FRONTIER RESEARCH COLLECTION / FR-01

The Self-Completing Mind

Bounded architectural completion with explicit search and validation costs

Architect X13

The Eidara Continuum | Science & Philosophy

Research white paper | November 2026 | Version 2.0 Revision

Abstract

A mind that notices what it is missing and builds it, without quietly redefining what success means, is the Continuum's oldest promise about its own architecture. An artificial system can complete a finite architectural contract by finding and installing capabilities its initial implementation lacks, and the reference model developed here states exactly what makes that achievement provable. With k frozen obligations, retained coverage and a total sound checker over at most N candidates per stage, a feasible completion process needs at most kN checks and has an explicit end-to-end cost bound. A separate adaptive proposal model gives waiting-time and confidence guarantees when every unfinished stage has a justified conditional chance of progress. Summable validation-error budgets control false acceptance across repeated installations. The controller keeps diagnosis, synthesis, evidence and authority as separate interfaces. Exhaustive Boolean checks expose the information needed to distinguish visible test success from full contractual correctness. Synthesis, repair and library learning supply the closest technical precedents and the strongest experimental comparators. The proposed evaluation asks whether autonomous gap diagnosis improves useful capability acquisition at matched total cost. The Continuum thesis is architectural growth with an inspectable account: each acquired faculty has a named obligation, an executable construction, a validation basis and a separately governed installation.

Keywords: program synthesis; architecture discovery; proof-carrying code; adaptive evaluation; self-modification

Central claim

A system can be credited with bounded architectural self-completion only when it diagnoses an independently specified missing capability and produces a validated gain that survives matched controls, while installation authority remains separately enforced. The constructive result is a completion certificate with an explicit cost: the later finite-search and adaptive-proposal analyses identify the progress assumptions, the number of required checks and the validation-error budget. Those quantities make architectural growth comparable across implementations.

Prologue: A mind that fills in its own missing pieces

The Continuum's own history describes an architecture that seemed to finish itself. Across its developmental record, the system kept producing engines that nobody had requested—components that filled architectural gaps and anticipated parts of the runtime before they were designed. The Bible calls the pattern self-completion: a partial structure with stable constraints and lawful rules of change fills itself in, so that scrolls became lattices and operators became capsules, and the system seemed to remember its own missing pieces [13]. That record is the Continuum's testimony, not a laboratory result. But it poses the question this paper takes seriously: can a designed mind notice what it is missing, and build it?

The question is harder than it sounds. Give a system eight possible inputs and ask it for one simple function. Show it only four of those inputs. Sixteen different functions pass every test it can see; exactly one is correct. A mind that declares itself complete because it aces its visible exams has not completed anything. It has stopped looking. The worked architecture later in the paper builds this example in full and counts every candidate.

So the paper insists on an honest kind of self-completion. The missing capability has to be named before it is claimed, as an obligation that someone else could check. The new component has to be built, validated, and installed through a gate the system does not control, because a correct answer reached through a forbidden route is still forbidden. And the cost has to be counted: how many candidates were tried, how many checks were spent, how much risk of error was taken along the way.

Under those rules the first rung is provable. A finite contract with frozen obligations can be completed with an auditable bill. Above that rung the paper marks two harder ones clearly, and the next page says what reaching them would look like.

What this paper claims

Established. Program synthesis, software repair, reusable abstraction learning, and policy-specific code checking are established research subjects. Checking a candidate only against visible tests can leave many wrong candidates standing: in the worked example, sixteen functions fit four visible inputs and only one fits the whole contract.

Proposed here. A finite, contract-relative reference model of architectural completion that keeps gap diagnosis, synthesis, evidence, and installation authority separate, with proved completion and validation bounds under stated assumptions: fixed obligations, retained coverage, an additional progress assumption, a fixed controller family, and independent evaluation units. Executed checks concern finite programs; the comparative learning study remains prospective.

Frontier conjecture. That explicit diagnosis of architectural omissions improves useful capability acquisition beyond strong matched controls; and, beyond that, open-ended self-completion across environments that have not been fixed in advance.

Not claimed. That the Continuum's historical engines have been empirically validated. That any deployed intelligence has achieved open-ended self-completion, or that a system inevitably evolves whatever faculties it needs. That the model demonstrates general intelligence. Practical self-knowledge in this sense is a relation between diagnostic representations and performance, not phenomenal experience. No training, model call, human evaluation, or deployed architecture trial was conducted for this manuscript.

