HybridOps: a contract runtime for reproducible infrastructure operations

Governing intent, policy, execution and evidence across heterogeneous environments

Jeleel Muibi
Version 1.1 | Technical review edition
First published June 2026 · revised 8 August 2026

Version note: Version 1.1 broadens the technical review from the governed network-emulation case used in v1.0 to the HybridOps runtime architecture itself. Version 1.0 remains unchanged as the institutional review edition.

Abstract

Hybrid infrastructure operations are often implemented as separate provisioning, configuration, networking, platform, recovery and teardown paths. Each path may be automated, but the wider operation can still depend on workflow-specific policy, ordering, readiness checks, handoffs and evidence. The resulting problem is not a lack of automation. It is the absence of a stable operator contract across the full lifecycle.

HybridOps Core is an open-source contract runtime for that boundary. A ModuleSpec defines intended capability. A Profile carries environment policy. A Driver binds execution. A versioned Pack carries implementation assets. A Blueprint composes modules into a dependency-aware lifecycle. The runtime resolves these contracts, rejects invalid dependency graphs, evaluates preflight conditions, executes the selected implementation, publishes outputs and writes a structured run record.

The architecture is exercised through several implemented paths: an authoritative on-prem foundation, a controlled machine-image lifecycle, PostgreSQL HA recovery, hybrid networking and governed network-emulation environments. The network-emulation case retained from v1.0 provides a useful substitution test because provider and workload implementations change while the higher-level lifecycle remains stable.

This paper defines the runtime model, its technical contribution, its control boundaries, the failure and evidence semantics that make it testable, and the questions that should be challenged by independent practitioners.

Keywords: infrastructure contracts; platform engineering; hybrid operations; lifecycle governance; preflight; operational evidence; reproducibility.

1. The operating problem

Infrastructure teams rarely operate one lifecycle through one control boundary. A platform path may begin with capacity, continue through image preparation and network authority, configure a service, publish state for a downstream dependency, verify health, and later enter recovery or teardown. Each stage can be automated independently while the overall operation remains coupled through pipeline logic, local conventions and operator procedure.

That coupling creates several recurring problems.

First, intent and procedure drift together. A change in target, backend, provider-specific behaviour or implementation version can force changes into the operator workflow even when the intended capability has not changed.

Second, environment policy spreads into implementation. Approval rules, backend selection, connectivity requirements, destructive-action policy and other environment concerns are repeated across scripts and workflow stages.

Third, readiness is evaluated inconsistently. A stage may be syntactically valid while a required credential, upstream state contract, network path or service prerequisite is unavailable. The failure then occurs after execution has already begun.

Fourth, lifecycle evidence is fragmented. Individual control planes retain state for the resources they own, but the wider operation still needs a record of the resolved request, policy context, execution result, published outputs and verification outcome.

HybridOps treats these as runtime concerns. The operator contract describes what capability should be operated. Environment policy, implementation binding, dependency ordering and execution records are resolved around that contract rather than embedded into each operator procedure.

2. Runtime model

The runtime is organised around six primary contracts.

2.1 ModuleSpec

A ModuleSpec declares the intended capability. It defines inputs, requirements, execution binding, published outputs and required checks without encoding the full target-specific command sequence.

The module contract is versioned and reviewable. Input defaults can be merged with environment and operator values under deterministic precedence. Required outputs provide an explicit state contract for dependent modules rather than leaving downstream consumers to infer values from console output.

2.2 Profile

A Profile carries environment policy and defaults. This keeps backend selection, workspace rules, connectivity expectations, execution policy and similar environment concerns outside the module's intended capability.

The separation is deliberate. A capability should not require a new module contract simply because it is operated under a different environment policy.

2.3 Driver

A Driver adapts the runtime contract to an execution class. The driver owns execution preparation, process invocation, result capture and the translation between runtime inputs and the selected implementation surface.

Drivers are registered runtime components. If a required driver is not available, the runtime can reject the selected execution path before an apply operation proceeds.

2.4 Pack

A versioned Pack contains implementation assets. The pack can evolve without redefining the operator-facing capability so long as the contract remains compatible.

