Opus

Proof-Carrying Streamed Workflow Graphs

Phillip Kingston*
Member of Technical Staff
AppliedAI

Théo Fagnoni
Member of Technical Staff
AppliedAI

27 August 2026

Abstract

Incremental construction of directed workflow graphs can be formulated as a constrained graph-extension problem: each proposed addition should preserve the existence of a viable completion. In AI systems, an emitted addition can be syntactically correct even when its evidence is inadequate, its scope is invalid, or it makes the remaining policy obligations unsatisfiable. We introduce *proof-carrying streams*, in which every addition updates the committed graph and transforms a certificate witnessing that at least one acceptable completion remains.

We model each step in the graph construction by a state s, which records the current committed graph and all information needed to apply future additions and determine whether the construction remains viable. For an arbitrary state space S, an alphabet Σ of possible additions, a deterministic transition function $\delta : S \times \Sigma \to S$ that assigns a unique successor state to each state and addition, and a set F of acceptable states, we define the canonical *viability residual* $\mathcal{R}_s$. The viability residual $\mathcal{R}_s$ contains exactly the finite addition streams that can be admitted from s while leaving the resulting state viable, i.e., still able to reach an accepting state by some finite continuation. As additions are admitted, the viability residual evolves to describe exactly the futures that remain viable from the resulting state. A certificate is sound at a construction state when every future stream it represents belongs to that state's viability residual. Graph construction that begins with a non-empty sound certificate and proceeds through sound certificate updates preserves completion viability through every admitted addition.

We encode each construction state into a proof state that retains enough information to determine completion viability. Without loss of generality, we restrict every proof state to the image of this encoding, so every proof state represents at least one construction state. A proof-state abstraction consists of the encoding, an update rule that computes the next proof state from the current proof state and the next addition, and a classification of proof states as viable or nonviable. The abstraction is viability-exact when two conditions hold: updating an encoded proof state produces the same result as first applying the addition to the construction state and then encoding its successor, and the proof state's viability classification agrees exactly with the viability of every construction state it represents. These requirements imply that construction states encoded as the same proof state must have equal viability residuals. Consequently, grouping construction states by equality of their residuals forms the coarsest viability-exact abstraction: every viability-exact abstraction must keep construction states with different residuals separate. A finite viability-exact abstraction exists if and only if the construction states have finitely many distinct residuals.

Keywords: proof-carrying streams, streamed workflow graphs, completion viability, exact admission, residual languages, incremental certificates, formal verification.

* Corresponding author: phillip.kingston@opus.com

1 Introduction

Workflow graphs are increasingly assembled in pieces. An AI agent may propose a task and its dependencies; a service may add evidence, data mappings, or approval edges; a human-operated tool may close one branch and open another. Once admitted, such operations become part of the committed workflow. We call a workflow graph constructed through such a sequence of admitted atomic operations a *streamed workflow graph*. Thus, graph construction is an incremental admission problem rather than a one-shot design exercise.

Local legality is not enough. A proposed addition can be well typed, refer only to existing objects, and pass every local policy check, yet still eliminate the last globally acceptable way to finish the workflow. An approval route may be closed after all alternatives have already disappeared. A new dependency may create an obligation for which no compatible discharge remains. Evidence may be valid for its immediate subject while committing the graph to a branch whose final policy cannot be satisfied. The addition is locally coherent but globally fatal.

The natural admission condition is therefore existential completion viability: after admitting the addition, at least one policy-compliant completion must remain.
The challenge is to maintain this guarantee as the workflow is constructed incrementally. Carrying a proof of one fixed completion would preserve viability, but would commit the workflow to a particular future. Recomputing viability from scratch after every addition would avoid that commitment, but would discard the proof information established at previous steps. Instead, we let the unfinished workflow carry a certificate that evolves with it: each admitted addition is accompanied by a checked update of the current certificate, so evidence of remaining completion viability is maintained throughout construction.

The semantic object underlying this view is the *viability residual*. For a current state s, it records every finite addition stream that may be consumed next while still leaving some acceptable continuation. Let w be a finite stream of additions. Write $\delta^*(s, w)$ for the state reached from s by applying the additions in w in order. For a second finite addition stream u, write wu for their concatenation, with w consumed first and u second. Let L be a language, meaning a set of finite addition streams. The *left quotient* of L by w, also called the language derivative, is

$$w^{-1}L = \{u : wu \in L\}$$

It contains exactly those streams u for which consuming w followed by u produces a stream in L. If w is consumed, the new residual is not an unrelated semantic object; it is exactly $w^{-1}\mathcal{R}_s$. This yields the central picture:

$$(s, \mathcal{R}_s) \xrightarrow{w} (\delta^*(s, w),\ w^{-1}\mathcal{R}_s) \tag{1}$$

Construction moves forward, while the proof of remaining futures is transformed by consuming the same stream. We use *certificate* for a concrete carried representation and *viability proof state*, shortened below to *proof state*, for the semantic information that every viability-exact incremental abstraction must preserve.

The theory is representation-independent. It assumes only a state set, an addition alphabet, deterministic application of one fully specified addition, and an accepting set. No finiteness, acyclicity, graph schema, decision procedure, or complexity bound is required. A workflow graph is one interpretation of the state; the results apply equally to other incrementally constructed objects.

The main contributions are as follows.

(i) **Proof-carrying streams and prefix-wise viability.** We define proof-carrying streams, in which each admitted addition is accompanied by a checked certificate update, and prove that an initially sound, nonempty certificate together with stepwise-sound updates preserves completion viability at every admitted prefix (theorem 4.4).

(ii) **Viability residuals as the semantic basis for admission and update.** We define the viability residual $\mathcal{R}_s$ as the finite addition streams whose consumption leaves the resulting state viable. We prove its residual–successor equivalence, derive the canonical admission criterion, and show that it updates under consumed streams by left derivatives (theorems 3.2 and 3.4 and corollary 3.3).

