Selective Continuation in Three Small Programs: Proposal, Record, Admission, and History*

Flyxion

2026

Abstract

Software does not merely determine what happens; it determines which proposed happenings are permitted to become operative state. This article develops that proposition through three deliberately small command-line programs. `hello-tool`, written in Go, governs admissible execution through subtractive security and a release procedure that halts before remote publication. `reality-permit`, written in Perl, governs institutional judgment while demonstrating that reproducibility does not imply legitimacy. `counterfactual-git`, written in Rust, governs historical continuation: every proposal is recorded, but only an admissible proposal advances the operative head. Together the artifacts instantiate a progression from execution, through judgment, to selective history. Their shared architecture distinguishes proposal, record, admission, and continuation, allowing history to remain append-only while operative reality remains selective.

1 Introduction

Small programs are often described as exercises, demonstrations, or scaffolds awaiting "real" functionality. This description misses their theoretical density. A compact program can expose its assumptions more clearly than a large application because the relation among state, transition, policy, memory, and output remains inspectable. The three programs considered here exploit that property. Each is runnable as a terminal artifact, each minimizes external dependencies, and each organizes a different relation between a proposed action and an authorized result.

The sequence begins conventionally. `hello-tool` is a secure starter for a Go command-line application. It greets a named user, reports its version, rejects malformed invocation, and supplies a management script for building, testing, checking, and tagging releases. The second program, `reality-permit`, replaces ordinary utility with comic administration. Written in Perl, it computes whether a procedurally generated universe is approved or denied by a fictional Ministry of Possible Affairs. The third, `counterfactual-git`, is a Rust program that treats impossible historical events as commits to an in-memory universe. It distinguishes between events preserved in the audit record and events admitted to the current timeline.

*The programs discussed are `hello-tool` (Go), `reality-permit` (Perl), and `counterfactual-git` (Rust), all written in 2026.

The three artifacts instantiate progressively richer constraint regimes. In the Go program, validity is operational: only well-formed invocations and verified releases proceed. In the Perl program, judgment is institutional and satirical: numerical measures are converted into official rulings. In the Rust program, admissibility is historical: an event may be recorded without being allowed to transform the active state. The movement across languages is therefore not merely stylistic. It reveals three layers of software authority—execution, judgment, and historical continuation—formalized below as V, J, and A.

The argument depends upon refusing several common equivalences:

$$\text{correct execution} \neq \text{correct model} \neq \text{legitimate rule} \neq \text{authorized action}. \tag{1}$$

A program can execute a reproducible rule correctly while the model remains defective, the rule illegitimate, or the resulting action unauthorized. The three artifacts separate these layers by giving each proposal a path through interpretation, judgment, recording, and possible enactment.

2 Scope and Method of the Artifacts

The three programs are executable witnesses to an architecture, not scale models of production infrastructure. Their methodological role is to isolate distinctions that frequently become entangled in larger systems: raw input and recognized proposal, persistence and enactment, deterministic judgment and legitimate authority, accepted state and documentary history, local preparation and external effect. Smallness makes these distinctions inspectable, but does not by itself establish safety, scalability, or institutional adequacy.

The claims of the article therefore occur at three levels. At the implementation level, the programs establish only properties that follow from their source code, tests, and bounded execution environments. At the architectural level, they exhibit separable operators for proposal formation, recording, admission, transition, and observation. At the governance level, they show where authorization predicates must occur, but do not derive the legitimacy of those predicates from computation alone. An implementation may instantiate the architecture without satisfying the operational requirements of a distributed or high-assurance deployment.

The artifacts should consequently be understood as constructive examples. They demonstrate that recording and operative continuation can be separated and that refusal need not imply erasure. They do not demonstrate that every implementation of this separation is secure, resource-bounded, convergent, or legitimate. Those stronger properties require additional mechanisms whose placement within the architecture can nevertheless be specified.

This distinction also limits the security claim made by subtractive design. Removing an unnecessary dependency or capability eliminates effects reachable only through that dependency or capability, but it does not prove the safety of what remains. If $\Gamma(P)$ denotes the capabilities available to program P and $\mathcal{E}(\Gamma)$ denotes the effects reachable through them, then capability removal supplies the monotonic relation

$$\Gamma(P') \subseteq \Gamma(P) \quad \Longrightarrow \quad \mathcal{E}(\Gamma(P')) \subseteq \mathcal{E}(\Gamma(P)). \tag{2}$$

This is a reduction in reachable effect, not a proof that every member of the smaller effect set is safe. Subtractive security is a design discipline with inspectable local consequences rather than a complete security theorem.

