

Executable Higher Semantics for Cloud-Portable Agent Swarms

Typed Specifications, Durable Virtual Machines, and Conditional Compilation

Anonymous

12 October 2026 - Submission manuscript, version 0.1

Abstract

We formalize an agent-swarm framework whose executable specifications can be lowered to a durable virtual machine and implemented through cloud execution, persistence, messaging, and model-generation interfaces. The semantic layer consists of typed state families, directed transition witnesses, finite interaction constraints, and admissibility predicates in an infinity-topos or another explicitly descent-compatible base. A computable presentation supplies serializable states, total guarded handlers, typed effect intents, and realization witnesses; abstract higher structure alone is not an executable representation. We define source and intermediate-language semantics, a version-conditional atomic commit protocol, persistent deduplication, and a provider contract. Written proofs establish finite-trace safety, guard-preserving lowering, at-most-once logical decisions, conditional target-to-source trace refinement, and checkpoint continuation. Model-generated code is admitted only through an explicitly scoped validator. Cloudflare, AWS, and Google Cloud are treated as candidate implementations of the contract, with provider-specific obligations. An executable finite companion compiles a guarded workflow to bytecode and three declarative target plans, checks 54,241 bounded request sequences and 813,615 checkpoint continuations, and detects four missing-hypothesis countermodels. The contribution is a conditional compilation architecture, not a verified production compiler or an automatic implementation of arbitrary higher semantic objects.

1. Problem, contribution, and observation boundary

An agent swarm combines stateful participants, communication, scheduling, code generation, and changes to its own architecture. A specification of its interfaces is insufficient to determine an implementation: execution also depends on request identity, rejection behavior, persistence, external responses, and recovery after interruption. Conversely, an operational framework expressed only through a particular provider’s APIs obscures which properties belong to the swarm and which depend on the provider. We seek a specification that exposes both obligations.

The preceding papers separate higher swarm semantics [1], temporal and interface composition [2], typed numerical compilation [3], stateful objective queries [4], and observation-preserving runtime gates [5]. The present paper connects these ingredients through an executable presentation and a durable machine. Numerical or learning components enter as registered implementations with their existing, restricted contracts. Their mathematical or numerical guarantees are not enlarged by placing them in a swarm.

The framework has three levels. A semantic specification describes what states and executions mean. An executable presentation determines how representable commands produce guarded decisions and effect intents. A deployment realization implements persistence, dispatch, and external

interaction. The compiler takes the second level, with witnesses linking it to the first, and produces an intermediate program plus a target plan for the third.

The main correctness result is finite target-trace inclusion in the source semantics under a stated adapter contract (Theorem 8.3). The commit protocol also yields a useful concrete result: one terminal logical decision per authenticated request key, despite retries and crashes (Theorem 6.2). These results use established simulation and categorical constructions [6-9]. The proposed contribution is their assembly into a swarm language and runtime contract that retains higher witness obligations, generated-code validation, resource authority, and migration state.

Definition 1.1 (Observable logical history). A logical decision is the first durable result assigned to a request key and its bound command. Its observation records the key, acceptance or normalized rejection reason, result identity, and created effect identities. A history orders these decisions by their durable commit order. Repeating a cached reply does not create another decision. Transport attempts, latency, billing, physical invocation counts, and diagnostics belong to a separate operational history.

Our theorem concerns logical histories. It does not assert equality of every observation available from a provider dashboard. If an application observes physical retries, elapsed time, numerical errors, or monetary charges, those quantities must be added to the source semantics and simulation obligations. This boundary is necessary rather than cosmetic: a retried model call can cost money even when only one response is admitted into the logical state.

The paper is self-contained at the specification level. Sections 2-4 define higher meaning and executable syntax; Sections 5-8 define compilation, the VM, and refinement; Sections 9-11 treat portability, synthesis, and migration; Section 12 reports a finite executable instance; Section 13 states scope and research obligations.

2. Semantic specifications and representable executions

Fix universes. All categorical limits and identifications in this section are homotopy coherent. The standard infinity-topos setting supports finite limits and effective descent [6]; related elementary higher topoi can supply the needed structure under explicitly chosen assumptions [7].

Definition 2.1 (Semantic base). An admissible base is an infinity-category $\mathcal{X}$ with finite limits. When contextual descent is used, choose pullback-stable covers $U \rightarrow T$ with Čech nerve $U_\bullet$ such that

$$\mathcal{X}_T \simeq \lim_{[n] \in \Delta} \mathcal{X}_{U_n}. \quad (1)$$

An infinity-topos, with effective epimorphisms as covers, supplies this property. A different structured object is acceptable only for the constructions it actually supports. Probability, resource orders, dependent products, and temporal metrics are additional named structure whenever a claim uses them.

Definition 2.2 (Semantic swarm). A swarm is a diagram

$$\mathfrak{S} = (A, R \rightarrow A, S_{\text{loc}} \rightarrow R, Z \rightarrow A, Z \xleftarrow{s} E \xrightarrow{t} Z, V \hookrightarrow Z, \ell : E \rightarrow L). \quad (2)$$

Here A specifies architectures; R specifies roles; S_{loc} specifies local states; Z is the complete configuration object; E consists of directed permitted execution witnesses; V is the admissibility predicate; and L contains event labels and observations. Wire and shared-resource constraints are included in the diagram. An execution witness is not an identity in Z .

For a fixed finite roster, compatible network executions are finite limits of component, port, and protocol diagrams. Complete state includes whichever buffers, memories, revisions, budgets, or pending obligations the specification observes. A variable roster constructed with a dependent product requires that product and its preservation to be supplied; finite-limit preservation alone does not justify it.

Write $E_V = E \times_{s,Z} V$. A certificate of safety is a target lift

$$q : E_V \longrightarrow V, \quad mq \simeq t \text{ pr}_E, \quad m : V \hookrightarrow Z. \quad (3)$$

Because m is monic, admissibility is propositional. The object of implementations of an admissible state may nevertheless contain multiple inequivalent witnesses. A predicate and an implementation family must not be conflated.