(iii) **Minimality and universality of viability-exact proof states.** We prove that every viability-exact incremental proof-state abstraction must distinguish states with different residuals. We then construct the quotient by residual equivalence, prove that it is the coarsest viability-exact abstraction, establish the unique factorization of every viability-exact abstraction onto this quotient, and show that residual languages provide a canonically isomorphic representation (theorems 5.5 and 5.6 and proposition 5.7).

We instantiate these results for streamed workflow graphs and use a two-route example to distinguish local legality, successor viability, and residual completeness (corollary 6.1 and example 6.2).

Residual languages, left derivatives, and Nerode-style indistinguishability are classical ideas in automata theory. Proof-carrying code and proof-carrying plans likewise establish the producer/checker architecture. We do not claim these ingredients as new in isolation. The contribution is their proof-system synthesis for unfinished construction streams: the proposition such a stream carries, the canonical way that proposition changes under additions, and the distinctions that no viability-exact incremental proof-state abstraction may discard.

2 Stream semantics and completion viability

Fix a set S of construction states, an alphabet Σ of fully specified atomic construction operations, called additions, a deterministic transition function

$$\delta : S \times \Sigma \to S$$

and a set $F \subseteq S$ of accepting states. We write Σ^* for finite addition streams and interpret them as observed prefixes of a potentially ongoing construction process. We write ε for the empty stream and use juxtaposition for stream concatenation. We identify each $a \in \Sigma$ with its one-letter stream; hence aw denotes addition a followed by stream w. An addition may represent more than the insertion of a node or edge; it can also attach evidence, close a branch, discharge an obligation, or mark the workflow as complete.

The extension of δ to finite streams is the function $\delta^* : S \times \Sigma^* \to S$ defined by

$$\begin{aligned} \delta^*(s, \varepsilon) &= s \\ \delta^*(s, aw) &= \delta^*(\delta(s, a), w) \end{aligned} \tag{2}$$

Lemma 2.1 (Stream composition)**.** *For all $s \in S$ and $u, v \in \Sigma^*$,*

$$\delta^*(s, uv) = \delta^*(\delta^*(s, u), v) \tag{3}$$

Proof. We induct on u, after generalizing the starting state s. The base case when $u = \varepsilon$ is immediate. For $u = au'$, assume the claim holds for u' and every starting state. Then

$$\begin{aligned}\delta^*(s, (au')v) &= \delta^*(\delta(s, a), u'v) \\ &= \delta^*(\delta^*(\delta(s, a), u'), v) \\ &= \delta^*(\delta^*(s, au'), v)\end{aligned}$$

where the second equality is the induction hypothesis applied at $\delta(s, a)$. This proves the claim. □

The assumption that δ is total is only a notational convenience. If illegal additions are possible, we extend the state space with an absorbing dead state $\bot$. Every illegal application transitions to $\bot$, with $\delta(\bot, a) = \bot$ for all $a \in \Sigma$, and $\bot \notin F$. Under this totalization, we define

$$\mathsf{Legal}(s, a) \quad \iff \quad \delta(s, a) \neq \bot$$

Failed evidence checks, invalid scopes, and ill-typed graph edits can all be represented this way. Determinism means that a submitted addition contains enough information to determine its successor state. Nondeterministic effects or uncontrollable environment steps require a relational or game-based extension and are outside the present theorem.

Definition 2.2 (Completion viability)**.** A state s is *viable* when there exists a finite continuation that leads from s to an accepting state:

$$\mathsf{Viable}(s) \quad \iff \quad \exists v \in \Sigma^* : \ \delta^*(s, v) \in F \tag{4}$$

This is an existential finite-completion property. It says neither that every continuation succeeds nor that the actual stream must eventually terminate. It says only that the current construction has not exhausted all acceptable futures.

Section 6 specializes these parameters to workflow graphs. In that interpretation, the state must contain every fact that can affect future acceptance; the accepting set may encode evidence, role separation, resource constraints, terminal graph shape, or other final requirements.

3 Viability residuals and canonical admission

Definition 3.1 (Viability residual)**.** The *viability residual* of s is the language

$$\mathcal{R}_s = \{u \in \Sigma^* : \exists v \in \Sigma^* : \ \delta^*(s, uv) \in F\} \tag{5}$$

Under the dead-state totalization of section 2, $\mathcal{R}_\bot = \varnothing$, because every stream from $\bot$ remains at the nonaccepting dead state.

Call x a *prefix* of w when $w = xu$ for some stream u; a language is *prefix-closed* when it contains every prefix of each of its words. A stream u belongs to $\mathcal{R}_s$ exactly when it may be consumed next and some suffix still completes the construction. Equivalently, $\mathcal{R}_s$ is the prefix closure of the ordinary accepting continuation language

$$\{w \in \Sigma^* : \delta^*(s, w) \in F\}$$

Thus $\mathcal{R}_s$ contains the continuation prefixes that do not yet destroy completion viability; it is not a set of predetermined complete plans.

