workload systems change but the higher-level operating contract remains stable.

The operational question is:

When one infrastructure outcome crosses several control planes, what should decide whether the overall operation may advance, publish or close?

1. Cross-system completion is its own runtime result

A cloud or virtualisation API can report that a resource exists. A configuration system can report that a play completed. An intended-state system can contain the approved addressing. A workload API can report health. The wider outcome can span all of them while requiring a separate operation-level completion decision.

This creates a recurring class of partial success:

- capacity exists but a required upstream authority is unavailable;
- configuration completes before the workload reaches its required usable state;
- one stage exits successfully before the state needed by a dependent stage is published;
- a provider object was created under the wrong environment policy;
- a recovery transition completes while service authority is still ambiguous;
- a destroy command succeeds for one resource while cost-bearing dependencies remain active.

The gap is a runtime boundary for the operation that spans those systems.

Established control models and the remaining gap

Declarative infrastructure tools such as Terraform and OpenTofu manage desired resource graphs and provider-backed state [1,2]. Configuration-management systems such as Ansible execute ordered configuration across managed systems [3]. Kubernetes controllers and operators reconcile declared resources toward desired state [4]. GitOps formalises declarative, versioned and reconciled operation around a source of desired state [5].

Where one controller owns the complete lifecycle, that controller is the natural lifecycle boundary. A practitioner review of v1.1 made this explicit using a Kubernetes-native estate, where dependency order, ready conditions, desired state and lifecycle history can all live inside one reconciliation plane [12].

The contract runtime applies when the operation leaves that shared plane and crosses legacy systems, SaaS APIs, provider workflows, one-shot automation, recovery actions or other bounded transitions outside one common controller and history.

The innovation introduced here is a versioned runtime contract for that wider operation. It treats capability intent, environment policy, implementation binding, dependency state, verification and lifecycle evidence as separate but composable concerns.

2. Six contracts define the runtime boundary

The model uses six primary contracts.

Contract	Responsibility
ModuleSpec	Declares the capability, inputs, required outputs and checks
Profile	Carries environment policy and defaults
Driver	Adapts the runtime contract to an execution class
Pack	Carries versioned implementation assets
Blueprint	Composes modules into a dependency-aware lifecycle
Run record	Captures the resolved operation, execution result and verification

The separation matters because each concern changes at a different rate. A capability keeps a stable operator contract while the target environment or implementation changes. A pack can evolve behind a compatible capability contract. A profile can change policy without redefining the capability. A blueprint can compose the same module into a different lifecycle.

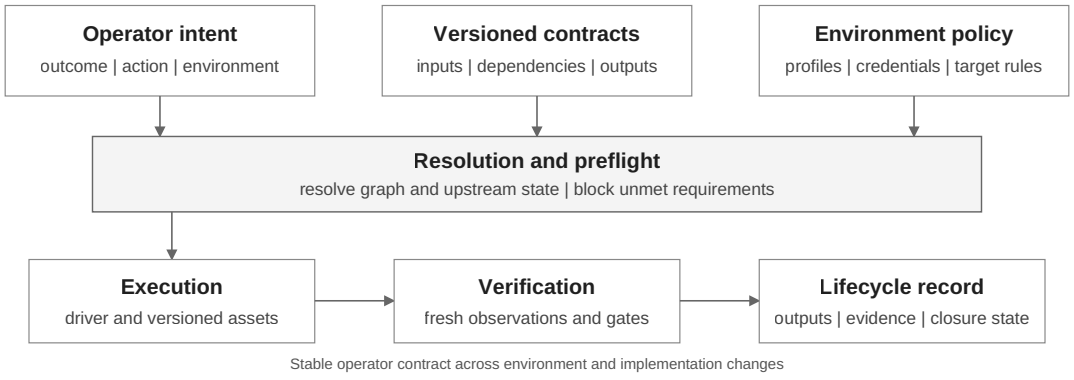

Figure 1. HybridOps separates intent, policy, implementation and verification.

3. Keep intent stable while implementation changes

A module contract describes the capability outcome the runtime is expected to operate, while the selected implementation carries the target-specific command sequence.

```
capability intent
      +
environment policy
      |
      v
resolved runtime contract
      |
      +--> driver / implementation A
      |
      +--> driver / implementation B
      |
      v
published result + verification + run record
```

The provider or workload system remains visible and authoritative for the resources it owns. The runtime preserves implementation differences while requiring the same declared capability and output contract where substitution is permitted.

4. Dependency state must be explicit

A process exit is an execution signal. A dependent stage consumes the specific outputs and capability state it requires and bases advancement on those declared handoffs.

Blueprints therefore make dependency order explicit, reject invalid cycles and allow a dependent step to fail at the contract boundary when required upstream state is absent or invalid.

The same dependency model constrains deployment, rebuild, recovery and closure, including reverse-order lifecycle actions.