3 Proposal, Record, Admission, and Continuation

The analysis treats source code, command-line behavior, tests, build metadata, and release procedures as a single artifact. Mackenzie's account of code as sociality directs attention to the relations assembled by executable systems rather than to algorithms in isolation [2]. Marino's critical code studies supplies the complementary interpretive claim that source code can be read for cultural and rhetorical commitments as well as function [3]. The relevant unit here is therefore the complete relation among an offered input, its recognition as a proposal, the constraints imposed upon it, the record retained, the state transition conditionally produced, and the account subsequently presented to a user.

Raw input and recordable proposal must first be distinguished. Let X be the space of offered inputs and I the space of finite, well-formed proposals. An ingress operator

$$Q : X \to I \cup \{\bot\} \tag{3}$$

performs framing, size enforcement, authentication where required, and syntactic normalization. The value $\bot$ means that no proposal was formed. A byte stream rejected because it exceeds an ingress limit is therefore not a rejected historical proposal; it is material that never entered the proposal domain.

Let S, I, H, M, and Y denote respectively the spaces of operative states, finite proposals, documentary histories, finite observation summaries, and observable outputs. Once $i_n \in I$ has been formed, the system supplies four distinct operators:

$$\mathcal{R} : H \times \mathrm{Record} \to H, \quad A : H \times S \times I \to \{0,1\}, \quad T : S \times I \to S, \quad O : S \times M \to Y. \tag{4}$$

The observation operator need not scan or reproduce the entire documentary history. An incremental summarizer

$$U : M \times \mathrm{Record} \to M \tag{5}$$

maintains the indexes, counters, authenticated roots, or bounded views required for output.

For a recognized proposal i_n, let $a_n = A(H_n, \sigma_n, i_n)$. Recording is logically unconditional with respect to a_n, while continuation is conditional:

$$H_{n+1} = \mathcal{R}(H_n, \rho(i_n, a_n)) = H_n \parallel \rho(i_n, a_n), \tag{6}$$

$$\sigma_{n+1} = \begin{cases} T(\sigma_n, i_n), & a_n = 1, \\ \sigma_n, & a_n = 0, \end{cases} \tag{7}$$

$$m_{n+1} = U(m_n, \rho(i_n, a_n)). \tag{8}$$

Here $\rho : I \times \{0,1\} \to \mathrm{Record}$ is a canonical recording function. It may retain the complete proposal when proposals are intrinsically bounded, or retain a canonical digest together with provenance, disposition, policy version, and a reference to separately

governed payload storage. The symbol $\|$ denotes append. These equations specify logical commitments rather than an implementation's instruction order: evaluation may precede the atomic append, but no recognized proposal completes without a recorded disposition.

The word "unconditional" is therefore relative to the proposal domain I: every proposal the system acknowledges as submitted receives a documentary disposition, including denial. It does not mean that the system accepts unbounded byte strings, bypasses authentication, promises infinite online storage, or refuses backpressure. Audit completeness begins after a proposal has crossed the explicitly defined ingress boundary.

The architectural sequence is more precisely

$$\boxed{\text{offered input} \longrightarrow \text{recognized proposal} \longrightarrow \text{record} \longrightarrow \text{admission} \longrightarrow \text{continuation}}, \tag{9}$$

with recording unconditional for recognized proposals and continuation selective. The ingress rule Q, admission rule A, transition rule T, recording representation ρ, and observation summary U are all part of the specification. None may be replaced by an unspecified instruction to "save everything."

Across the artifacts, the governing predicates become progressively more relational. In their simplest forms,

$$V : I \to \{0,1\}, \qquad V(i): \text{ syntactic executability}, \tag{10}$$

$$J : I \to \{0,1\}, \qquad J(i): \text{ institutional approval}, \tag{11}$$

$$A : H \times S \times I \to \{0,1\}, \qquad A(H,\sigma,i): \text{ historical continuation}. \tag{12}$$

V is local and stateless, J is algorithmic and may remain stateless with respect to earlier cases, and A is stateful and path-dependent. In particular, $A(H,\sigma,i) \neq A(i)$ in general: there may be states σ_1, σ_2 for which $A(H_1,\sigma_1,e) = 1$ while $A(H_2,\sigma_2,e) = 0$. Historical admissibility is not simply stricter validity. It is a different regime in which a well-formed, judged, and recorded event may still fail to continue the operative world.

Artifact	**Language**	**Primary constraint**	**Retained consequence**
`hello-tool`	Go	invocation and release validity	exit status, tests, Git tag
`reality-permit`	Perl	calculated administrative threshold	reproducible ruling
`counterfactual-git`	Rust	causal admissibility	audit commit and conditional HEAD

Table 1: Execution, judgment, and history as increasingly relational constraint regimes.