Theorem 3.2 (Residual–successor equivalence)**.** *For every $s \in S$ and $u \in \Sigma^*$,*

$$\boxed{u \in \mathcal{R}_s \quad \Longleftrightarrow \quad \mathsf{Viable}(\delta^*(s,u))} \tag{6}$$

Equivalently, the residual consists exactly of the streams whose consumption leaves a viable state. In particular, since ε denotes the empty stream,

$$\mathsf{Viable}(s) \Longleftrightarrow \varepsilon \in \mathcal{R}_s \tag{7}$$

Proof. By definition, $u \in \mathcal{R}_s$ if and only if there exists v with $\delta^*(s,uv) \in F$. By lemma 2.1, this is equivalent to the existence of v with $\delta^*(\delta^*(s,u),v) \in F$, which is precisely viability of $\delta^*(s,u)$. Taking $u = \varepsilon$ gives (7). □

Corollary 3.3 (Canonical admission criterion)**.** *A proposed addition a preserves completion viability exactly when its one-letter stream belongs to the current residual:*

$$\boxed{a \in \mathcal{R}_s \quad \Longleftrightarrow \quad \mathsf{Viable}(\delta(s,a))} \tag{8}$$

Proof. For the one-letter stream a, we have $\delta^*(s,a) = \delta(s,a)$. The claim then follows from theorem 3.2. □

The criterion is semantically exact: it has neither false admissions nor false rejections relative to successor viability. This is not a decidability claim; residual membership may be undecidable.

The predicate $\mathsf{Legal}(s,a)$ determines whether the addition has a non-dead successor; residual membership determines whether that successor retains an acceptable future. Since $\mathcal{R}_\perp = \varnothing$, an illegal addition is never canonically admitted.

For a language $L \subseteq \Sigma^*$ and stream $w \in \Sigma^*$, define its left quotient, also called the language derivative, by

$$w^{-1}L = \{u \in \Sigma^* : wu \in L\} \tag{9}$$

Theorem 3.4 (Derivative law)**.** *For all $s \in S$ and $w \in \Sigma^*$,*

$$\boxed{\mathcal{R}_{\delta^*(s,w)} = w^{-1}\mathcal{R}_s} \tag{10}$$

Equivalently, for every $u \in \Sigma^*$,

$$u \in \mathcal{R}_{\delta^*(s,w)} \quad \Longleftrightarrow \quad wu \in \mathcal{R}_s \tag{11}$$

Proof. Using definition 3.1 and lemma 2.1,

$$\begin{aligned} u \in \mathcal{R}_{\delta^*(s,w)} &\Longleftrightarrow \exists v :\ \delta^*(\delta^*(s,w),uv) \in F \\ &\Longleftrightarrow \exists v :\ \delta^*(s,wuv) \in F \\ &\Longleftrightarrow wu \in \mathcal{R}_s \end{aligned}$$

This is exactly the definition of $w^{-1}\mathcal{R}_s$. □

The derivative law is the semantic update rule for proof-carrying streams. Once w is consumed, the derivative removes that consumed prefix and exposes exactly the futures available from the successor state. Left derivatives compose in the expected order:

$$(uw)^{-1}L = w^{-1}(u^{-1}L)$$

Thus stepwise proof updates and one update by the whole consumed stream agree.

Proposition 3.5 (Prefix closure)**.** *Each $\mathcal{R}_s$ is prefix-closed. If $w \in \mathcal{R}_s$ and $w = xu$, then $x \in \mathcal{R}_s$. Consequently,*

$$w \in \mathcal{R}_s \Longrightarrow \mathsf{Viable}(\delta^*(s, x)) \quad \textit{for every prefix } x \textit{ of } w \tag{12}$$

Proof. Choose v with $\delta^*(s, wv) \in F$. If $w = xu$, then $\delta^*(s, x(uv)) \in F$, so $x \in \mathcal{R}_s$. Equation (12) follows from theorem 3.2. □

Corollary 3.6 (Residual nonemptiness)**.** *For every state s,*

$$\mathcal{R}_s \neq \varnothing \quad \Longleftrightarrow \quad \mathsf{Viable}(s) \tag{13}$$

Proof. If R_s is nonempty, choose $w \in R_s$. Since ε is a prefix of w, Proposition 3.5 gives $\varepsilon \in R_s$, hence s is viable by Theorem 3.2. Conversely, if s is viable, then $\varepsilon \in R_s$ by Theorem 3.2, so $R_s \neq \varnothing$. □

Residual membership is defined by the existence of a completion suffix, yet it guarantees viability at every state encountered while consuming that prefix.

4 Proof-carrying streams

4.1 Certificates and stepwise checking

The viability residual is a semantic object and may be infinite or undecidable. A concrete implementation instead carries a certificate that denotes some language of future streams.

Definition 4.1 (Certificate semantics)**.** A certificate semantics consists of a set C and a denotation map

$$[\![\cdot]\!] : C \to 2^{\Sigma^*}$$

A certificate c is *sound at s* when

$$[\![c]\!] \subseteq \mathcal{R}_s \tag{14}$$

and it is *nonempty* when

$$[\![c]\!] \neq \varnothing \tag{15}$$

If c is sound at s and nonempty, then $\mathcal{R}_s$ is nonempty and therefore s is viable by corollary 3.6. For an implementation-dependent checking witness η, let $\mathsf{Check}(c, a, c', \eta)$ be the trusted judgment that the checker validates η as a witness that addition a transforms certificate c into c'.

Definition 4.2 (Stepwise-sound certificate update)**.** The checker is *stepwise sound* when every validated judgment satisfies

$$\mathsf{Check}(c, a, c', \eta) \Longrightarrow \Big([\![c']\!] \subseteq a^{-1} [\![c]\!] \ \wedge \ [\![c']\!] \neq \varnothing \Big) \tag{16}$$

Here *stepwise* means one consumed addition at a time; it does not imply that checking is local to the changed subgraph.

The first clause is refinement: after consuming a, every future claimed by c' was already supported by c. The second says that the successor certificate retains at least one certified future. The producer may use any method to construct c' and η; only the checker is trusted. We represent a checked transition of the construction state and its certificate by

$$(s, c) \xrightarrow{a,\eta} (\delta(s, a), c')$$

whenever $\mathsf{Check}(c, a, c', \eta)$ holds.