This gives implementation changes an explicit version boundary. The operator contract and the implementation artifact are related, but they are not the same object.

2.5 Blueprint

A Blueprint composes modules into a dependency-aware lifecycle. The runtime validates the graph, rejects cycles, resolves ordering and evaluates required preflight conditions before dependent execution advances.

Blueprints also provide the boundary for guarded lifecycle operations such as rebuild and destroy. This is important because a deployment graph is not only a creation graph. Recovery, replacement and closure need the same dependency awareness.

2.6 Run record

Every runtime operation produces a structured, non-secret run record. The record captures resolved inputs, policy context, execution metadata, published outputs, verification results and redacted process output under a stable run identity.

The run record does not replace the authoritative state held by an individual resource or platform control plane. It records the HybridOps operation that crossed those boundaries.

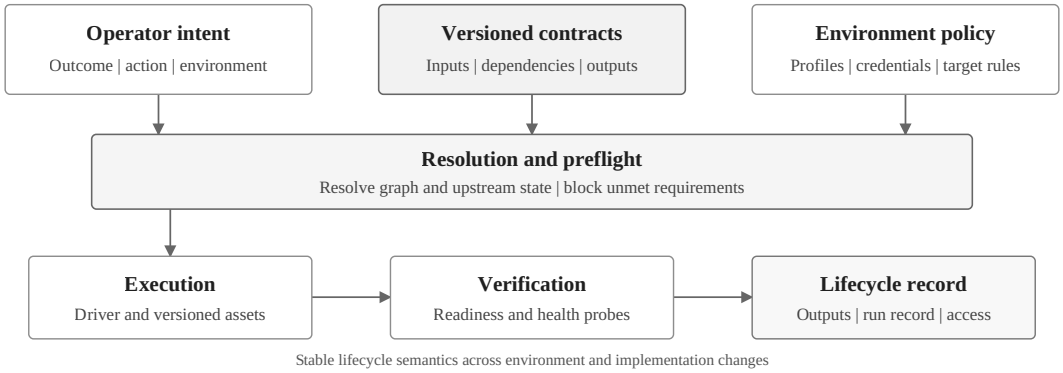

Figure 1. HybridOps separates intent, policy, implementation and verification.

The resulting operator path can be summarised as:

ModuleSpec + Profile → hyops apply → Driver + Pack → Run record

Blueprint ordering constrains the apply path.

The model is intentionally small. ModuleSpec carries intent. Profile carries policy. Driver and Pack bind implementation. Blueprint carries composition. The run record captures the resolved operation and its result.

3. Technical contribution

The HybridOps contribution is the runtime contract formed by these boundaries and the lifecycle semantics implemented around them.

The architecture is expressed through seven testable properties.

1. **Stable intent contracts.** Module and blueprint specifications remain independent of implementation-specific command sequences.
2. **Separation of intent, policy and implementation.** Environment policy and implementation assets can change without redefining the intended capability.

3. **Dependency-aware lifecycle operations.** Blueprints validate graph structure, reject cycles and preserve explicit ordering for deployment, recovery, rebuild and teardown paths.
4. **Preflight and verification as gates.** Required credentials, policy, upstream state and capability checks can block dependent work before or after execution as appropriate.
5. **Published state contracts.** Modules expose declared outputs that can be consumed by downstream operations without parsing incidental process output.
6. **Replaceable implementation.** A Driver or Pack can change behind the operator contract while preserving the expected lifecycle semantics.
7. **Structured lifecycle records.** Each operation produces a stable, redacted record of what was resolved, executed, published and verified.

These are not independent features presented as a checklist. They form one runtime model. The technical claim is that the operator lifecycle can remain stable while environment and implementation details change, and that failures can be located at explicit contract boundaries rather than only at the process that happened to fail.

The contribution is architectural rather than primitive-level. Contracts, dependency graphs, validation, reconciliation, health checks and state are well-established engineering mechanisms. HybridOps gives those mechanisms specific semantics inside a versioned infrastructure runtime and tests whether the combined operating model remains coherent across heterogeneous infrastructure classes.