4 Logical Invariants and Physical Bounds

Append-only history is a logical property of the record, not a requirement that every payload remain forever in primary memory. A practical implementation may

combine a bounded ingress queue, durable journal segments, immutable archival tiers, content-addressed payloads, checkpoints, and authenticated compaction. What must survive compaction is the ability to establish which proposal was acknowledged, which policy version evaluated it, what disposition resulted, and which accepted predecessor it claimed.

Let $\preceq$ denote the prefix relation over documentary histories. The central invariants are

$$H_n \preceq H_{n+1}, \qquad \text{documentary monotonicity}, \tag{13}$$

$$A(H_n, \sigma_n, i_n) = 0 \Longrightarrow \sigma_{n+1} = \sigma_n, \qquad \text{rejection non-effect}, \tag{14}$$

$$A(H_n, \sigma_n, i_n) = 1 \Longrightarrow \sigma_{n+1} = T(\sigma_n, i_n), \qquad \text{admitted transition}, \tag{15}$$

$$r_{n+1} \in \mathrm{id}(\mathrm{Accepted}(H_{n+1})) \cup \{\varnothing\}, \qquad \text{head traceability}. \tag{16}$$

Here r_n denotes the operative reference later instantiated by the Rust artifact. Unlike set inclusion, the prefix relation preserves order and multiplicity. Two identical proposals submitted at different times remain two documentary events.

Resource boundedness is expressed independently. For configured proposal bound b_I, ingress capacity b_Q, and online storage budget b_H,

$$|\,\mathrm{encode}(i)\,| \le b_I, \qquad |\,\mathrm{queue}\,| \le b_Q, \qquad |\,\mathrm{online}(H)\,| \le b_H. \tag{17}$$

When a bound is reached, the system may apply backpressure, refuse to form a new proposal, or move sealed segments to archival storage. It may not acknowledge a proposal and then silently omit its disposition. This yields the distinction

$$\text{refusal before acknowledgment} \neq \text{recorded rejection after acknowledgment}. \tag{18}$$

Compaction preserves logical append-only semantics when it replaces a sealed prefix by a verifiable commitment and replay checkpoint rather than rewriting the meaning of earlier decisions. If $H_{0:k}$ is a sealed prefix, an archived representation may retain

$$\chi_k = \big(\mathrm{root}(H_{0:k}), \sigma_k, m_k, \nu_k\big), \tag{19}$$

where root is an authenticated commitment, σ_k is the operative checkpoint, m_k is the observation summary, and ν_k identifies the transition and policy versions required to interpret the prefix. Online cost can then depend on the active suffix rather than the total age of the system. These mechanisms do not make storage infinite. They make exhaustion behavior explicit while preserving the distinction among resource refusal, documentary rejection, and operative transition.

The comic language of the latter two programs is not incidental decoration. Humor makes the contingency of institutional rules visible. A rule such as “an apocalypse requires an admissible alibi” is absurd in content yet precise in operation. The mismatch separates formal enforceability from substantive justification: software can apply a rule perfectly even when the rule is ridiculous.

5 Secure Minimalism in `hello-tool`

`hello-tool` is intentionally ordinary. Its runtime behavior is implemented entirely with the Go standard library. A private `run` function accepts arguments and explicit output

streams, returning an integer exit status. The process-level `main` function performs only the final boundary operation:

```
func main() {
    os.Exit(run(os.Args[1:], os.Stdout, os.Stderr))
}
```

This separation is a small but consequential design decision. It turns process behavior into a testable value. Successful greeting and version queries return zero; malformed flags and unexpected positional arguments return two. Output can be captured in memory without replacing global streams. Security here does not mean the presence of elaborate cryptography. It means narrowing the artifact's powers and making its boundary conditions observable.

The dependency policy is equally restrictive. The program does not add `golang.org/x/mod` merely because a security release of that module motivated the surrounding discussion. A patched dependency is not safer than no unnecessary dependency. The principle is qualitative but concrete:

$$\begin{aligned} &\text{attack surface} \uparrow \quad \text{as unnecessary capability} \uparrow \\ &\qquad\qquad\qquad\quad \text{or unnecessary dependency surface} \uparrow \end{aligned} \tag{20}$$

Security is achieved here by refusing powers rather than surrounding unnecessary powers with additional filters. Capabilities are few, third-party runtime dependencies are zero, and the principal behaviors have tests.