Definition 4.3 (Proof-carrying stream)**.** A proof-carrying stream is any finite chain of checked steps

$$(s_0, c_0) \xrightarrow{a_0,\eta_0} (s_1, c_1) \xrightarrow{a_1,\eta_1} \cdots \xrightarrow{a_{n-1},\eta_{n-1}} (s_n, c_n)$$

4.2 Stepwise checking gives a global invariant

Theorem 4.4 (Proof-carrying stream soundness)**.** *Assume the checker is stepwise sound. Let*

$$(s_i, c_i) \xrightarrow{a_i,\eta_i} (s_{i+1}, c_{i+1}), \qquad 0 \le i < n \tag{17}$$

be the checked steps of a proof-carrying stream. If c_0 is sound at s_0 and nonempty, then for every $i \le n$,

$$[\![c_i]\!] \subseteq \mathcal{R}_{s_i}, \qquad [\![c_i]\!] \neq \varnothing, \qquad \mathsf{Viable}(s_i) \tag{18}$$

Proof. Induct on i. The base assumptions give $[\![c_0]\!] \subseteq \mathcal{R}_{s_0}$ and $[\![c_0]\!] \neq \varnothing$, hence $\mathsf{Viable}(s_0)$ by corollary 3.6. For the step, stepwise soundness gives

$$[\![c_{i+1}]\!] \subseteq a_i^{-1} [\![c_i]\!] \subseteq a_i^{-1} \mathcal{R}_{s_i} = \mathcal{R}_{s_{i+1}}$$

where the second inclusion is monotonicity of left derivatives and the equality is theorem 3.4. The checked step also makes $[\![c_{i+1}]\!]$ nonempty, so corollary 3.6 yields $\mathsf{Viable}(s_{i+1})$. □

Writing $w_0 = \varepsilon$ and $w_i = a_0 \cdots a_{i-1}$ for $1 \le i \le n$, repeated refinement also gives

$$[\![c_i]\!] \subseteq w_i^{-1} [\![c_0]\!] \qquad (0 \le i \le n) \tag{19}$$

Thus the final certificate satisfies the semantic refinement of the initial certificate through the recorded sequence of admitted additions. Each certificate keeps only futures supported by the initial certificate after the admitted additions. A future omitted initially or removed by an update cannot be added back through these certificate updates.

Definition 4.5 (Residual completeness and derivative exactness)**.** A certificate update is *derivative-exact* if the updated certificate represents exactly the futures remaining after the addition a is consumed. That is,

$$[\![c']\!] = a^{-1} [\![c]\!]$$

A certificate c is *residually complete at s* when $[\![c]\!] = \mathcal{R}_s$. Thus, unlike a merely sound update, a derivative-exact update does not discard any futures represented by the previous certificate that remain after a is consumed.

If c_0 is residually complete at s_0 and every checked update is derivative-exact, induction with theorem 3.4 gives $[\![c_i]\!] = \mathcal{R}_{s_i}$ along every checked chain. Hence each successor certificate in the chain still denotes the residual used in corollary 3.3. This does not guarantee that such a successor exists or can be validated for every viable addition. More generally, a sound, nonempty certificate c at s may be a proper under-approximation, $[\![c]\!] \subsetneq \mathcal{R}_s$. Under a stepwise-sound checker, theorem 4.4 still guarantees viability along admitted prefixes, although viable additions may be rejected when their surviving futures are omitted.

5 The coarsest viability-exact proof-state abstraction

The preceding section gives denotational conditions for checking a certificate lineage. We now ask a representation-independent question: which distinctions must any viability-exact incrementally updated proof state retain?

Definition 5.1 (Viability-exact proof-state abstraction)**.** An incremental proof-state abstraction is a tuple (P, E, U, Live) consisting of a proof-state set P, a surjective encoding $E : S \to P$, meaning that every proof state represents at least one construction state, an update function $U : P \times \Sigma \to P$, and a set of live proof states $\mathsf{Live} \subseteq P$. It is *viability-exact* when the following conditions hold:

$$E(\delta(s, a)) = U(E(s), a) \qquad \forall s \in S,\ a \in \Sigma \tag{20}$$

$$E(s) \in \mathsf{Live} \iff \mathsf{Viable}(s) \qquad \forall s \in S \tag{21}$$

For viability-exact abstractions, surjectivity loses no generality. Starting from an arbitrary encoding, restrict the proof-state set to $E(S)$ and the live set to $\mathsf{Live} \cap E(S)$. Equation (20) gives $U(E(s), a) = E(\delta(s, a)) \in E(S)$, so this image is closed under updates. Equation (20) says that encoding commutes with addition application, while (21) says that live proof states have neither false positives nor false negatives for viability.

Extend U to streams by

$$\begin{aligned} U^*(p, \varepsilon) &= p \\ U^*(p, aw) &= U^*(U(p, a), w) \end{aligned} \tag{22}$$

Lemma 5.2 (Proof-state updates compose)**.** *If* (20) *holds, then for every* $s \in S$ *and* $w \in \Sigma^*$*,*

$$E(\delta^*(s, w)) = U^*(E(s), w) \tag{23}$$

Proof. Induct on w after generalizing the starting state. The empty case is immediate. For $w = au$, use (20) for the first addition and apply the induction hypothesis at $\delta(s, a)$. □

Definition 5.3 (Residual equivalence)**.** States $s, t \in S$ are *residual equivalent*, written $s \equiv_{\mathcal{R}} t$, when

$$\mathcal{R}_s = \mathcal{R}_t \tag{24}$$

Equivalently, by theorem 3.2,

$$\mathsf{Viable}(\delta^*(s, w)) \iff \mathsf{Viable}(\delta^*(t, w)) \qquad \text{for every } w \in \Sigma^*$$

Thus residual-equivalent states are indistinguishable by every future viability test. Since the relation is defined by equality of residual languages, $\equiv_{\mathcal{R}}$ is an equivalence relation.