Proposition 2.3 (Finite semantic safety and transport). A finite composable path in E whose initial state lifts to V has every intermediate state in V . A finite-limit-preserving functor transports a fixed finite wiring or path diagram and the certificate (3).

Proof. Apply q to the first event together with its source lift, then repeat at its target. This finite induction supplies a lift at every vertex. Every required path and network object is a finite limit, so the functor preserves its cone. Monomorphy is characterized by the diagonal being an equivalence to the self-pullback and is therefore preserved. Applying the functor to (3) supplies the transported certificate. $\square$

This result says nothing about an infinite roster, an infinite execution, or a scheduler's fairness. Nor does it manufacture executable witnesses from their existence.

Definition 2.4 (Executable presentation). A presentation of $\mathfrak{S}$ consists of a computable state set S_P , a typed command set C_P , a decision function, wire codecs, and a realization assignment in a chosen context T . Each reachable $u \in S_P$ has a map $z_u : T \rightarrow Z$ and an admissibility lift. Each logical source decision has a supplied map $e : T \rightarrow E$ with identified source and target readouts and label interpretation. These assignments must be compatible with substitution when multiple contexts are used. Additional higher composition claims require a coherent extension into a specified operational category.

A sequence of such witnesses gives $T \rightarrow E \times_Z \cdots \times_Z E$ by the pullback universal property. A graph of primitive events does not automatically have the composition maps of a Segal object. The finite executable model in Section 12 uses ordinary sets; the general definition permits richer witnesses without claiming that a finite machine stores every higher identity.

Realization is a compilation input or proof obligation. A compiler cannot infer a coherent global implementation from pointwise inhabitation of implementation fibers. For example, in the topos of

C_2 -sets, a two-element set with the swapping action covers the terminal object but has no equivariant global element. Descent data can reconstruct structure; they do not constitute a choice algorithm for an arbitrary realization family.

3. SwarmSpec: a typed executable language

We call the proposed source language SwarmSpec. Its trusted computational core uses finite algebraic data, exact natural-number metadata, finite maps and lists, and a registry of total pure primitives. Bytes are opaque until a specified decoder validates them. A deployment profile fixes bounds on every stored value and atomic write set. The use of natural numbers in the semantics does not license silent overflow in a machine representation.

Definition 3.1 (Module). A module is

$$P = (\Sigma, S_P, C_P, s_0, V_P, H, \text{Caps}, \text{Obs}, \text{Codec}, \text{Real}). \quad (4)$$

Σ is the typed primitive registry; s_0 is the initial state; V_P is the full invariant; H is a finite collection of guarded handlers; **Caps** is the authority context; and **Real** contains Definition 2.4's realization obligations. A handler consumes one typed command and a state snapshot, evaluates pure expressions, and returns a business-state proposal, a finite ordered list of effect intents, and an observation. Each command variant has one handler, or an explicitly ordered and exhaustive decision tree whose order is part of the semantics. It does not execute an external effect.

The abstract grammar is as follows. Named records and enumerations are finite declarations; collection bounds and numeric ranges are profile parameters.

```

type    ::= Unit | Bool | Nat | Bytes | Enum(names)
          | Record(fields) | Sum(variants) | List(type,bound)
          | Map(key,type,bound)
expr    ::= value | field | pure_primitive(exprs)
          | if expr then expr else expr
handler ::= on command when guard
          write assignments
          intend typed_intents
          observe observation
module  ::= types; state; initial; capabilities; handlers;
          invariant; codecs; realization_obligations

```

External model generation, execution, storage, and publication use typed intent constructors. Awaiting a response is expressed as a future input command, bound to an existing ticket; it is not an implicit blocking operation inside a handler. Long-running computations split at explicit durable continuations. Every handler must terminate, for example by structural recursion on bounded collections or a separately proved total primitive.

Definition 3.2 (Effect and authority typing). The judgment

$$\Gamma; \Delta \vdash h : S_P \times C_h \longrightarrow S_{\text{biz}} \times \text{List}(\text{Intent}) \times \text{Obs} ! \varepsilon \quad (5)$$

means that h returns well-typed proposals and uses only effects in the finite row ε . The persistent capability context Γ names reusable restricted authorities; Δ names consumable reservations, tickets, and budgets. Producing an intent requires its effect capability and consumes or transfers the specified linear resources. A runtime resource ledger enforces the same rule at commit time.

For example, a generation intent requires a model endpoint capability, input-artifact read authority, and a reserved call budget. A code execution intent requires a validated artifact identity and an execution capability scoped to the target and resources. A generated program cannot enlarge these authorities by writing a capability name into its output.

The type system checks interface compatibility and resource usage. An invariant such as “no unreviewed artifact executes” still requires a handler certificate or a trusted invariant gate. Ordinary typing does not establish arbitrary semantic assertions. Pure primitives have explicit totality, typing, and meaning contracts; a foreign numerical primitive additionally has its declared numerical contract.

Definition 3.3 (Logical source decision). Write a state as

$$u = (b, a, v, N, O, J, K), \quad (6)$$

where b is business state, a the architecture epoch, v the logical revision, N the inbox of terminal decisions, O the logical intent ledger, J the decision and oracle records, and K the resource/capability ledger. A fresh authenticated command is (k, c) , with key k absent from N . Evaluate the handler to obtain (b', I, o) , or a normalized guard rejection $(b, [], o)$. Allocate intent identities (a, k, i) in list order, debit the required reservations, append the decision, store the command binding and result in N , and increment v . This defines the total fresh-decision function $F_P(u, k, c)$.

Keys are scoped to a swarm, authenticated principal, and admission namespace, with a client nonce inside that scope. The binding includes the complete canonical command or a collision-resistant digest under an explicit cryptographic assumption. For exact mathematical identity, use the bytes themselves. A reused key with different bytes is a protocol conflict. A repeated identical request returns the existing terminal decision. Those responses affect the operational history but do not append another logical decision.