4. Position relative to existing control boundaries

HybridOps is not defined by a list of third-party products. Its boundary is the infrastructure operation itself: declared capability, environment policy, lifecycle composition, execution binding, published state, verification and run evidence.

Several established control models address adjacent concerns:

Control boundary	Primary concern	Distinction in this paper
Declarative resource control	Desired infrastructure resources, planning and managed resource state	HybridOps governs the wider operation and its lifecycle contracts rather than only the resource graph
Configuration automation	Ordered configuration and deployment across managed systems	HybridOps keeps capability intent and environment policy separate from the implementation procedure
Reconciliation controllers	Continuous convergence of declared platform resources	HybridOps also covers bounded operations such as preflight, recovery, rebuild, publication and teardown
Application runtime services	Portable application-level service, state, workflow or messaging concerns	HybridOps operates at the infrastructure lifecycle boundary rather than the application runtime API boundary
HybridOps	Intent, policy, lifecycle composition, execution binding, verification and run records	One operator contract across heterogeneous infrastructure operations

Representative systems in those adjacent categories include Terraform and OpenTofu, Ansible, Kubernetes controllers and operators, and Dapr. They are important comparison points because their control boundaries are familiar to practitioners. The HybridOps claim is not that those boundaries are deficient. It is that a separate infrastructure lifecycle boundary exists when one operation crosses resource, configuration, platform, recovery and closure concerns.

This distinction is visible in the source model. HybridOps owns the ModuleSpec, Profile, Driver, Pack, Blueprint and run-record contracts. The implementation behind a Driver or Pack remains explicit and inspectable, but it does not define the operator contract.

5. Runtime lifecycle and authority

A contract runtime needs more than a command router. Its value depends on how it behaves before execution, across dependencies, under partial failure and at closure.

5.1 Validation before execution

Specification validation checks the declared contract before the runtime selects an execution path. Blueprint validation also rejects cyclic dependency graphs. This catches structural errors without contacting an infrastructure target.

Preflight is a separate boundary. A specification can be valid while the selected operation is not ready to execute. Preflight therefore evaluates runtime prerequisites such as required credentials, policy, implementation availability and upstream state.

The separation matters because syntax validity and operational readiness answer different questions.

5.2 Dependency state

Blueprint stages publish explicit outputs and capability state for downstream consumers. This avoids a common failure mode in which a pipeline assumes that an earlier process exit implies every required downstream fact now exists.

A dependent stage can instead require the state contract it needs. The runtime can stop at that boundary if the contract is absent or invalid.

5.3 Partial failure and recovery

HybridOps records stage outcomes under the run and environment that produced them. Completed state is not discarded simply because a later stage fails. This gives recovery, resume, rebuild and teardown paths a known starting point.

The runtime must still respect authority boundaries. A run record is evidence of a HybridOps operation, not a claim that HybridOps is the source of truth for every resource touched by that operation. Authoritative state remains with the system designated to own it, while HybridOps records how that state was resolved, consumed or published during the lifecycle.

5.4 Destructive operations

Rebuild and destroy are first-class lifecycle operations rather than cleanup scripts appended to deployment. Blueprint ordering and operator confirmation are applied to those paths as well.

This is particularly important in recovery and cost-sensitive environments, where an incomplete teardown can be as operationally significant as a failed deployment.

6. Implemented evaluation paths

The implementation is evaluated through four complete operational paths that stress different contract properties. Implementation sources for these paths are collected in the References section rather than embedded as link lists in the main text.