Lemma 5.4 (Addition congruence)**.** *If $s \equiv_{\mathcal{R}} t$, then $\delta(s,a) \equiv_{\mathcal{R}} \delta(t,a)$ for every $a \in \Sigma$.*

Proof. By theorem 3.4, $\mathcal{R}_{\delta(s,a)} = a^{-1}\mathcal{R}_s$ and $\mathcal{R}_{\delta(t,a)} = a^{-1}\mathcal{R}_t$. Equal languages have equal derivatives. □

Theorem 5.5 (Residual separation)**.** *Let P be any set, let $E : S \to P$ and $U : P \times \Sigma \to P$, and let $\mathsf{Live} \subseteq P$. If (20) and (21) hold, then*

$$\boxed{E(s) = E(t) \quad \Longrightarrow \quad s \equiv_{\mathcal{R}} t} \tag{25}$$

Surjectivity is not required. Equivalently, if some future stream distinguishes the viability of s and t, then any (P, E, U, Live) satisfying (20) and (21) must assign them different proof states.

Proof. Assume $E(s) = E(t)$ and fix $w \in \Sigma^*$. By lemma 5.2,

$$\begin{aligned} E(\delta^*(s,w)) &= U^*(E(s), w) \\ &= U^*(E(t), w) \\ &= E(\delta^*(t,w)) \end{aligned}$$

The equal encoded states have the same membership in Live; using (21) gives

$$\mathsf{Viable}(\delta^*(s,w)) \quad \Longleftrightarrow \quad \mathsf{Viable}(\delta^*(t,w))$$

Since w was arbitrary, $s \equiv_{\mathcal{R}} t$. □

This is an information lower bound, not a storage or computational complexity bound. It does not say that a proof state must literally store a language. It says that no viability-exact abstraction may erase a distinction that can change the correct viability answer after a common future stream.

Theorem 5.5 therefore suggests identifying exactly those states that a viability-exact abstraction may safely merge. Accordingly, define the quotient

$$Q = S/\equiv_{\mathcal{R}}$$

whose elements are the residual-equivalence classes $[s]$. By lemma 5.4, the update

$$\bar{\delta}([s], a) = [\delta(s,a)] \tag{26}$$

is well defined. Define the live quotient classes $\mathsf{Live}_Q \subseteq Q$ by

$$[s] \in \mathsf{Live}_Q \quad \Longleftrightarrow \quad \varepsilon \in \mathcal{R}_s \tag{27}$$

This is well defined because equivalent states have equal residuals.

Theorem 5.6 (Coarsest viability-exact proof-state abstraction)**.** *The quotient with encoding $s \mapsto [s]$, update $\bar{\delta}$, and live classes Live_Q is a viability-exact incremental proof-state abstraction. Moreover, for every viability-exact abstraction (P, E, U, Live) there is a unique map $\phi : P \to Q$, necessarily surjective. Its first identity below is defining; update and live-set preservation are consequences.*

$$\phi(E(s)) = [s] \qquad \forall s \in S, \tag{28}$$

$$\phi(U(p,a)) = \bar{\delta}(\phi(p), a) \qquad \forall p \in P,\ a \in \Sigma, \tag{29}$$

$$p \in \mathsf{Live} \Longleftrightarrow \phi(p) \in \mathsf{Live}_Q \qquad \forall p \in P \tag{30}$$

Consequently, the partition induced by every viability-exact encoding refines residual equivalence, and Q is the coarsest viability-exact quotient.

Proof. The quotient update is well defined by lemma 5.4, and the quotient live set is well defined by residual equality. Equation (26) gives the required update equation. By (27) and theorem 3.2,

$$[s] \in \mathsf{Live}_Q \quad \Longleftrightarrow \quad \varepsilon \in \mathcal{R}_s \quad \Longleftrightarrow \quad \mathsf{Viable}(s)$$

so the quotient is viability-exact.

Now let (P, E, U, Live) be viability-exact. Define $\phi(E(s)) = [s]$. If $E(s) = E(t)$, theorem 5.5 gives $s \equiv_{\mathcal{R}} t$, so $[s] = [t]$ and ϕ is well defined. It is surjective because $\phi(E(s)) = [s]$ for every residual class. Given $p \in P$, choose s with $p = E(s)$. Then

$$\begin{aligned} \phi(U(p, a)) &= \phi(E(\delta(s, a))) \\ &= [\delta(s, a)] \\ &= \bar{\delta}([s], a) \\ &= \bar{\delta}(\phi(p), a) \end{aligned}$$

which proves (29). The viability-exact live-state condition and theorem 3.2 give (30). Surjectivity of E makes (28) determine ϕ on all of P, proving uniqueness. □

This factorization makes Q coarsest: an abstraction may retain extra distinctions, but it cannot merge distinct residual classes.

Proposition 5.7 (Residual-language abstraction)**.** *Let*

$$P_{\mathcal{R}} = \{\mathcal{R}_s : s \in S\}, \qquad E_{\mathcal{R}}(s) = \mathcal{R}_s$$

and, for $L \in P_{\mathcal{R}}$, define

$$U_{\mathcal{R}}(L, a) = a^{-1}L, \qquad \mathsf{Live}_{\mathcal{R}} = \{L \in P_{\mathcal{R}} : \varepsilon \in L\}$$

Then $(P_{\mathcal{R}}, E_{\mathcal{R}}, U_{\mathcal{R}}, \mathsf{Live}_{\mathcal{R}})$ is viability-exact, and its factor map to Q is a bijection.