Every fresh rejection is retained. Thus “rejection preserves state” means preservation of the business-state component, not of the entire durable record: revision, inbox, and audit data change. Unsupported commands fail at a typed boundary or enter an explicit unsupported-command variant. They cannot be coerced into a nearby permitted command.

4. Invariants, composition, and certificates

The invariant V_P includes business validity, resource accounting, command binding, and intent consistency. A typical instance asserts that every intent identity is unique, each intent is linked to a recorded decision, each accepted code-execution intent names an artifact with the required validator receipt, and each live ticket belongs to the current architecture epoch. Fields omitted from V_P receive no safety conclusion.

Definition 4.1 (Handler certificate). A certificate for h proves, for all fresh well-typed inputs,

$$V_P(u) \implies V_P(F_P(u, k, c)), \quad (7)$$

and provides the corresponding realization witness of Definition 2.4. A guard restricts the permitted transition relation. It must be represented in the source protocol rather than justified merely because its outputs happen to satisfy V_P .

Theorem 4.2 (Source safety). If $V_P(s_0)$ and every handler has (7), every finite logical source execution remains in V_P and realizes a certified finite path in the semantic swarm.

Proof. Induct on the number of fresh logical decisions. The base case is the initial certificate. The induction step applies (7). The realization assignment supplies the new semantic event and identified endpoints; adjoining it to the previous path uses a pullback. Proposition 2.3 supplies the admissibility lift along that realized path. $\square$

Proposition 4.3 (Compositional certificate rule). Suppose $V_P = \bigwedge_{j=1}^r V_j$. For each handler, prove preservation of every V_j that can depend on a written field, and prove a frame rule for the remaining predicates. Then (7) holds.

Proof. Each conjunct is preserved either by its supplied preservation proof or by its frame proof. Combining the conclusions proves their conjunction. The write/read dependency analysis is sound only if it includes the full semantic dependencies of each predicate, including shared-resource and pending-obligation fields. $\square$

A component-local certificate is therefore insufficient when another component can invalidate a shared constraint. Two agents with separate local “free resource” flags cannot both acquire a globally exclusive resource without a shared guard or atomic authority ledger.

Finite network constraints remain interpreted by finite limits. Coherent equivalence of the complete component, port, and protocol diagram induces equivalence of the resulting network object. This familiar compositional principle [9] justifies replacing a component’s representation only when the boundary and protocol witnesses are also preserved. Replacing an implementation with the same endpoint state set but different executions can change the protocol and observations.

Proposition 4.4 (Contextual structure descent). Under (1), a coherent descent datum of semantic swarm diagrams and certificates descends to $\mathcal{X}_T$. Pullback commutes with fixed finite network and path constructions.

Proof. Apply the diagram functor category to (1). Functor categories preserve limits, so objects, maps, and coherences descend together. Pullback between slices preserves finite limits. The limiting-cone and monomorphism conditions are invariant under equivalence of categories, and (3) descends as a map with its identification. $\square$

Neither this proposition nor finite-limit transport proves that separately generated provider adapters are coherent implementations. Adapter realizations and their overlap agreements must be supplied. This separates categorical descent from compiler validation and prevents an unjustified local-to-global implementation claim.

5. SwarmIR and the compiler

SwarmIR is a typed, provider-independent control-flow language. Its persistent state schema is (6). Its operations manipulate pure values, evaluate guards, create intent descriptions, and propose one terminal decision. It has no direct network or storage-write instruction inside a handler block.

```

instruction ::= LOAD field | CONST value | PURE primitive
              | BRANCH label label | GUARD reason
              | STORE field value | INTENT constructor args
terminator  ::= REJECT reason | COMMIT observation
program     ::= typed_schema; entry_points; acyclic_handler_blocks;
              effect_manifest; codec_manifest; obligations

```

The complete language can support proved terminating recursion, but the reference fragment uses acyclic handlers. Execution on a snapshot produces a proposal, never a partial durable update. STORE changes a private working copy. INTENT builds a typed description; a later dispatcher invokes the corresponding adapter after the atomic commit.

Definition 5.1 (Compilation result). A compiler returns either an error with an unmet obligation or

$$\text{compile}_B(P) = (Q, M_B, \text{Req}, \text{Cert}), \quad (8)$$

where Q is SwarmIR, M_B is the deployment plan, Req lists backend capabilities and resource bounds, and Cert records checked or outstanding proof obligations. A deployment plan identifies entry points, atomic storage domain, routing, credentials by reference, content-addressed artifacts, and recovery procedures. It contains no raw credentials.

Compilation proceeds through type/effect checking, explicit elaboration of fresh-request semantics, normalization to pure handler blocks, guard and intent lowering, codec generation, and profile selection. Handler branching order and rejection reasons are retained. A plan is rejected if required persistence, effect isolation, identity range, or resource bounds cannot be provided. Provider syntax generation is a final implementation pass with its own validation.

Lemma 5.2 (Pure expression lowering). For the first-order fragment with constants, typed field reads, Boolean conjunction, equality, and conditional branching, structural lowering to a typed stack machine preserves the value of every expression on a well-typed snapshot.

Proof. Constants and reads push their specified values. In the equality and conjunction cases, the induction hypotheses give the same operand values, and the registered operation has the same typed meaning. A conditional evaluates the same guard, selects the same branch, and applies its induction hypothesis. Stack typing ensures the operands are present and that the final stack contains the expression’s result. $\square$

Theorem 5.3 (Guard-preserving handler lowering). Suppose expression lowering satisfies Lemma 5.2, assignments use the source’s specified snapshot or explicitly sequenced evaluation convention, and intent allocation preserves order and arguments. Then source and IR handlers return equal business proposals, intent lists, and observations on every well-typed snapshot and command.

Proof. The guard has the same Boolean result. A failed guard yields the same rejection, original business state, and empty intent list. A successful guard follows the same branch. Each assignment

computes the same value under the agreed evaluation convention; induction on the ordered assignment sequence gives the same working state. Each intent has the same constructor and arguments, and list-order preservation gives equal intent lists. The terminal observation is likewise preserved. $\square$

The general source can specify simultaneous assignments; the compiler must evaluate their right-hand sides into temporary values before writing. The finite companion uses literal assignments, so this issue cannot arise there.