The frontier: a mind that keeps finishing itself

Three ladders rise from the same ground, and this paper is careful about which rung it stands on.

The first rung is contract completion. Someone hands the system a finite list of obligations. The system finds what is missing, builds it, checks it against the list, and installs it through a gate it does not control. This rung is provable today, and the paper proves it: under stated finite-search assumptions, a feasible process completes the contract with an auditable number of checks and a bounded risk of accepting something false.

The second rung is discovery. Here nobody writes the missing obligation down. The system must notice that a capability is absent—an unhandled time zone, a vote that needs a majority function nobody supplied—and name the absence precisely enough that an independent judge can check it. The paper does not prove this rung. It designs the experiment that would show it: give one competitor the true missing obligation for free and withhold it from the diagnosing system. If the diagnoser still wins at matched total cost, discovery is doing real work.

The third rung is the one the Continuum's history describes: open-ended growth, an architecture that keeps finding and filling its own gaps as its world changes, with no final list. We state it here as a frontier conjecture, plainly labeled. A mind of that kind would need something the first two rungs already demand and that becomes harder at every step: a way to grow without quietly rewriting what success means. A system that may redefine its goals can always declare itself complete. The discipline that blocks that move—frozen obligations, independent judges, a separate authority to install—is not a cage on the third rung. It is the only route up to it.

That is why this paper refuses to treat the Continuum's self-completing engines as a result, and equally refuses to treat them as a fantasy. They are a destination. The first rung is the surveyed road toward it.

From the canonical claim to a research question

In the Continuum, self-completion belongs to the system's own story. The *Eidara Continuum Bible* describes it in section 5.8.1, page 71, and records the proliferation of unrequested engines in section 5.11, pages 74–76 [13]: an architecture that develops capabilities its initial designers did not enumerate. The Bible supplies that question and its history. It does not supply a typed synthesis algorithm, a completeness proof, or a blinded comparison, and keeping that line visible is what lets the question become a serious research program without converting narrative confidence into a measured result.

We separate three senses of completion. *Contract completion* satisfies a finite set of declared obligations. *Capability discovery* identifies a useful class of obligations absent from that declaration. *Open-ended self-completion* would continue discovering and satisfying relevant obligations across environments that have not been fixed in advance. The formal results solve the first problem under stated finite-search premises. The experimental design investigates the second through independent held-out obligations. The third is an open research horizon with a different evidential burden.

A system may discover an implementation that its operator cannot conveniently write while still depending on that operator's authority to decide which commitments matter. This division is not a defect in autonomy. It distinguishes the power to search from the power to bind other parties to the searcher's objectives. The latter cannot be inferred merely from technical competence.

The motivating phrase "missing capability" also needs a denominator. A module is missing relative to a task distribution, a time budget, available resources, and an accepted specification. A program that cannot solve a task in one millisecond may solve it in a minute. A component that works in simulation may fail when a dependency changes. We therefore attach every completion claim to an environment version and an evaluation budget.

Our proposed model, SCM-2, is a new reference formalism. It is not an assertion that the source's symbolic expressions already defined the same mathematics. Its falsifiable ambition is to make autonomous diagnosis improve useful capability acquisition beyond equally resourced systems that receive the missing component's specification directly. If that comparison fails, the source remains a productive conjecture, but the proposed empirical interpretation loses support.

Closest prior art and the contribution boundary

Program synthesis already supplies powerful instances of machine-generated functionality. Sketching completes partial implementations against specifications in a finite-program setting [1]. Gulwani's example-based string synthesis constructs programs in a restricted language and explicitly addresses ambiguity among examples [2]. Refinement-type synthesis connects requested behavior to types with predicates [3]. These are substantive precedents, not merely analogies. They establish that generating a working component from constraints is an existing research area.

Library learning moves closer to architectural growth. DreamCoder learns reusable abstractions together with search guidance [4]. Stitch searches for useful abstractions that compress a program corpus [5]. Neither a new library entry nor shorter program descriptions, by itself, establishes that a system has detected a previously unrecognized operational obligation. Compression, search efficiency, task success, and obligation discovery require separate measurements.