Proof. For $L = \mathcal{R}_s$, theorem 3.4 gives $a^{-1}L = \mathcal{R}_{\delta(s,a)} \in P_{\mathcal{R}}$, so $U_{\mathcal{R}}(L, a) = a^{-1}L$ is again an element of $P_{\mathcal{R}}$ and is defined directly from L, without choosing a representative state. The encoding $E_{\mathcal{R}}$ is surjective by definition; the derivative and residual–successor theorems give exact update and viability classification. Finally, $\mathcal{R}_s \mapsto [s]$ is well defined and bijective because residual equality is exactly residual equivalence. □

Thus, S is the concrete state space, Q is its minimal viability-exact quotient, and $P_{\mathcal{R}}$ is the same quotient represented canonically by residual languages.

Sections 4 and 5 give different guarantees. A nonempty sound certificate may be a proper under-approximation of the residual and therefore preserve viability while rejecting some viable additions. Residually complete certificates with derivative-exact checked updates track the residual along the checked chain; a viability-exact state-indexed abstraction must preserve all distinctions between different residual classes.

Corollary 5.8 (Finite viability-exact proof-state abstractions)**.** *A finite viability-exact incremental proof-state abstraction exists if and only if the set of distinct residuals $\mathcal{R}_s$ is finite. When finite, $|Q|$ is the minimum possible value of $|P|$ among viability-exact abstractions.*

Proof. If the set of distinct residuals $\{\mathcal{R}_s : s \in S\}$ is finite, then the residual quotient $Q = S/\equiv_{\mathcal{R}}$ is finite. By Theorem 5.6, Q is a viability-exact proof-state abstraction, so a finite viability-exact abstraction exists.

Conversely, suppose (P, E, U, Live) is a finite viability-exact abstraction. By Theorem 5.6, there is a surjective factor map $\phi : P \to Q$. Hence Q is finite, and therefore the set of distinct residuals is finite.

Moreover, for every viability-exact abstraction P, the surjectivity of $\phi : P \to Q$ gives

$$|Q| \leq |P|$$

Since the quotient abstraction itself has proof-state set Q, this bound is attained. Thus $|Q|$ is the minimum possible size of a viability-exact proof-state abstraction. □

The corollary does not assert that the quotient can be computed, nor that a finite abstraction is small. It characterizes exactly when finite viability-exact representation is possible, independently of computational considerations.

6 Specialization to streamed workflow graphs

The abstract model specializes to workflow graphs by taking S as partial directed workflow graphs G containing all information relevant to final acceptance, Σ as fully specified atomic additions, δ as atomic application with invalid additions sent to the dead state, and F as the completed graphs satisfying the final-graph acceptance predicate. With this instantiation, the abstract results apply without further graph-specific assumptions.

Corollary 6.1 (Workflow-prefix viability)**.** *Under this instantiation, assuming each graph state contains all information relevant to final acceptance, let* $(G_0, c_0) \xrightarrow{a_0,\eta_0} \cdots \xrightarrow{a_{n-1},\eta_{n-1}} (G_n, c_n)$ *be a proof-carrying stream under a stepwise-sound checker, with* c_0 *sound at* G_0 *and nonempty. Then every admitted partial workflow graph* G_i *has a finite completion satisfying the final-graph acceptance predicate.*

Proof. Immediate from theorem 4.4, since F is the set of completed graphs satisfying that predicate. □

Example 6.2 (Two policy-compliant routes)**.** Consider a release node that may be completed through route A or route B. Each route requires its own evidence edge and final approval. At state G, both routes remain possible, so $\mathcal{R}_G$ contains continuation prefixes compatible with either route.

Suppose an addition a permanently closes route A while leaving route B available. Then

$$\mathcal{R}_{\delta(G,a)} = a^{-1}\mathcal{R}_G$$

contains only the surviving B-compatible futures. The addition remains viable, but a certificate denoting only the incompatible A-route future has empty derivative under a and cannot yield a nonempty checked successor; route B nevertheless survives. The canonical admission criterion admits a because $\varepsilon \in a^{-1}\mathcal{R}_G$. A certificate can track this derivative if it can represent all the remaining futures and the checker can validate the update.

If route B had already been made unavailable, the same locally legal addition would yield a derivative that omits ε, so the canonical admission criterion rejects it. The difference is not visible in the syntax of a alone; it lies in the viability residual of the current state.

This example separates three notions that are often conflated:

(a) *local legality*: whether $\mathsf{Legal}(G, a)$ holds;
(b) *successor viability*: whether the resulting state has at least one acceptable completion; and
(c) *residual completeness*: whether the certificate denotes the full viability residual rather than a sound under-approximation.

Possible certificate representations include a symbolic automaton, a satisfiability-modulo-theories (SMT) formula with a satisfying assignment, a proof term with open obligations, or a family of candidate plans. The theory is neutral among them. A certificate denoting the full viability residual tracks that residual under derivative-exact checked updates. Proper under-approximations remain sound but may reject viable additions.

The graph-extension motivation also explains the phrase *proof-carrying*. The proof is not merely stored beside the graph; it has a lineage. Each admitted addition comes with evidence that the next certificate is a valid derivative refinement of the preceding certificate. An auditor can therefore check the certificate lineage and, when the workflow closes, the final graph.