Toolchain selection is project-local. The `go.mod` file declares Go 1.26.0 semantics and requests the patched Go 1.26.6 toolchain. This avoids the global setting `goenv-wGOTOOLCHAIN=...`, which would silently affect unrelated repositories. The distinction is important: reproducibility is not achieved simply by choosing a version, but by attaching that choice to the artifact whose behavior it governs.

The management script extends the same philosophy to the release boundary. Its `check` operation formats, vets, race-tests, and invokes `govulncheck` when available. Its `release` operation requires a clean working tree, executes the checks, and creates a signed local tag. It does not push the tag. Publication remains a separate, explicit act. This resembles what Nissenbaum calls contextual integrity [4]: an action that is acceptable within one informational boundary does not automatically authorize transmission across the next. A release may be locally formed and cryptographically identified without yet being made public.

6 Procedural Administration in `reality-permit`

The Perl program begins where the Go scaffold ends: with a controlled interface and deterministic state. It then uses those properties to satirize official judgment. A seed initializes a small linear-congruential generator. More generally, let

$$F : (s, p, a) \longmapsto (r, q, m), \tag{21}$$

where s is the seed, p the permit number, a the appeal state, and (r, q, m) the ruling, score, and remedial measure. The generator selects an adjective, noun, allegation,

remedy, and three numerical measurements: coherence, majesty, and paperwork. Their mean, adjusted on appeal, determines whether the proposed universe is approved:

$$S = \left\lfloor \frac{C + M + P}{3} \right\rfloor + 7a, \qquad A = \mathbf{1}_{\{S \geq 55\}} \tag{22}$$

where $a \in \{0, 1\}$ records whether an appeal has been lodged and $\mathbf{1}_{\{S \geq 55\}}$ is the Iverson indicator, equal to one exactly when the scoring threshold is met.

The equation is transparent and reproducible, but it is not rational in any meaningful substantive sense. "Majesty" and "paperwork" receive the same weight as "coherence." An appeal contributes seven points regardless of its argument. The resulting system captures an important property of quantified administration: reproducibility can stabilize arbitrariness rather than remove it. As Bowker and Star observe, classifications acquire force through infrastructures that make categories consequential [1]. The fictional ministry does precisely this. It converts whimsical categories into a ruling, a permit identifier, and a required remedial act.

The bureaucracy is deterministic because identical application triples produce identical decisions,

$$(s, p, a) = (s', p', a') \implies F(s, p, a) = F(s', p', a'), \tag{23}$$

but the governing contradiction remains:

$$\mathrm{Det}(F) = 1 \not\Rightarrow \mathrm{Leg}(F) = 1. \tag{24}$$

Determinism answers whether equivalent inputs receive equivalent outputs. It does not answer whether the categories, weights, threshold, or authority of the scoring system are justified. The code knows only that its arbitrariness is repeatable.

The deterministic seed is crucial to the joke. Seed 1982 always yields the same administrative world for a given permit number and appeal state. The ministry cannot defend itself by claiming discretion, accident, or changing circumstances. Its absurdity is repeatable. Tests explicitly assert that "the bureaucracy is deterministic." This test name condenses the program's thesis: computation does not oppose bureaucracy; it can perfect bureaucracy's repeatability.

At the same time, the program sharply limits its real authority. It uses only core Perl modules. It reads no files, opens no network connections, launches no subprocesses, and modifies no environment. Its ornate declarations are inversely proportional to its actual capabilities. The Ministry of Possible Affairs claims jurisdiction over universes but possesses only the authority to print text. This gap between rhetorical and effective power is itself a security feature. The program demonstrates that maximal imaginative scope can coexist with minimal machine privilege.

7 Policy, Legitimacy, and Contestability

Determinism and legitimacy belong to different kinds of claim. Determinism is a property of a function under fixed inputs and fixed semantics. Legitimacy is a relation among a rule, an authority, a jurisdiction, affected participants, and procedures for revision or contestation. It cannot be generated merely by repeating a computation.

A decision system should therefore make its policy context explicit. Let

$$\Pi = (p, \nu, \alpha, \delta, \epsilon, \kappa) \tag{25}$$

denote a policy bundle consisting of rule text or executable predicate p, version ν, asserted authority α, jurisdiction δ, effective interval ϵ, and contestation procedure κ. Admission is evaluated as

$$A(H, \sigma, i; \Pi), \tag{26}$$

and the resulting record identifies the policy bundle used. Deterministic replay requires the same ordered proposals, initial state, transition semantics, and policy versions. A later policy must not be silently substituted for the policy under which an earlier decision was made.