A legal optimization must preserve decisions and intent identities as well as business values. Removing an apparently unused generation call changes the intent history and budget. Reassociating a floating-point primitive requires a declared numerical allowance. Moving an effect across a commit boundary changes crash behavior. Thus ordinary pure-value equality is insufficient as an effectful rewrite criterion.

The theorem proves the described lowering rule relative to primitive contracts. It does not verify a provider SDK, container builder, JavaScript implementation, or general optimizing compiler. The executable companion validates a small implemented lowering; it is not a mechanization of these general proofs.

6. Durable VM operational semantics

The Swarm VM consists of a decision engine, a transactional state store, dispatchers, and typed response adapters. Let its durable logical state be u from (6). Physical state additionally contains transport acknowledgments, delivery attempts, and ephemeral computations. Only the commit point changes the logical decision history.

Definition 6.1 (Atomic decision protocol). For an authenticated envelope (k, c) :

1. Decode the exact schema and command binding. Read the inbox. If the key has the identical bound command, return its stored terminal result; if it has another binding, return a key-conflict diagnostic.
2. Read a consistent snapshot u and revision v . Run the pure IR handler on that snapshot. Validate its proposed complete logical result against F_P , or use a compiler-correct handler with the equivalent contract.
3. In one serializable atomic operation, require the revision still to be v and the key still absent. Write the next business and control state, the terminal inbox entry, the decision record, resource ledger changes, and all new outbox intents.
4. After durable commit, return the stored result. On a conditional-write conflict, reread and retry. On uncertainty about whether the commit occurred, resolve it by the durable key before proposing another decision.

There is no external I/O in step 3’s transaction body. Providers that rerun transaction callbacks must be able to rerun only pure evaluation. An acknowledgment before durable commitment is prohibited by the protocol.

Theorem 6.2 (Unique logical decision under retry). Assume the atomic operation in Definition 6.1 serializes conflicting updates and the inbox binding is retained throughout the accepted replay horizon. Then at most one logical decision is committed for a given key. Every retry with the same command after that commit returns its existing terminal result.

Proof. Suppose two commits for the same key exist, ordered by serialization. The first inserts the key. The second requires the key absent in its atomic precondition, contradicting that insertion

and retention. A retry after commitment reads the bound terminal entry and returns it without appending a decision. A concurrent stale snapshot cannot defeat the atomic absence and revision checks. $\square$

Crashing before step 3 leaves no durable decision and permits recomputation. Crashing after step 3 but before the reply preserves the inbox and outbox, so a retry is cached. The theorem is at-most-once, not eventual completion: a permanently unavailable store can prevent every commit.

Proposition 6.3 (Invariant preservation by the VM). Under Theorem 5.3, the backend’s atomic decision contract, and Theorem 4.2’s certificates, every committed logical state satisfies V_P .

Proof. The atomic operation commits exactly a source decision computed from its checked predecessor revision. Intermediate private working states are not logical states. Apply Theorem 5.3 to identify the IR result, then (7). Serialization orders every finite sequence of successful commits, permitting induction. Failed attempts and cached replies do not change logical state. $\square$

Definition 6.4 (Correspondence gate). For a fresh request, a gate accepts a candidate next logical state $\hat{u}$ precisely when

$$\Gamma_P(u, k, c, \hat{u}) = \text{accept} \iff \hat{u} = F_P(u, k, c). \quad (9)$$

Equality covers the complete logical record, including payload identities, observations, reservations, inbox entries, and intent arguments. A weaker gate that compares only a control projection certifies only that projection. The gate itself, its decoder, and its primitive registry are trusted unless separately verified.

Lemma 6.5 (Gate soundness). A candidate admitted by (9) is a source decision and preserves every certified invariant (7).

Proof. Substitute the equality from (9) into (7). $\square$

Translation validation of individual handler or transition results can reduce trust in a code generator, following the general separation between compiler verification and output validation [8]. It cannot compensate for an incorrect specification or an unsound gate.

Deduplication garbage collection is semantically relevant. A bounded replay horizon requires an authenticated epoch or watermark protocol that makes older requests permanently inadmissible. Deleting an inbox record while still allowing its old key destroys Theorem 6.2. Garbage collection must therefore be a specified state transition, not an unobserved storage optimization.

7. Effect dispatch, fencing, and progress

An outbox record is an intent to perform an external operation. A dispatcher may publish or invoke it repeatedly after timeouts or lost acknowledgments. All attempts retain the logical identity (a, k, i) and the same argument binding. Dispatch bookkeeping never rewrites a committed logical intent.

Definition 7.1 (Destination contract). An idempotent destination durably binds an effect identity to one argument record and one terminal result, returns that result for repetitions, and rejects a different argument binding. If execution changes protected resources, it also enforces the supplied fencing epoch at the actual mutation point. A proxy that merely remembers completed calls after forwarding them does not meet this contract.

Proposition 7.2 (One logical destination effect). Under Definition 7.1, arbitrarily many attempts for a committed intent cause at most one destination mutation with that identity. If the destination and result-delivery mechanisms eventually succeed, the result can be admitted through a guarded response command.

Proof. The destination’s atomic identity binding excludes a second mutation, by the argument of Theorem 6.2 in its own domain. Repeated attempts retrieve the same terminal result. Eventual successful return and delivery provide a response command; whether its guard accepts depends on the current epoch and pending-ticket state. $\square$

Countermodel 7.3 (External exactly-once without a destination contract). An external service increments a counter. The VM invokes it; the service increments; the reply is lost before the VM records success. After recovery, the VM cannot distinguish this execution from one where the invocation never reached the service. Retrying can increment twice; refusing to retry can lose the operation in the second execution. A local outbox or inbox alone does not solve this uncertainty.

The framework therefore promises unique committed intents and logical decisions. Exactly one external mutation additionally needs destination idempotence, transactional coupling with the destination, or another specifically proved protocol. Model-generation APIs need not meet that condition. Repeated generation can produce multiple candidates and multiple charges; the source accepts one guarded response and retains the accepted oracle record. Physical cost bounds require reservations for attempts, not merely for logical intents.