7 Lean formalization

The accompanying Lean 4 development, supplied as `code/ProofCarryingStreams.lean`, is written for Lean 4 [1]. It is parameterized only by arbitrary types `State` and `Action`, a transition function `step : State → Action → State`, and an acceptance predicate `accept : State → Prop`. Here `Action` corresponds to an addition, and `run` is the list extension of `step`, corresponding to δ^* in (2). The source imports only Lean's `Init` library; its central definitions mirror (4) and (5):

```
def Viable (step : State → Action → State)
    (accept : State → Prop) (s : State) : Prop :=
  ∃ w : List Action, accept (run step s w)

def Residual (step : State → Action → State)
    (accept : State → Prop) (s : State)
    (u : List Action) : Prop :=
  ∃ v : List Action, accept (run step s (u ++ v))
```

Paper result	**Lean declaration**
Stream composition	`run_append`
Residual–successor equivalence	`residual_iff_viable_after_run`
Derivative update	`residual_after_run_iff`
Prefix closure and prefix viability	`residual_prefix_closed`, `viable_at_every_prefix`
Proof-carrying stream soundness	`proof_carrying_stream_soundness`
Residual-equivalence congruence	`residualEq_step`
Stream lifting of proof-state updates	`encode_run_eq_updateMany`
Residual separation theorem	`exact_proof_state_preserves_ residual_distinctions`
Residual-language update and viability classification	`canonical_abstraction_is_exact`

Table 1. Correspondence between paper statements and the Lean development.

Table 1 gives the correspondence. The canonical Lean result states residual update pointwise and exact viability classification. The quotient construction, universal factorization, quotient bijection, finite-state minimality result, and arbitrary prefix-closed-language realization are proved in the

paper and are not formalized in the accompanying source. The development does not verify that a production workflow engine implements the selected `State`, `Action`, `step`, and `accept`; that remains a model-validation obligation.

8 Relation to prior work

Proof-carrying systems. Proof-carrying code places the burden of constructing a safety proof on an untrusted producer and leaves a comparatively small checker in the trusted base [2]. Foundational and modular variants extend that architecture to richer program logics and separately certified modules [3]. Proof-carrying hardware applies the same producer/checker asymmetry to dynamically loaded circuits [4]. Our checker architecture follows this lineage, but the certified object is an unfinished construction and the certified proposition changes after every streamed addition. Abstraction-Carrying Code uses an abstract model as the carried certificate and validates it with a specialized abstract interpreter [5]; it targets a fixed program and safety policy rather than an evolving completion-viability residual. Si and Zhang make source-to-source rewrites certificate carrying: an untrusted optimizer proposes a rewrite and a fail-closed checker recomputes the semantic side conditions, with the central refinement theorem mechanized in Lean [6]. They certify behavior preservation of individual transformations, whereas our result preserves a nonempty acceptable-future space across a stream of additions.

Proof-carrying plans certify complete plans against resource-aware planning semantics and provide a machine-checked soundness development [7]. A proof-carrying stream instead avoids choosing a complete plan: its certificate denotes continuation prefixes that retain at least one completion, and an addition transforms that certificate by a derivative. The prefixes of a single certified plan therefore form a sound certificate, but generally not a residually complete certificate.

Residual languages and minimality. Language derivatives provide the update $w^{-1}L$ after consuming a prefix [8]. Nerode equivalence identifies histories that no continuation can distinguish and underlies minimal deterministic realization [9]. Our coarsest-abstraction theorem is a Nerode-style result for the viability predicate under common future addition streams. Its role here is proof-theoretic: it characterizes the proof-state distinctions required by viability-exact incremental checking and states the corresponding factorization directly in proof-state terms.

Workflow soundness and supervisory control. Workflow-net soundness asks whether cases can complete correctly in a fixed workflow model [10]; here the model itself is assembled while every admitted prefix retains a completion. In supervisory-control terminology, F is the marked set and $\mathsf{Viable}(s)$ is marker-coreachability, whereas nonblockingness requires every relevant reachable state to be marker-coreachable. The accepting continuation language from s, called the marked continuation language in that terminology, is $\{w : \delta^*(s, w) \in F\}$, and $\mathcal{R}_s$ is its prefix closure. Residual equivalence is therefore Nerode equivalence for future viability and may be strictly coarser than marked-language equivalence because prefix closure can merge distinct marked languages. Supervisory control synthesizes behavior under controllability constraints [11, 12]; here proposed additions are controllable and the object of study is the proof state accompanying admission.

Runtime enforcement and streamed AI actions. Runtime enforcement checks or edits an execution against a policy [13–15]. Proof-carrying streams can be used at such a boundary, but the policy studied here is specifically preservation of a nonempty acceptable-completion space. Recent agent-runtime work emphasizes action identity, commit-time checking, and durable authority or attestation attached to agent actions [16, 17]. Those systems-level concerns are complementary: once a committed action is represented as a deterministic addition, the residual theorem describes the corresponding future-viability proof state.

The contribution is therefore not new residual or minimization machinery, but its use as a semantics for unfinished proof-carrying construction streams, together with the stepwise certificate-lineage theorem, the residual separation and factorization results, and the workflow-graph interpretation.

9 Scope and limits of generality

The sparse assumptions give broad applicability, but the resulting claims are semantic rather than algorithmic.

First, viability is existential over finite streams. The theory guarantees that some acceptable continuation remains after every certified prefix. It does not guarantee that every continuation is acceptable, that an adversarial scheduler cannot avoid completion, or that an infinite stream eventually terminates.

Second, additions are deterministic after they are fully specified. If an admitted addition has several possible effects on the committed state, or if uncontrollable environment steps can intervene, the correct abstraction is a transition relation or game. The viability semantics would then require the appropriate combination of existential and universal choices rather than the plain residual language used here.

Third, viability-exactness is semantic, not algorithmic. The canonical residual may be infinite, undecidable, or impossible to represent compactly. Corollary 5.8 characterizes when a finite viability-exact abstraction exists; it does not provide an algorithm for finding one.

Fourth, no local graph-update theorem follows from the bare model. A derivative-exact certificate update may require global knowledge of the graph and policy. Locality, compositionality, bounded proofs, or efficient checking require additional assumptions about how acceptance decomposes over graph structure.

The generality boundary can be made explicit.