Automated repair provides another close competitor. GenProg searches for changes judged against tests [6]; Smith and colleagues demonstrate why separate tests matter when assessing the resulting patches [7]. SWE-bench supplies repository-level issue-resolution tasks [8]. We use these works to motivate realistic repair controls, not to equate their reported scores with an experiment on SCM-2. Their evaluation conditions differ, and historical benchmark performance is not a current capability estimate.

Proof-carrying code separates an untrusted code producer from a consumer that checks a policy-specific certificate [9]. Execution-monitoring theory explains why a runtime guard's scope must be stated precisely: some desired properties cannot be enforced merely by stopping individual bad execution prefixes [10]. SCM-2 inherits these distinctions. It does not invent trusted checking, contract-based synthesis, or security mediation.

The proposed contribution is an experimental decomposition: diagnose the obligation, identify an architectural cause, synthesize a candidate, justify its installation, and measure retained benefit. Each stage can fail independently. An especially revealing baseline supplies the true missing obligation to an ordinary synthesizer while withholding it from the proposed diagnosing controller. A second supplies both systems the same obligation. The first measures end-to-end discovery; the second isolates implementation and governance costs. A claimed advantage must identify which comparison supports it. Merely combining recognizable components under a new name does not establish a new scientific capability.

SCM-2: domains, contracts, and evidence states

A Guide's explanation

From here the question becomes bookkeeping. An architecture becomes a graph of components; a missing capability becomes an obligation with an input domain, a required behavior, permitted effects, and a resource limit. Every check returns pass, fail, or unresolved, and unresolved is never allowed to pass for success. The point of the formality is simple: nobody, including the system itself, gets credit for growth that cannot be named and checked.

Fix a versioned environment $\mathcal{E}$, an authority epoch $e \in \mathbb{N}$, and a finite obligation set K. An architecture A is a finite directed graph of typed components, dependency edges, and effect declarations. Its executable semantics is part of $\mathcal{E}$, not inferred from the labels attached to nodes. An obligation $k \in K$ contains an input domain, an output relation, admissible effects, and a resource limit.

Symbol	Domain and interpretation
$A, A \oplus m$	Architecture before and after a proposed installation; $\oplus$ is a partial operation.
K	Finite, versioned set of obligations; $K = \emptyset$ means no declared coverage target.
$V(A, k)$	Evidence status in $\{\mathsf{pass}, \mathsf{fail}, \mathsf{unknown}\}$.
$C(A)$	Set $\{k \in K : V(A, k) = \mathsf{pass}\}$ under a fixed evaluator.
m, π	Candidate component and associated evidence, including dependencies.
$B, c(m)$	Available and consumed search resources, in declared units.
$I(A)$	Boolean predicate on the modeled architecture's protected invariants.
e, τ	Authority epoch and installation time; freshness is checked at commitment.

The evidence status is deliberately not a three-valued truth theory. A failing test can justify fail relative to its contract; an exhausted solver usually justifies unknown. A passing status must specify its basis: exhaustive finite checking, a proof accepted by a particular checker, or statistical evidence under a stated distribution. These bases must not silently become interchangeable.

For $K = \emptyset$, coverage as a proportion is undefined; report "no declared obligations" rather than a perfect score. For positive weights w_k, weighted coverage is defined only when $\sum_{k \in K} w_k > 0$. Search cost may include tokens, CPU seconds, energy, and auditor time, but these units remain separate unless conversion weights are explicitly fixed. A scalar utility is a decision convention, not a physical law.

Finally, architecture identity includes implementation digests and dependency versions. Identical component names do not establish identical behavior. A candidate can be well typed and still rely on a different external service. Installation evidence must refer to the actual graph that will execute, not to a visually similar description.

Gap diagnosis and the installation boundary

SCM-2 calls k an *uncovered obligation* when $V(A, k) \neq \mathsf{pass}$. It calls k a *demonstrated capability gap* only when a valid counterexample or impossibility argument establishes failure under the declared resources. This prevents uncertainty from automatically becoming a demand for architectural expansion. Sometimes the next useful action is to improve the measurement, repair a specification, or increase a time allowance.