5. Separate admission checks, runtime readiness and completion evidence

External review of v1.1 sharpened the gate model into three different jobs.

Admission or structural validation rejects errors that are cheap, deterministic and independent of live execution. If a contract cannot compile, its types are invalid or its dependency graph is structurally impossible, the operation fails before persistence or provider contact [12].

Runtime readiness covers facts that require the selected environment: credentials, provider functions, upstream state, live network paths or another stage's outputs.

Completion evidence asks whether the executed outcome matches what was declared strongly enough to advance or close.

A Metal3/Ironic review exposed an important ownership case: a native controller can validly complete image provisioning and reboot while the wider requested role still depends on readiness owned elsewhere. Native controller completion remains authoritative for the lifecycle it owns; it becomes an input rather than proof for a broader outcome whose remaining condition belongs to another system [21].

Separating these gates catches deterministic errors early while reserving live correctness and completion evidence for runtime observation.

6. Completion evidence must be authoritative and fresh

Completion evidence comes from the resulting authoritative state. Events and timestamps can supplement that evidence, while causal freshness is established through revisions, generations, ETags or equivalent tokens. A reconciler may also reach the desired state without producing a new write event.

The stronger completion rule from external review is:

1. read the resulting state from the authoritative server or control plane;
2. compare the required fields with the declared outcome; and
3. where the system exposes a meaningful freshness token, prove that the read is at or beyond a known operation floor using a resource version, observed generation, ETag or equivalent causal revision [12].

When a system exposes final state without a meaningful revision or generation, the record preserves the observed point-in-time state and classifies causal freshness as unverifiable for freshness.

The runtime evidence model therefore uses the strongest causal freshness available from each underlying system.

7. Point-in-time verification and interval stability are distinct evidence classes

A second v1.1 review challenge exposed a harder case: authority can change between gates. A human, agent, autoscaler or security tool can mutate a resource while the operation is in flight, and the final state may later reconverge.

HybridOps uses operation-level gates and can re-evaluate live state in selected paths. A successful final check establishes final point-in-time state; interval stability requires revision, event-history, ETag or equivalent evidence.

Core issue #286 tracks that stronger evidence contract with outcomes such as stable, drifted and unverifiable where the underlying system exposes a basis for identifying intervening mutation [13]. Event history may help detect interval drift, but final completion still requires a fresh authoritative read.

The lifecycle record therefore distinguishes point-in-time evidence from interval-stability evidence explicitly.

8. Publish declared state as the handoff contract

Downstream operations consume declared outputs through the state contract rather than incidental stdout or copied provider identifiers from earlier runs.

Modules publish declared outputs under an environment-scoped state contract. A dependent operation resolves the named state it requires and rejects the handoff if that state is missing or invalid.

The purpose is an operation-scoped state handoff that makes cross-stage consumption explicit inside one runtime-managed lifecycle.

9. Treat recovery and closure as first-class lifecycle operations

Infrastructure completion includes creation, replacement, recovery and closure, with each transition carrying its own operational conditions.

The contract model applies to:

- rebuild and replacement decisions;
- authority-changing recovery transitions;
- state preservation before destructive actions;

- reverse dependency order;
- terminal resource-state checks; and
- evidence of what remained intentionally active.

This is particularly important for temporary or recovery environments where closure determines residual authority and continuing cost-bearing state.

10. Structured evidence belongs to the runtime

A run record is written as part of the operation rather than assembled later from terminal history.

Representative records tie together:

- resolved inputs;
- environment policy;
- implementation binding;
- process metadata and redacted output;
- published state;
- verification results; and
- stable run identity.

Provider state, intended-state authority and workload telemetry retain their native roles. The run record captures how the cross-system operation resolved and used those systems together with the evidence level established by the runtime.

11. Implemented evaluation paths

The implementation exercises the runtime through several materially different paths.

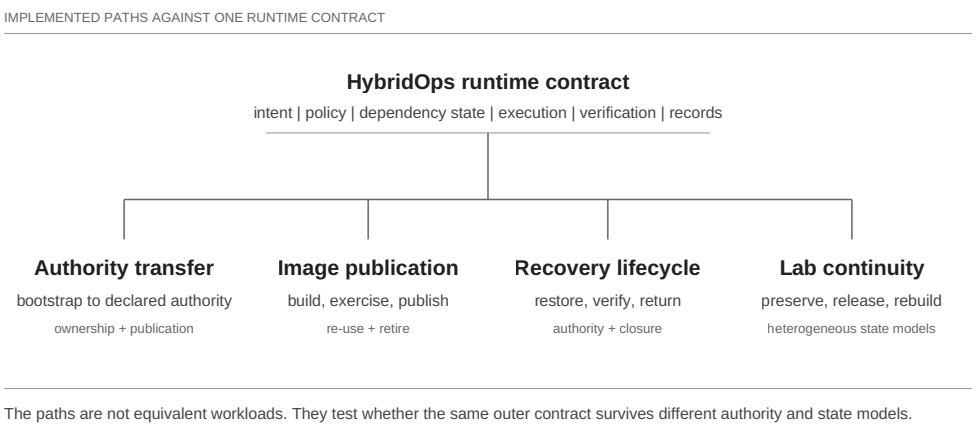

Figure 2. Implemented paths exercise different boundaries of the same runtime contract.