For governance arrangement G, one may define a policy-authorization predicate

$$L_G(\Pi) \in \{0, 1\}, \tag{27}$$

but the subscript is indispensable. L_G records that a policy satisfies the publicly stated requirements of a particular governance arrangement; it does not prove universal moral legitimacy. Those requirements might include provenance, delegated authority, jurisdiction, notice, consistency constraints, expiration, review, and an available means of appeal. The architecture can make such requirements inspectable and testable, but cannot settle the prior political question of who is entitled to define G.

Contestation is itself representable without retroactively deleting the contested decision. If d_j is a recorded disposition, an appeal is a new proposal

$$i_{n+1} = \text{contest}(d_j, \gamma), \tag{28}$$

where γ supplies grounds or evidence. The appeal receives its own record and may produce a reversal, amendment, or affirmation. A successful reversal changes operative state through a subsequent admitted transition:

$$H_j \preceq H_{n+1} \quad \text{while} \quad \sigma_{n+1} = T_{\text{remedy}}(\sigma_n, d_j). \tag{29}$$

The earlier decision remains historically true as an occurrence even when it no longer governs current state.

The comic ministry deliberately supplies a policy whose categories are reproducible but substantively indefensible. Its function is not to define legitimacy but to demonstrate why legitimacy cannot be inferred from procedural regularity. Humor makes the missing authorization visible; the policy bundle and contestation relation show where a noncomic system must state it.

8 Append-Only Counterfactual History

The Rust program gives the bureaucratic ruling a historical substrate. A generated `Event` contains a year, subject, verb, object, and hash. Submitting the event creates a `Commit` containing an identifier, the accepted predecessor reference, the event, an

acceptance flag, the applicable policy version, and a reason. Every recognized proposal enters the documentary sequence, including rejected proposals. Only accepted proposals occupy a year and advance the operative reference.

To prevent documentary history from being confused with its accepted head, let H_n denote the complete audit history after n recorded proposals, let r_n denote the identifier of the current accepted head, and let σ_n denote the operative state derived from that head. For proposed event e_{n+1}, define

$$a_{n+1} = A(H_n, \sigma_n, e_{n+1}; \Pi_n), \tag{30}$$

$$c_{n+1} = \mathsf{record}(r_n, e_{n+1}, a_{n+1}, \Pi_n), \tag{31}$$

$$H_{n+1} = H_n \parallel c_{n+1}, \tag{32}$$

$$r_{n+1} = \begin{cases} \mathsf{id}(c_{n+1}), & a_{n+1} = 1, \\ r_n, & a_{n+1} = 0, \end{cases} \tag{33}$$

$$\sigma_{n+1} = \begin{cases} T(\sigma_n, e_{n+1}), & a_{n+1} = 1, \\ \sigma_n, & a_{n+1} = 0. \end{cases} \tag{34}$$

The documentary history therefore grows monotonically even when both the operative reference and operative state remain unchanged:

$$H_n \preceq H_{n+1}, \qquad |H_{n+1}| = |H_n| + 1, \qquad a_{n+1} = 0 \Longrightarrow (r_{n+1}, \sigma_{n+1}) = (r_n, \sigma_n). \tag{35}$$

Recording, reference advancement, and state transition are thereby distinct operations rather than three names for a single mutation. A denial becomes part of documentary history without becoming part of the accepted world.

The admissibility predicate can be made explicit as the conjunction

$$A(H_n, \sigma_n, e_{n+1}; \Pi_n) = P_1(H_n, e_{n+1}) \wedge P_2(H_n, e_{n+1}) \wedge P_3(H_n, e_{n+1}), \tag{36}$$

where P_1 requires $\mathrm{Year}(e_{n+1}) \notin \mathrm{OccupiedYears}(H_n)$, P_2 preserves institutional identity by forbidding the silent replacement of the history that contains the reviewing institution, and P_3 requires all causal prerequisites already to be reachable from r_n—in the example, an accepted global alibi before an apocalypse. Thus A is not a generic validity flag attached to an isolated event. It states whether this event may continue this history under policy Π_n.

Three rules define causal inadmissibility. A year cannot be occupied twice. An event in which history itself "quietly replaced" something is rejected because it would erase the committee reviewing the commit. Finally, a legally distinct apocalypse is rejected unless an earlier accepted event has supplied everyone's alibi. These rules are intentionally comic, but their architecture resembles validation in append-only systems, event sourcing, and version control. In each case, the system distinguishes the existence of a proposed mutation from its admissibility as current state.

Rust supports the conceptual model through ownership and explicit data types, although the program does not depend on unsafe code or external crates. The `Universe` owns both its commits and its set of occupied years. Accepted state is derived through a controlled method rather than unrestricted mutation. The implementation detail matters insofar as it protects the distinction between the larger audit record and the narrower operative trajectory.