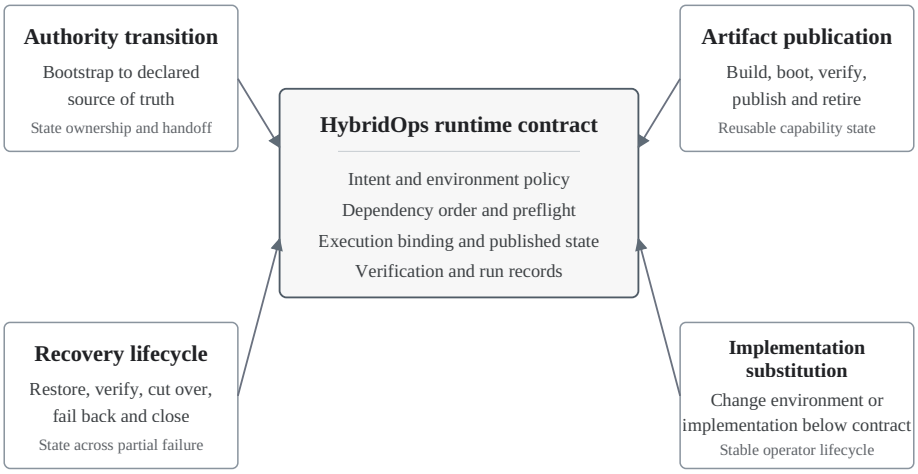

Figure 2. Four implemented paths exercise different boundaries of the same HybridOps runtime contract.

6.1 Authoritative infrastructure foundation

The authoritative foundation tests the transition from bootstrap state to a source-of-truth-governed operating model. Network foundation, machine provisioning, service bootstrap and later platform expansion are composed as separate stages with published state between them.

The important runtime property is authority. Later stages should not silently reconstruct state that has moved behind an authoritative contract. The blueprint therefore exercises dependency publication, policy gates and the handoff from bootstrap to authoritative operation.

6.2 Build, boot, verify, publish

The machine-image lifecycle tests whether an artifact is treated as usable only after it has been built, booted through a consumer path, verified and published for downstream use.

A separate controlled evaluation completed 20 build, boot, verification, publication and retirement observations across four operating-system families, with three additional consumer handoffs resolving and using the published state. Median time to published state ranged from 7.04 to 13.25 minutes in the reported observations. Storage returned to the recorded baseline after each retirement cycle.

The runtime property under test is not image creation alone. It is the publication boundary between an implementation artifact and a reusable capability state.

6.3 Recovery operating model

The PostgreSQL HA recovery path tests a longer lifecycle: backup readiness, restoration, verification, controlled cutover, continued backup, failback and final service state.

Recovery is useful here because success cannot be reduced to one process exit. The runtime must carry state and verification across several stages and must leave enough evidence to determine what happened if the sequence stops partway through.

6.4 Implementation substitution

The governed network-emulation implementation from v1.0 provides one substitution test for the runtime boundary. It exercises two infrastructure routes and two emulation workloads under the same higher-level lifecycle while changing implementation below the contract.

The path covers image and workload preparation, private access, guest connectivity, verification, cost visibility, portable lab-definition persistence, writable QEMU node-state preservation, rebuild and teardown. The test is whether those implementation changes require the operator to adopt a different lifecycle.

The case also demonstrates why published state and explicit closure matter: a cost-bearing host can be removed while portable definitions and writable node state remain recoverable.

7. Operational records and evidence

A HybridOps run record is a runtime artifact, not an after-the-fact report. The runtime writes it as part of command execution.

Representative module records include:

```
<runtime-root>/logs/module/<module_id>/<run_id>/
  driver_meta.json
  driver_result.json
  inputs_runtime.json
  workspace_policy.json
  backend_binding.json
  stdout.redacted
  stderr.redacted
```

The exact files depend on the operation, but the contract is stable: resolved inputs, execution metadata, redacted output and published results are tied to a run identity.

This serves three engineering purposes.

1. **Failure analysis.** The operator can distinguish a contract failure, readiness failure, execution failure and post-execution verification failure.
2. **Operational review.** A later reviewer can inspect what the runtime resolved and observed without reconstructing the operation from terminal history.
3. **Lifecycle handoff.** Published outputs provide an explicit boundary for downstream stages and external consumers.

The evidence model is intentionally non-secret. Secret material must not be required to make the record useful for review.

8. Engineering validation

The current implementation provides evidence for several runtime properties.