Proposition 9.1 (Arbitrary prefix-closed residuals)**.** *For every prefix-closed language $L \subseteq \Sigma^*$, there are choices of S, δ, F, and an initial state s_0 such that*

$$\mathcal{R}_{s_0} = L$$

Proof. Take $S = \Sigma^*$, let $\delta(w, a) = wa$, set $s_0 = \varepsilon$, and take $F = L$. A one-line induction on x gives $\delta^*(w, x) = wx$. Hence

$$u \in \mathcal{R}_{s_0} \iff \exists v :\ uv \in L$$

If $uv \in L$, prefix closure gives $u \in L$; if $u \in L$, choose $v = \varepsilon$. Hence $\mathcal{R}_{s_0} = L$. □

Corollary 9.2 (Residual-language characterization)**.** *A language over Σ arises as $\mathcal{R}_s$ for some deterministic addition system and state s if and only if it is prefix-closed.*

Proof. Every viability residual is prefix-closed by proposition 3.5, and every prefix-closed language is realized by proposition 9.1. □

Arbitrary prefix-closed residual languages therefore arise under the minimal assumptions; no universal result on finiteness, decidability, finite or succinct representability, or graph locality follows without further structure. The residual and quotient remain the semantic targets.

10 Conclusion

A streamed workflow addition should not be admitted merely because it is locally legal. It should be admitted only with evidence that the successor still admits a globally policy-compliant completion. Proof-carrying streams make that evidence persistent: the graph and its certificate advance together.

The canonical proof state is the viability residual of future addition streams that preserve completion viability. Its update after a consumed stream is exactly a left derivative. This identity turns stepwise-checked certificate refinements into a global invariant over every admitted prefix. It also yields a separation theorem: any viability-exact incremental proof-state abstraction must preserve every residual distinction that can affect a later viability answer. The quotient by those distinctions is the coarsest viability-exact proof-state abstraction, and every viability-exact proof-state space has a canonical surjective factor map onto it.

The result does not prescribe a certificate format or promise efficient computation. Starting from a sound, nonempty certificate, stepwise-sound checked updates preserve viability, although under-approximations may reject viable additions. Within the viability-exact incremental abstraction model, construction states with different viability residuals must be assigned different proof states.

References

[1] de Moura, L. and Ullrich, S. (2021). The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, LNCS 12699, 625–635. Springer. doi:10.1007/978-3-030-79876-5_37.

[2] Necula, G. C. (1997). Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 106–119. ACM. doi:10.1145/263699.263712.

[3] Feng, X., Ni, Z., Shao, Z., and Guo, Y. (2007). An Open Framework for Foundational Proof-Carrying Code. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 67–78. ACM. doi:10.1145/1190315.1190325.

[4] Drzevitzky, S., Kastens, U., and Platzner, M. (2010). Proof-Carrying Hardware: Concept and Prototype Tool Flow for Online Verification. *International Journal of Reconfigurable Computing*, 2010, Article 180242. doi:10.1155/2010/180242.

[5] Albert, E., Puebla, G., and Hermenegildo, M. V. (2005). Abstraction-Carrying Code. In *Logic for Programming, Artificial Intelligence, and Reasoning*, LNAI 3452, 380–397. Springer. doi:10.1007/978-3-540-32275-7_25.

[6] Si, Y. and Zhang, J. (2026). Certificate-Carrying Transformation of Event-Driven Block Programs. arXiv:2607.00563. doi:10.48550/arXiv.2607.00563.

[7] Hill, A., Komendantskaya, E., and Petrick, R. P. A. (2020). Proof-Carrying Plans: A Resource Logic for AI Planning. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming*, Article 14. ACM. doi:10.1145/3414080.3414094.

[8] Brzozowski, J. A. (1964). Derivatives of Regular Expressions. *Journal of the ACM*, 11(4), 481–494. doi:10.1145/321239.321249.

[9] Nerode, A. (1958). Linear Automaton Transformations. *Proceedings of the American Mathematical Society*, 9(4), 541–544. doi:10.1090/S0002-9939-1958-0135681-9.

[10] van der Aalst, W. M. P., van Hee, K. M., ter Hofstede, A. H. M., Sidorova, N., Verbeek, H. M. W., Voorhoeve, M., and Wynn, M. T. (2011). Soundness of workflow nets: classification, decidability, and analysis. *Formal Aspects of Computing*, 23, 333–363. doi:10.1007/s00165-010-0161-4.

[11] Ramadge, P. J. and Wonham, W. M. (1987). Supervisory Control of a Class of Discrete Event Processes. *SIAM Journal on Control and Optimization*, 25(1), 206–230. doi:10.1137/0325013.

[12] Wonham, W. M. and Ramadge, P. J. (1987). On the Supremal Controllable Sublanguage of a Given Language. *SIAM Journal on Control and Optimization*, 25(3), 637–659. doi:10.1137/0325036.

[13] Schneider, F. B. (2000). Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1), 30–50. doi:10.1145/353323.353382.

[14] Ligatti, J., Bauer, L., and Walker, D. (2005). Edit Automata: Enforcement Mechanisms for Run-Time Security Policies. *International Journal of Information Security*, 4(1–2), 2–16. doi:10.1007/s10207-004-0046-8.

[15] Pinisetty, S., Preoteasa, V., Tripakis, S., Jéron, T., Falcone, Y., and Marchand, H. (2017). Predictive Runtime Enforcement. *Formal Methods in System Design*, 51(1), 154–199. doi:10.1007/s10703-017-0271-1.

[16] Lilienthal, D. and Hong, S. (2025). Mind the Gap: Time-of-Check to Time-of-Use Vulnerabilities in LLM-Enabled Agents. arXiv:2508.17155. doi:10.48550/arXiv.2508.17155.

[17] Santos-Grueiro, I. (2026). Temporary Authority, Permanent Effects: Commit-Time Authorization for LLM Agents. arXiv:2607.10487. doi:10.48550/arXiv.2607.10487.