The name `counterfactual-git` also matters, but it names a conceptual reinterpretation rather than an implementation of Git semantics. Git supplies the broad distinction between persistent objects and operative references: an object may exist without being named by the current branch. Ordinary Git itself has no normative predicate deciding whether a proposed history ought to become real; it permits non-linear forks whose tips may later be merged. The Rust artifact adds precisely this missing normativity. It maintains a strictly linear accepted timeline alongside a monotonically increasing audit spine containing every proposal. A rejected event resembles an unmerged, non-advancing branch tip, but the implementation grants it neither an independent mutable reference nor a future merge operation. History is not the set of everything recorded, but a selected trajectory through a larger field of proposals.

9 Replay, Projection, and Sufficient Historical State

Projection need not appeal to an unspecified global scan. It can be defined as a deterministic fold over ordered records under versioned semantics. Let an evaluation state be

$$z_n = (\sigma_n, r_n, \mu_n), \tag{37}$$

where μ_n contains the historical indexes required by admission, such as occupied years, reachable prerequisites, policy versions, or institutional continuity markers. Then

$$z_{n+1} = \Phi_{\nu_n}(z_n, c_{n+1}), \qquad z_n = \mathrm{fold}(\Phi, z_0, H_{1:n}), \tag{38}$$

where ν_n identifies the semantics used for the recorded decision. Rejected records update documentary indexes and observation summaries but do not apply the operative transition.

The full history need not be consulted by every admission decision. A summary μ_n is sufficient for predicate A when

$$\mu(H_1) = \mu(H_2) \Longrightarrow A(H_1, \sigma, i; \Pi) = A(H_2, \sigma, i; \Pi) \tag{39}$$

for every relevant σ, i, and Π. Admission may then be implemented incrementally by a predicate $\widehat{A}$ satisfying

$$\widehat{A}(\mu_n, \sigma_n, i; \Pi_n) = A(H_n, \sigma_n, i; \Pi_n). \tag{40}$$

Different policies induce different sufficient summaries; the model assumes no universal summary of constant size.

Conflicting entries are not resolved by projection after the fact. They are evaluated in documentary order against the operative state produced by the accepted prefix. If two proposals claim the same exclusive resource, the first admitted proposal may alter μ_n so that the second is rejected. The record retains both the ordering decision and both dispositions.

Replay determinism is conditional rather than absolute. If two replicas share an initial checkpoint, the same totally ordered records, identical transition semantics, identical policy bundles, and identical canonicalization rules, then

$$(z_0, H, \Phi, \Pi) = (z_0', H', \Phi', \Pi') \Longrightarrow z_n = z_n'. \tag{41}$$

Version skew is not silently absorbed. It is either excluded by agreement on versions or represented as an explicit migration event with its own admission rule.

10 Ordering, Concurrency, and Remote Admission

Selective continuation does not itself solve consensus. It specifies what a system does once it has an authoritative proposal order, or once it has selected a branch of a partially ordered proposal structure. The in-memory Rust artifact uses one process and obtains order from sequential method calls. A networked implementation must supply an ordering mechanism appropriate to its trust and failure model.

Let $\rightsquigarrow$ denote causal precedence among proposals. Concurrent proposals may initially form a partial order

$$\mathcal{D} = (I, \rightsquigarrow) \tag{42}$$

rather than a single vector. A deployment requiring one operative state must add an ordering service

$$\Lambda : \mathcal{D} \to (i_1, i_2, \ldots) \tag{43}$$

whose output is a linear extension of $\rightsquigarrow$. The service might be supplied by a single trusted writer, an append service, a transactional database, a quorum protocol, or a consensus algorithm. The selective-continuation model begins after this deployment-specific authority has produced an order; it does not treat one process's arrival order as a universal fact.

This separation identifies two independently necessary questions:

$$\boxed{\text{Which proposal is next?} \quad \neq \quad \text{May that proposal advance the state?}} \tag{44}$$

Consensus or serialization answers the first. Admission answers the second. A system can agree perfectly on proposal order while applying an illegitimate admission policy, just as it can possess a defensible policy but fail to agree on which proposal it evaluated first.

The local-to-remote boundary introduces a further distinction. A locally prepared proposal may satisfy a project-local predicate without being authorized by the receiving system:

$$A_{\text{local}}(H_L, \sigma_L, i; \Pi_L) = 1, \tag{45}$$

$$A_{\text{remote}}(H_R, \sigma_R, i; \Pi_R) \in \{0, 1\}. \tag{46}$$

Local admission authorizes transmission or preparation; it does not predetermine remote admission. The receiving system evaluates the proposal against its own current state, policy, ordering position, and authority structure. A complete exchange therefore produces a receipt

$$\eta = \text{receipt}\big(\text{id}(i), \text{position}(i), \Pi_R, a_R, \text{reason}\big), \tag{47}$$

which the sender may record locally.