Property	Implemented evidence
Contract validation	Module and blueprint validation, deterministic input resolution and dependency-cycle rejection
Environment policy	Profile resolution kept separate from ModuleSpec intent

Property	Implemented evidence
Preflight	Required readiness evaluated before selected lifecycle stages proceed
Implementation binding	Registered Driver and versioned Pack resolved behind the module contract
Dependency publication	Blueprint stages publish and consume declared outputs and capability state
Structured records	Runtime writes redacted command and module records under stable run paths
Lifecycle operations	Deploy, rebuild, recovery and teardown paths exercised through public reference scenarios
Implementation substitution	Network-emulation case exercises multiple infrastructure and workload routes under the same lifecycle model
Controlled artifact publication	Machine-image evaluation completed build, boot, verification, publication, consumer handoff and retirement cycles

These results establish implemented behaviour, not universal operational benefit. The next stage of evaluation should test the runtime with independent operators, additional implementation substitutions, injected partial failures, concurrent execution and longer-lived authority transitions.

9. Limits and open questions

Several areas deserve direct practitioner scrutiny.

Concurrency. The runtime model needs clear behaviour when separate operations address the same environment or publish overlapping state. Locking, conflict detection and idempotent recovery need to remain visible at the contract boundary.

Long-running reconciliation. HybridOps is designed around bounded infrastructure operations and explicit lifecycle transitions. Continuous reconciliation belongs to control planes designed for that model. The handoff between bounded operation and long-running reconciliation should remain explicit.

Authority changes. Bootstrap, source-of-truth adoption, recovery and failback can move authority between systems. The runtime must reject stale or ambiguous authority rather than treating every available state source as equivalent.

Implementation compatibility. Replaceable Packs and Drivers require clear compatibility rules. A version change that alters published outputs or lifecycle semantics is a contract change, not an implementation detail.

Scale. Dependency ordering that is clear in a small blueprint may expose scheduling, observability and failure-recovery limits in larger graphs. These limits should be measured rather than inferred from small scenarios.

These are engineering boundaries for further work, not reasons to narrow the runtime to one provider, workload or infrastructure class.

10. Request for technical review

This paper seeks review from platform engineers, infrastructure engineers, SREs, network engineers, cloud architects and practitioners responsible for operating heterogeneous infrastructure.

The review can focus on any of the following questions:

1. Are the boundaries between ModuleSpec, Profile, Driver, Pack, Blueprint and run record technically coherent?
2. Does the runtime model preserve a meaningful operator contract as implementation and environment details change?
3. Are the preflight, dependency publication, verification and run-record semantics strong enough for partial failure and recovery?
4. Where are the most important authority, concurrency, compatibility or scale limits?
5. Do the public implementation and exercised paths support the claims made in this paper?

Specific criticism, counterexamples and implementation-level observations are welcome.

References

Implementation and evaluation sources

- Muibi, J. (2026). HybridOps Core.
- Muibi, J. (2026). Blueprint catalogue.
- Muibi, J. (2026). Authoritative on-prem foundation.
- Muibi, J. (2026). Authoritative Foundation blueprint.
- Muibi, J. (2026). Template-image module.
- Muibi, J. (2026). *Build, boot, verify, publish*.
- Muibi, J. (2026). PostgreSQL HA DR cycle.
- Muibi, J. (2026). *Recovery is an operating model, not a pipeline*.
- Muibi, J. (2026). Governed network-emulation reference scenario.
- Muibi, J. (2026). *Reproducible network labs without dedicated learner hardware*.
- Muibi, J. (2026). Preflight contract.
- Muibi, J. (2026). Evidence and redaction standard.

Adjacent control models

- HashiCorp (2026). Terraform documentation and Terraform state (accessed 8 August 2026).
- OpenTofu (2026). OpenTofu documentation (accessed 8 August 2026).
- Ansible (2026). Ansible playbooks (accessed 8 August 2026).
- Kubernetes (2026). Controllers and Operator pattern (accessed 8 August 2026).
- Dapr (2026). Dapr overview and building blocks (accessed 8 August 2026).
- Anuket (2026). CNTT reference model (accessed 8 August 2026).

Worked network-emulation case

- Cisco Systems (2026). Cisco Modeling Labs (accessed 28 July 2026).
- Network Development Group (2026). NETLAB+ VE documentation (accessed 28 July 2026).
- GNS3 (2026). GNS3 documentation (accessed 28 July 2026).
- EVE-NG (2026). EVE-NG features (accessed 28 July 2026).