Fencing also concerns authority. An old worker may continue running after a lease expires. Incrementing an epoch in the coordinator stops old receipts from committing into the swarm but does not stop an external resource mutation unless that resource checks the epoch. Time-based leases need clock and expiry hypotheses; a monotone fencing token does not by itself provide a real-time guarantee.

Proposition 7.4 (Conditional progress). Fix a particular pending logical intent. Suppose it is not canceled, remains eligible, dispatch is fair, retries are not permanently exhausted, required services eventually remain available, and a valid response is eventually delivered and scheduled. Suppose also that the response guard remains enabled until commitment and that competing commits do not starve it. Then the intent eventually reaches its specified terminal state.

Proof. Fair dispatch and eventual service availability produce a valid response attempt. Eventual delivery and scheduling bring it to the decision engine. Its guard remains enabled. The nonstarvation and eventual-store-availability hypotheses ensure a successful atomic commit after finitely many conflicts. That commit supplies the terminal transition. $\square$

These assumptions are stronger than safety. An execution that repeatedly crashes before commit, or a scheduler that never dispatches a particular intent, satisfies all invariants while making no progress. Dead-letter routing, quota exhaustion, and exhausted retry policies must create explicit failure states or invalidate the progress assumptions; they cannot be silently regarded as successful delivery.

8. Refinement and the conditional portability theorem

Let $\text{Tr}_{\text{fin}}(P)$ be the set of finite logical histories of the source semantics. A target execution is a sequence of physical provider and adapter steps, including reads, failed transactions, dispatch, and interruptions. Let hide_B retain first committed decisions and erase physical attempts according to Definition 1.1.

Definition 8.1 (Backend realization contract). A backend B realizes a compiled module when it supplies:

1. An abstraction α_B from reachable physical states to the complete logical state, with the initial-state correspondence.
2. Exact typed encoding and decoding, identity preservation, and atomic updates to state, inbox, outbox, and resources with the protocol's revision and key preconditions.
3. A step classification: each reachable physical transition either leaves α_B and logical history unchanged, or implements one F_P decision with the same observation and complete logical effect record.
4. Conforming effect and response adapters, authority isolation, and explicit resource/retention bounds. Additional external-mutation guarantees require Definition 7.1.

Provider documentation can motivate an implementation of these clauses; it is not itself a proof of the adapter. The classification must include crashes and recovery states rather than only successful happy-path calls.

Lemma 8.2 (Establishing the step classification). Suppose an adapter implements Definition 6.1, preserves logical fields through exact codecs, exposes an indivisible durable transaction point, and confines dispatch/recovery bookkeeping to nonlogical fields. With Theorem 5.3's lowering, its decision-engine steps have the classification in Definition 8.1.

Proof. Decoding, snapshot reads, and pure proposal evaluation change no logical field. A failed conditional transaction changes none; a successful transaction has one linearization point and writes precisely the source decision, by Theorem 5.3 and the atomic protocol. Reading a cached entry changes none. A crash before or after that point preserves the corresponding durable state; recovery resolves an uncertain outcome by its retained inbox entry. Confined dispatch bookkeeping changes only fields forgotten by the abstraction. Admitting a response is a new typed command processed by the same argument. These cases exhaust the described engine operations. Any adapter operation outside them requires its own case proof. $\square$

Theorem 8.3 (Finite target-to-source trace refinement). Let P satisfy Theorem 4.2, let its IR satisfy Theorem 5.3, and let B satisfy Definition 8.1. Then

$$\text{hide}_B(\text{Tr}_{\text{fin}}(\text{run}_B(\text{compile}_B(P)))) \subseteq \text{Tr}_{\text{fin}}(P). \quad (10)$$

Every reachable target logical state satisfies V_P and its logical decisions realize certified finite semantic paths.

Proof. Start with the initial-state correspondence. Induct on a finite physical execution. A classified silent step preserves the abstraction and adds no logical decision, so the existing source prefix remains a witness. A classified commit implements one F_P step with the same observation and effect record; append that source decision. Theorem 5.3 connects the handler lowering to this step.

Theorem 4.2 preserves V_P and supplies the semantic event witness. Repeating the argument yields (10) and the stated endpoint property. $\square$

The theorem applies to every backend that meets the contract. Its quantification is a conditional portability statement, not a claim that every cloud provider already satisfies it. It also shows why compilation may reject a target: omitting an atomic primitive or exceeding a transaction bound removes a hypothesis.

Corollary 8.4 (Preservation of finite safety properties). If a predicate on logical histories holds for every finite source history, it holds for every hidden finite target history.

Proof. Apply (10) to each target history. $\square$

Finite trace inclusion does not establish divergence-sensitive equivalence, liveness, deadlines, or equality of all source behaviors. A target can serialize a nondeterministic source and realize fewer schedules. Two targets may return different generated artifacts, each permitted by the source oracle relation. Equality of deterministic executions is available only after fixing the same logical input order and accepted oracle transcript and proving that both targets complete those inputs.

Composition of compilation passes is likewise conditional. If each pass simulates its target steps by zero or more predecessor steps with the same projected observations, compose the abstraction relations and concatenate their finite witnesses. A final cloud adapter cannot inherit correctness merely because its bytecode is well typed.

9. Cloud profiles and code-generation interfaces

The baseline profile uses one atomic coordinator domain per swarm. Worker computation can be distributed and parallel; admission of its decisions is serialized through that domain. This avoids assuming a transaction that spans unrelated agents, object stores, message brokers, and model APIs. Sharding a global invariant requires a separately proved ownership partition or coordination protocol.

The following mappings are proposed implementations, grounded in current official API documentation accessed on 12 October 2026. They are deployment obligations, not measured cloud experiments.