The Go release procedure is an elementary instance of this boundary. Creating a signed local tag establishes a local object and records an intention to release. It does not imply that a remote repository has received, ordered, authorized, or published that tag. The explicit push marks the point at which a second authority and a second state become involved.

11 Continuation Space and Historical Evidence

Moving V, J, and A ahead of the case studies changes their interpretation. The artifacts no longer lead belatedly to admissibility; they instantiate increasingly relational versions of a common architecture. `hello-tool` governs whether a command may execute. `reality-permit` governs whether an application may receive institutional approval. `counterfactual-git` governs whether a recorded event may generate the next operative state.

The accepted timeline is consequently a sequence of admissible continuations:

$$\sigma_{n+1} = T(\sigma_n, e_n) \quad \Longleftrightarrow \quad A(H_n, \sigma_n, e_n; \Pi_n) = 1. \tag{48}$$

Define the presently admissible continuation space by

$$\boxed{\mathcal{C}(H, \sigma; \Pi) = \{e \in I : A(H, \sigma, e; \Pi) = 1\}}. \tag{49}$$

A viable operative history is a path whose next event always lies in the current continuation space. A rejected event is not thereby unrepresentable, unrecordable, unimaginable, or intrinsically false. It lies outside $\mathcal{C}(H, \sigma; \Pi)$ under the present constraints. Because $\mathcal{C}$ depends on accumulated history, the same proposal may become admissible or inadmissible along different trajectories.

The continuation space is an extensional specification, not necessarily a set intended for enumeration. The proposal domain may be extremely large or countably infinite while membership remains decidable for individual finite proposals. Operationally, the system answers

$$\mathrm{member}_{\mathcal{C}}(H, \sigma, e; \Pi) = A(H, \sigma, e; \Pi) \tag{50}$$

rather than materializing every member of $\mathcal{C}(H, \sigma; \Pi)$.

A deployment may further restrict consideration to a finitely represented candidate language I_G generated by grammar, schema, capability discipline, or typed interface G:

$$\mathcal{C}_G(H, \sigma; \Pi) = \{e \in I_G : A(H, \sigma, e; \Pi) = 1\}. \tag{51}$$

The distinction parallels that between a language and an enumeration of all its sentences. A continuation predicate can specify which proposals are admissible without generating all possible proposals in advance.

Accordingly, a narrowing intervention transforms predicates rather than an already materialized set. If $\chi_{\mathcal{C}} : I \to \{0, 1\}$ is the characteristic predicate of the current continuation space, intervention N_j produces

$$N_j(\chi_{\mathcal{C}})(e) = \chi_{\mathcal{C}}(e) \wedge K_j(H, \sigma, e), \tag{52}$$

where K_j is the additional constraint introduced by the intervention. Consequently,

$$\mathcal{C}_{N_j}(H, \sigma; \Pi) \subseteq \mathcal{C}(H, \sigma; \Pi) \tag{53}$$

without assuming that either space can be exhaustively constructed. Rejected proposals do not disappear. They remain in the recorded history as evidence concerning the boundary of the continuation space. If H_{op} is the accepted-event subsequence

and H_{rec} is the documentary sequence of all proposals, let $\sqsubseteq$ denote the subsequence relation. Then

$$H_{\text{op}} \sqsubseteq H_{\text{rec}}, \qquad e \notin H_{\text{op}} \;\not\Rightarrow\; e \notin H_{\text{rec}}. \tag{54}$$

The documentary record has greater informational scope because it preserves alternatives that did not become true of the operative world. Persistence therefore precedes operative truth: a distinction can survive as information before, and even without, being selected as reality.

This relation gives a compact form to computation-as-history. The current state is not the whole computational object; Equation 38 defines it as the operative component of a deterministic fold over a richer documentary history. The significant object is not merely the sequence $(\sigma_0, \sigma_1, \ldots)$ but the larger space of attempted continuations from which the operative path is selected. A failed transition is not necessarily a false proposition or disposable error. It can be a transformation that fails the current viability criterion while remaining useful substrate for later diagnosis, correction, or repair.