11.1 Authority transfer

The authoritative-foundation path tests a bounded bootstrap followed by transfer to intended-state authority. Dependent infrastructure is blocked when the required authority is absent or not ready.

11.2 Machine-image publication

The image lifecycle tests the handoff from a completed image build to a reusable logical dependency. A controlled evaluation completed 20 fresh construction, exercise, publication, reuse and retirement observations across four operating- system families, plus three state-resolved downstream consumer handoffs.

11.3 Recovery operating model

The PostgreSQL recovery paths test authority, readiness, workload verification, service-location changes, failback and residual-state closure across self-managed and managed recovery mechanisms.

11.4 Network-lab continuity

The network-lab path tests whether high-resource execution capacity can be separated from the continuity requirement of the lab. Existing EVE-NG, GNS3 and Containerlab environments can enter through a same-platform migration path that assesses the requested capture streams, verifies checksums and target capacity, binds the import to the selected blueprint and restores through the normal lifecycle [20].

For EVE-NG, the retained set can include lab definitions, saved device configuration where available, guest-quiesced writable QEMU disks and separately verified base images referenced by the selected labs; licence material remains outside the bundle [15,20]. GNS3 retains project and controller state including writable node disks [16]. The validated GCP Containerlab path adds direct fresh-host evidence: off-host recovery verification gated deletion of the original VM, the lab was reconstructed on fresh compute with a different resource identity, final health passed, and the declared compute was removed [14].

These materially different workloads stress different parts of the same runtime model while provider and workload semantics remain native to the systems that own them.

12. What makes the model falsifiable

Falsifiability requires visible failure boundaries.

The model should be challenged by:

- conflicting operations against the same environment;
- missing or stale upstream state;
- authority changes during execution;
- implementation substitution that breaks a declared output contract;
- process success followed by failed workload verification;
- stale completion evidence or replayed events;
- recovery or rebuild with incomplete reverse state;
- failed preservation before a destructive transition; and
- closure that leaves undeclared residual resources.

Concurrency and in-flight authority drift form the next evidence layer. Locking, conflict detection, evidence freshness and interval observability remain explicit where several operations or actors can address the same environment.

13. HybridOps implementation boundary

HybridOps is the open-source implementation of the contract runtime. It owns its ModuleSpec, Profile, Driver, Pack, Blueprint and run-record contracts and the lifecycle semantics that join them.

Terraform/OpenTofu, Ansible, provider APIs, intended-state systems, workload controllers and other native control planes retain authority within their own boundaries. The implemented contribution is the runtime contract that governs the cross-system operation when no single underlying reconciliation plane owns the whole lifecycle.

HybridOps provides operation-scoped gates and records. Stronger proof of in-flight authority stability and causal completion freshness remains outside the implemented contract and is tracked in issue #286.

14. Practitioner review question

For an operation that leaves a shared reconciliation plane, what evidence would you require before a cross-system runtime is allowed to declare the outcome complete?

A reviewer can also identify cases where one existing controller already owns the complete lifecycle, which helps define the runtime's natural applicability boundary.

References

1. HashiCorp, Terraform documentation.
2. OpenTofu, Documentation.
3. Red Hat, Ansible documentation.
4. Kubernetes, Controllers and Operator pattern.
5. OpenGitOps, GitOps principles.
6. HybridOps Core, open-source runtime implementation.
7. HybridOps documentation, Blueprint index.
8. HybridOps documentation, Preflight contract.
9. HybridOps documentation, Evidence and redaction standard.
10. HybridOps, Build, Boot, Verify, Publish controlled evaluation.
11. HybridOps documentation, PostgreSQL HA DR Cycle showcase.
12. HybridOps Core issue #269, external technical review of Contract Runtime v1.1.
13. HybridOps Core issue #286, Detect in-flight authority drift across runtime gates.
14. HybridOps Core, validated GCP Containerlab lifecycle, PR #303.
15. HybridOps Core, EVE-NG lab archive and stopped-QEMU state contract.
16. HybridOps Core, GNS3 lab archive contract.
17. HybridOps documentation, Authoritative On-Prem Foundation showcase.
18. HybridOps documentation, Managed PostgreSQL DR with Cloud SQL showcase.
19. HybridOps documentation, Network Lab Continuity Across Ephemeral Compute reference scenario.
20. HybridOps Core, merged existing-lab migration implementation, PR #366.
21. Metal3 Development List, discussion of native controller completion and wider requested-outcome readiness.