Target	Decision engine	Atomic coordinator	Transport
Cloudflare	Worker handler	One SQLite-backed Durable Object	Queues
AWS	Lambda handler	DynamoDB transaction within one Region	SQS
Google Cloud	Cloud Run service	Firestore server transaction	Pub/Sub

For Cloudflare, Durable Object storage documents transactional, strongly consistent storage private to a unique object [10]. A synchronous transaction can hold the coordinator record, inbox, intent records, and resource ledger. SQL cursors must be consumed within their supported snapshot discipline; no model or network call belongs in the transaction. Cloudflare Queues documents at-least-once delivery [11], so logical identity and persistent deduplication remain necessary.

For AWS, DynamoDB transactions provide atomic multi-item writes within their documented scope [12]. The adapter binds the revision condition to the state write and writes distinct inbox and outbox items in the same transaction. It must not generate two transaction actions for the same item. The documented transaction limit is 100 actions and 4 MB aggregate; individual item limits also apply. The compiler must bound or split handlers before deployment, preserving semantics through explicit staged decisions if splitting is necessary. DynamoDB’s client-token idempotency window is not a substitute for a durable inbox. Lambda’s SQS integration can deliver records repeatedly [13].

For Google Cloud, Firestore documents transactions serialized and isolated by commit time [14]. Use server-side transactions that read before writing, bind the command to its terminal inbox record, and atomically update the coordinator and intent documents. A callback retried by the SDK must have no external side effects. Pub/Sub’s optional exactly-once delivery has subscription and regional qualifications, including pull-only support [15]. Those qualifications do not establish exactly-once external mutation or remove application-level command identity.

Each profile needs a separate immutable-artifact adapter. A content-addressed key names bytes with schema, length, and digest validation. Publishing an intent that depends on an artifact requires the artifact’s durable availability before the decision, or a specified staged upload/availability protocol. The object store and coordinator are not silently placed in one transaction. Orphaned immutable uploads can be collected after a reachability and retention check.

Interface	Abstract requirement	Candidate API family
Generate candidate	Typed request; response binding; explicit oracle record	Workers AI; Bedrock Converse; Vertex AI-compatible generation
Execute artifact	Restricted runtime ABI; validated artifact; response ticket	Worker/Wasm; Lambda package; Cloud Run container
Store artifact	Immutable bytes; verified identity; declared availability	Configured provider object-store adapter
Commit decision	Atomic state, inbox, ledger, outbox; fencing	Coordinator transaction from the first table

Workers AI exposes model invocation through bindings or an API [16]. Bedrock Converse supplies a message interface for supporting models [17]. Google’s generation endpoint exposes a typed generate-content operation [18]; its current documentation redirects the historical Vertex AI path to the Gemini Enterprise Agent Platform reference. These are ways to request generated text or candidate code, not correctness certificates for that code. A model or external generator may be selected independently of the compute provider.

Execution portability concerns an ABI, not arbitrary source languages. A module targeting Worker/Wasm must fit the supported runtime and imports [19]. A Lambda package must fit its selected runtime and invocation interface. Cloud Run can execute packaged code or containers [20]. A portable task declares its dependencies, memory, time, input/output codec, and effect imports. A task requiring unsupported native libraries is rejected for the Worker profile or assigned to an explicitly declared remote execution capability; it is not silently translated into equivalent Wasm.

The baseline compiler produces a plan whose requirements can be checked before credentials or deployment are supplied. A new provider is supported by implementing the same VM and adapter contract. An API for execution and a persistent store with only eventual consistency do not alone establish that contract: a stronger coordinator service or a weaker explicitly named semantics is required.

10. Generated programs, learning components, and authority

An agent that writes code is part of the environment-facing computation, not an unrestricted trusted compiler pass. Its output is a candidate artifact. We distinguish syntactic well-formedness, sandbox confinement, tested behavior, and a proof of a semantic specification.

Definition 10.1 (Validated artifact). For a contract C and bytes p , an admission receipt contains the exact artifact binding, source contract and dependency identities, validator version, checked obligation, declared effects, runtime ABI, and evidence. Let $\text{Check}_C(p, w)$ be its checkable admission predicate. A semantic assurance claim requires an independently supplied soundness implication

$$\text{Check}_C(p, w) = \text{true} \implies \text{Meaning}(p) \models C. \quad (11)$$

Bounded tests satisfy only their stated finite test obligation. A signature establishes provenance or authorization, not semantic soundness. A proof-carrying receipt requires a trusted proof checker and a specification of the machine semantics it checks.

Proposition 10.2 (Conditional synthesis safety). Suppose every execution intent requires a receipt bound to its exact bytes, contract, ABI, and effects; (11) holds; and execution adapters preserve that ABI and authority boundary. Then accepted execution intents satisfy the admitted contract C .

Proof. The intent guard supplies a matching successful receipt. Equation (11) gives the candidate’s semantic property. Adapter preservation transfers that property to the specified execution interface. The authority boundary excludes unlisted imports; its enforcement is an explicit hypothesis. $\square$

A compromised generator or a misleading prompt cannot bypass that guard through ordinary candidate output. A compromised validator or an overprivileged execution adapter can violate the theorem’s hypotheses. Restricting a program’s tools and credentials supplies an enforceable boundary only if the runtime mediates every relevant effect. Merely recording an allowed-tool list does not do so.

Generation follows an explicit protocol: commit a generation intent; obtain candidate bytes and provider metadata; store their immutable identity; commit a bound candidate receipt; validate; then commit admission or rejection. Generation retries may return different bytes. The first admissible bound response wins under the source response handler; later responses are stale or separately recorded candidates according to the chosen protocol. Replay consumes the recorded accepted response rather than invoking the model again.

Numerical agents can use LensIR primitives and optimizer query contracts [3,4]. Their state retains parameters, optimizer memory, objective identities, request tickets, and query records. A provider switch preserving only tensor shape or business-control state is insufficient. Real-number derivative correctness, machine arithmetic, approximate numerical agreement, and declared surrogate responses remain distinct claims. Swarm-level invariant proofs can depend on a numerical-domain receipt, but then that receipt’s soundness joins the trust boundary.