The progression is recursive rather than teleological. Historical admissibility does not supersede operational control; it re-imports control at the level of history. A bureaucratic ruling can be reversed, but an append-only system represents reversal as a subsequent event whose own validity, judgment, and admissibility must be evaluated. Reversion changes the operative trajectory without making the original ruling cease to have occurred. The higher-order system therefore contains the earlier regimes as constraints on what may be remembered next.

12 Language as Argument

The choice of three languages produces an implicit history of programming practice. Go supplies the austere utility: explicit error codes, a standard-library flag parser, straightforward tests, and conventional module metadata. Perl supplies the textual machine: compact arrays of phrases, interpolated proclamations, and an administrative document assembled as a stream. Rust supplies the historical machine: enums are not required here, but structs, ownership, iterator-based queries, and controlled mutation make state transition the center of the design.

It would be simplistic to claim that each language has an essential personality. Any of the programs could be rewritten in either of the other languages. The point is instead rhetorical fit. The Go artifact emphasizes operational legibility. The Perl artifact emphasizes generative language and the instability between solemn form and absurd content. The Rust artifact emphasizes the integrity of state and transition. Language choice participates in the argument without determining it.

All three programs also reject dependency accumulation. This makes the language runtimes themselves visible. There is no framework to absorb responsibility for parsing, state, generation, or output. Such minimalism has pedagogical value, but its larger significance is epistemic: a reader can audit the path from input to ruling. The cost is that each program implements a small amount of infrastructure itself. In these artifacts that trade is appropriate because inspectability is one of the intended results.

13 Humor, Authority, and Safe Power

The ridiculous content performs a technical function. By making rules obviously contestable, it prevents the reader from confusing successful execution with justified authority. A machine may calculate coherence to the nearest percentage point, generate a stable permit number, and produce an official-looking seal. None of those operations establishes that the Ministry of Possible Affairs ought to govern universes. Likewise, a causal validator may preserve a flawless audit trail while enforcing a foolish policy.

The artifacts therefore instantiate the separations stated in Equation 1. `hello-tool` addresses execution and authorization by limiting capabilities and stopping before a network push. `reality-permit` exposes the gap between a reproducible model and a legitimate rule. `counterfactual-git` shows that even a rule judged legitimate must be evaluated relative to history, with refusal preserved as evidence rather than erased as noise.

This is a model of safe power through bounded effect. The Perl ministry can proclaim but cannot alter the host. The Rust universe can mutate only its in-memory state. The Go release script can create a local signed tag but cannot publish it automatically. Each artifact grants itself enough power to make its conceptual claim while withholding a consequential external action. Their restraint is architectural rather than apologetic.

The temporal and operational distinctions can now be stated together:

$$\boxed{\text{existence} \neq \text{recording} \neq \text{admission} \neq \text{continuation} \neq \text{external effect}}. \tag{55}$$

The inequality signs mark categorical distinctions, not numerical comparisons. A generated universe can exist without authorization; a proposed event can be recorded without admission; an admitted local release can stop before external publication.

14 Conclusion

The three programs form a small computational essay in selective continuation. `hello-tool` demonstrates that execution and external effect can be separated by refusing unnecessary capability. `reality-permit` demonstrates that deterministic judgment can be reproducible without becoming legitimate. `counterfactual-git` demonstrates that historical systems need not choose between allowing an event and forgetting it; refusal can be recorded monotonically while operative state remains unchanged.

The artifacts therefore instantiate a general architecture rather than merely suggest a design preference:

$$\boxed{\text{A proposal may be preserved without being permitted to continue operative state.}} \tag{56}$$

In this architecture, history consists of persistent proposals, explicit admissibility constraints, and selective continuation. The current world is a projection of a documentary field richer than the trajectory it authorizes. Rejected and reversed events remain available as evidence about the boundary of that trajectory. The artifact need not choose between forgetting an event and allowing it to become part of the world.

This result applies beyond the three toys: to releases, policy systems, scientific model selection, simulations, institutional decisions, and version histories. A trustworthy system should make the predicates governing proposal, record, admission, continuation, and external effect inspectable. It should preserve enough history to explain refusals and reversals without pretending that reversal deletes occurrence. Software's philosophical weight lies not only in what it does, but in what it remembers having done. History can be append-only while reality remains selective.

References

[1] Geoffrey C. Bowker and Susan Leigh Star. *Sorting Things Out: Classification and Its Consequences*. MIT Press, 1999.

[2] Adrian Mackenzie. *Cutting Code: Software and Sociality*. Peter Lang, 2006.

[3] Mark C. Marino. *Critical Code Studies*. MIT Press, 2020.

[4] Helen Nissenbaum. *Privacy in Context: Technology, Policy, and the Integrity of Social Life*. Stanford University Press, 2010.