Budget accounting is similarly explicit. A resource reservation is taken before a physical attempt under a cost-observing profile. Unknown completion may retain a reservation until reconciliation. A logical call budget alone cannot bound repeated billable physical calls. A profile with such a weaker budget must say so.

11. Checkpoints, migration, and correction compatibility

A continuation-complete checkpoint retains all state that can affect permitted future logical histories: business and optimizer memory, architecture epoch, logical revision, command bindings, pending tickets, effect identities and arguments, accepted oracle records, capabilities by scoped reference, validator receipts, and artifact identities. It also retains recovery-relevant dispatch metadata or an equivalent replay protocol. Secret material need not be embedded; its referenced authority and availability must be restored.

Definition 11.1 (Typed checkpoint contract). A codec has encoder enc , partial decoder dec , and domain D satisfying

$$\text{dec}(\text{enc}(u)) = u \quad (u \in D). \quad (12)$$

It rejects out-of-range metadata, malformed sums, ambiguous numbers, and unavailable schema versions. A restore plan also establishes that every referenced artifact and authority exists with its recorded meaning.

Lemma 11.2 (Injectivity and deterministic continuation). Equation (12) makes enc injective on D . If F_P is deterministic for a fixed typed command and recorded-oracle stream, an exact restore has the same finite continuation under that stream.

Proof. If two encodings are equal, applying dec yields equal source states. For continuation, start with equality from (12) and induct on commands, applying the same function F_P and the same recorded response at each step. $\square$

The codec law alone does not restore a deleted external object or reproduce an unrecorded model answer. It also cannot recover an external action whose outcome was unknown without a recovery protocol. These are separate operational obligations.

Definition 11.3 (Cross-provider migration). A migration freezes or fences old admissions, obtains a consistent checkpoint at a logical revision, transfers required immutable artifacts and obligations, restores on the new backend, validates the new abstraction and capability bindings, and changes routing only through an atomic authority handoff. Old workers' receipts carry their old epoch and cannot commit into the new epoch. External mutations require destination fencing or safe completion before handoff.

A coordinator-local epoch change does not create a cross-cloud transaction. If the authority handoff itself is distributed, its implementation needs a shared linearizable authority, consensus, or another proved single-writer protocol. During an uncertain handoff the framework may stop admissions; availability through arbitrary partitions is not asserted.

A change of epoch or schema is an explicit certified source reconfiguration decision, with a conversion of pending obligations and realization witnesses. The comparison boundary is its resulting handoff state. A change of physical location alone leaves the logical state fixed. Copying a checkpoint and editing its epoch outside the source protocol is not a certified migration.

Theorem 11.4 (Conditional migration continuation). Suppose migration establishes equal continuation-complete logical states, equivalent pending external obligations and recovery protocols, compatible realization witnesses, and a unique fenced admission authority. If both backends satisfy Definition 8.1, their post-handoff finite logical executions refine the same source continuation. For a fixed input and accepted-oracle stream on which both complete, their logical histories agree.

Proof. Use Lemma 11.2 for the restored logical state. Equivalent pending obligations supply the same admissible response interface; compatible realization witnesses identify the semantic starting boundary. Unique admission authority excludes competing unfenced continuations. Apply Theorem 8.3 to each backend. For a fixed completed deterministic stream, the source continuation is unique, so the two logical histories agree. $\square$

A schema correction is a new versioned specification, not an assertion that all old claims remain true. Preserve the old source, observation contract, evidence, and counterexample under immutable identities. Supply an explicit conversion or refinement relation to the corrected state and prove the invariants actually transported. Pending obligations that cannot be converted must be settled, rejected, or represented in a legacy compartment with its original semantics.

For example, a legacy completion ticket lacking an architecture epoch cannot be silently assigned the current epoch: doing so can accept stale work. Retaining that format as a historical artifact, rejecting ambiguous live tickets, and documenting a proved conversion for already-settled histories preserves research relevance without preserving an incorrect operational assertion. This is the correction discipline adopted by the runtime-gate lineage [5].

12. Executable finite instance and reproducibility

The companion implements a finite SwarmSpec fragment, its stack-bytecode compiler and interpreter, a simulated atomic VM with inbox/outbox, canonical checkpoints, and three declarative target plans. It uses only the Python standard library. It makes no cloud API calls and does not deploy a service.

The example has proposer, reviewer, and executor roles. Its business state is

$$b = (\text{stage}, \text{artifact}), \quad \text{stage} \in \{\text{idle}, \text{proposed}, \text{reviewed}, \text{done}\}, \quad \text{artifact} \in \{\text{none}, \text{good}, \text{bad}\}. \quad (13)$$

Admissibility permits exactly five states: idle/none, proposed/good, proposed/bad, reviewed/good, and done/good. The good/bad symbols are an abstract validator result, not a claim about actual generated source code. Five command variants propose either candidate, review, run, or abort. Review requires a proposed good artifact; run requires reviewed state and emits one execute-good intent; abort returns a proposed candidate to idle. Failed guards preserve business state and record rejection.

For illustration, the review handler and its lowering are:

```
on review when stage == proposed and artifact == good
    write stage = reviewed
    intend []
    observe accepted
```

```

LOAD stage; CONST proposed; EQ
LOAD artifact; CONST good; EQ
AND; GUARD guard
STORE stage reviewed; COMMIT accepted

```

A separate direct transition table is the test oracle. It does not reuse the source handler representation, expression evaluator, or bytecode interpreter. Compiling the same handlers for three profiles retains identical control bytecode and changes only the declarative deployment plan. Thus profile agreement is not a test of any provider service.

The enumeration uses request keys $\{0, 1, 2\}$ in one fixed admission namespace and the five commands, making a 15-symbol request alphabet. Every word of length zero through four is executed from the initial state. Its first-use requests are compared with the direct transition table; repeated identical and conflicting keys must leave the durable checkpoint unchanged. At each word’s endpoint, checkpoint restore is compared with uninterrupted execution for every next request in the alphabet.

Check	Observed result
Business states and commands	5 admissible states; 5 variants
Source/bytecode comparisons	75 across three target plans
Request words through length four	54,241
Restored next-request comparisons	813,615
Cached or conflicting steps inspected	83,700
Crash and competing-snapshot scenarios	3 pass
Destination redelivery scenario	4 attempts; 1 idempotent logical effect
Missing-hypothesis controls	All four detected

The crash scenarios cover interruption before commit, interruption after commit before acknowledgment, and two proposals derived from the same revision. The destination simulation returns one effect for four repeated attempts. These tests assume an indivisible store update; they do not test an SDK’s atomicity.

The negative controls remove the artifact check from review, expose a state write without its outbox intent, use a nonidempotent destination, and remove required backend capabilities from the plan. They respectively witness invariant failure, lost effect intent, repeated external mutation, and compilation rejection. They are counterexamples to missing hypotheses rather than simulated production fault injection.

The canonical example checkpoint stores business state, revision, inbox command bindings/results, outbox records, and the decision trace. The restore checker validates canonical encoding and business admissibility for checkpoints produced by this model. It is not a hardened arbitrary-input decoder for the complete framework. The artifact’s general codec obligations remain separate.

To reproduce, run the companion’s `reference_vm.py` with Python 3.12 or later. It regenerates `example-spec.json`, three target-plan JSON files, and `verification-results.json`. The manuscript sources and build instructions accompany the paper. The reported enumeration is bounded evidence for this finite ordinary model; it proves neither arbitrary program correctness nor the general higher-categorical results.

13. Scope, related work, and remaining obligations

The categorical foundations are standard higher-topos descent and finite-limit semantics [6,7]. Compositional dynamical systems provide a precedent for structured interfaces and temporal behavior [9]. Semantic preservation and translation validation are established compiler methods [8]. The earlier swarm and runtime papers [1,2,5] supply the research lineage. This paper’s contribution is the explicit executable boundary and the joint contract tying directed semantic witnesses to a durable, effect-aware cloud VM.

The formal results are written mathematical proofs. They have not been checked by a proof assistant. The implemented finite compiler is intentionally smaller than the proposed language. The three target plans are declarative profiles, with no implemented provider SDK adapters and no cloud deployment experiment. Documented service semantics support the proposed architecture but do not discharge Definition 8.1 for an actual deployment.

A production milestone would implement one provider adapter, verify its crash-state abstraction and codec, validate authorization and resource limits, and then prove or test a second adapter against the same source observations. Extending beyond the single coordinator requires a compositional ownership or distributed transaction theorem. Unbounded populations, probabilistic equivalence, Byzantine participants, quantitative cost bounds, deadlines, and convergence of learning agents require additional semantic structure and results.

The framework nevertheless makes compilability precise: a supported finite presentation lowers to pure decision code and typed intents; a target realizing the required durable VM contract preserves its finite logical behavior. Higher semantics determines which executions and witnesses are admissible. The compiler and VM specify how a representable part of that semantics is realized, recovered, and transported between providers.

Automation disclosure. OpenAI ChatGPT/Codex assisted with formalization, written proofs, source development, bounded checks, reference checking, and typesetting. The manuscript is prepared for journal review under an Anonymous byline. No independent referee assessment or acceptance decision is asserted.

References

1. Anonymous. *Higher Semantics of Agent Swarms: Contexts, Directed Executions, and Certified Reconfiguration*. Higher Categorical Semantics, 2026. [Published paper](#).
2. Anonymous. *Higher Lenses and Temporal Descent for Certified Adaptive Systems*. Higher Categorical Semantics, 2026. [Published paper](#).
3. Anonymous. *LensIR: Compositional Reverse Semantics for a Typed Tensor Fragment*. Higher Categorical Semantics, 2026. [Published paper](#).
4. Anonymous. *Oracle and Jet Semantics of Stateful Optimization*. Higher Categorical Semantics, 2026. [Published paper](#).
5. Anonymous. *Observation-Preserving Runtime Gates and Checkpoint Semantics for Adaptive Learning*. Higher Categorical Semantics, 2026. [Published paper](#). The restricted control, codec, and gate results are used as precedent.
6. Jacob Lurie. *Higher Topos Theory*. Princeton University Press, 2009. Definition 6.1.0.4 and Section 6.1.3. [Author’s electronic edition](#).
7. Nima Rasekh. *A Theory of Elementary Higher Toposes*. 2018; revised 2022. [arXiv:1805.03805](#).
8. Xavier Leroy. *Formal verification of a realistic compiler*. Communications of the ACM 52(7), 107-115, 2009. [doi:10.1145/1538788.1538814](#); [author manuscript](#).
9. Patrick Schultz, David I. Spivak, and Christina Vasilakopoulou. *Dynamical Systems and Sheaves*. [arXiv:1609.08086](#).

10. Cloudflare. *SQLite-backed Durable Object Storage*. [Official API documentation](#). Accessed 12 October 2026.
11. Cloudflare. *Queues: Delivery guarantees*. [Official documentation](#). Accessed 12 October 2026.
12. Amazon Web Services. *Amazon DynamoDB Transactions: How it works*. [Official documentation](#). Accessed 12 October 2026.
13. Amazon Web Services. *Using Lambda with Amazon SQS*. [Official documentation](#). Accessed 12 October 2026.
14. Google Cloud. *Data contention in transactions: Serializable isolation*. [Official Firestore documentation](#). Accessed 12 October 2026.
15. Google Cloud. *Pub/Sub: Exactly-once delivery*. [Official documentation](#). Accessed 12 October 2026.
16. Cloudflare. *Workers AI overview*. [Official documentation](#). Accessed 12 October 2026.
17. Amazon Web Services. *Amazon Bedrock: Converse*. [Official API reference](#). Accessed 12 October 2026.
18. Google Cloud. *Models: generateContent*. [Official API reference](#). Accessed 12 October 2026; current redirect recorded in the source notes.
19. Cloudflare. *Workers: WebAssembly*. [Official runtime documentation](#). Accessed 12 October 2026.
20. Google Cloud. *What is Cloud Run?* [Official documentation](#). Accessed 12 October 2026.