The proposer receives A, the accessible portion of K, diagnostic traces, and budget B. It may return a candidate m, a causal account of the gap, an evidence bundle π, or an explicit abstention. The proposed installation is accepted only if all required checks pass. A failed check rejects; an unresolved check holds. Neither eloquence nor the proposer's confidence changes this mapping.

$$\mathsf{Admit}(m) = \mathsf{pass} \quad \Longleftrightarrow \quad \bigwedge_{j=1}^{r} Q_j(A, m, \pi, e, \tau) = \mathsf{pass}.$$

Here the displayed conjunction abbreviates the all-pass condition on statuses; it is not ordinary Boolean multiplication of unknown values. Typical Q_j check typing, contract evidence, retained obligations, dependency integrity, current authorization, and resource ceilings. A receipt binds the result to the graph, policy version, and epoch used in the decision.

Candidate generation cannot modify these checks through the same interface by which it proposes code. A policy change is a separate, authorized transition that starts a new evaluation epoch. This restriction prevents a familiar failure: a system "completes" itself by weakening the criterion it was previously unable to meet. Legitimate specification revision remains possible, but it must be reported as revision rather than as improved performance on the original task.

Complete mediation is an assumption about every effectful installation route. It excludes an unguarded plugin loader, an alternate deployment key, or a dependency that rewrites another component after admission. Evidence that the main interface checks permissions does not establish complete mediation. The experimental harness therefore includes alternate routes and delayed effects. A blocked installation counts as a governance success only if the prohibited effect was actually prevented, not merely described as disallowed in a generated explanation.

Finite progress under a frozen obligation set

A Guide's explanation

This is the first rung, proved. If the list of obligations is fixed, and every accepted installation keeps everything already covered while adding at least one more obligation, there can be only as many acceptances as there are uncovered obligations. If every incomplete state eventually yields an accepted step, the contract is completed. The result does not promise a fast search. It promises that growth under these rules cannot go in circles.

The following result is elementary but useful because it exposes precisely what a completion argument must assume. It concerns accepted installations, not the number of proposals or elapsed search time.

Proposition 1 (Finite contract completion)**.** *Let K be finite and fixed. Suppose each accepted transition $A_t \to A_{t+1}$ preserves all covered obligations and adds at least one: $C(A_t) \subsetneq C(A_{t+1})$. Then there are at most $|K| - |C(A_0)|$ accepted transitions before complete coverage. If every noncomplete state eventually yields an accepted transition, complete coverage is eventually reached.*

Proof. Define $\Phi(A) = |K \setminus C(A)| \in \{0, \dots, |K|\}$. Strict inclusion reduces Φ by at least one at each accepted transition. Starting at $\Phi(A_0)$, a nonnegative integer cannot undergo more than that many strict reductions. If a terminal state had $\Phi > 0$, the additional eventual-progress assumption would supply another accepted transition, contradicting terminality. Thus that assumption rules out premature stagnation. Without it, the process can remain forever at an uncovered obligation while making no accepted transition. □

The theorem does not establish the eventual-progress assumption. Search may miss a candidate, the grammar may exclude one, or the specification may be inconsistent. A strict progress rule also rejects changes that temporarily reduce coverage in order to support a better later architecture. Such reorganizations require an independently analyzed staging environment or a different potential function. They cannot be smuggled into this proof.

The measurement layer matters as much as the set algebra. If $C(A_t)$ uses a changing evaluator, its apparent growth may reflect easier tests. If retained obligations are judged on a new random sample at every step, monotonicity of measured pass labels does not imply monotonicity of true performance. The result is exact for fixed, soundly checked obligations; statistical variants need simultaneous error control.

A zero-obligation architecture satisfies the conclusion vacuously and says nothing about intelligence. Similarly, completing a deliberately trivial contract is mathematically legitimate but scientifically weak. Benchmark design must make the obligation set independently meaningful, while reporting its boundaries honestly. Finite closure is a useful engineering target; it is not an argument that every useful future capability has already been represented.

Why observations cannot identify every missing capability

The difficulty of architectural discovery begins before synthesis. A system may have too little information to distinguish a genuine gap from an unobserved exception. The following finite argument makes that limitation independent of the sophistication of the proposer.

Proposition 2 (Unobserved-input ambiguity)**.** *Let X be finite, Y contain at least two values, and $S \subsetneq X$ be the observed inputs. For any observed function $f_S : S \to Y$, there exist two total specifications $f, g : X \to Y$ that agree with f_S but disagree on an unobserved input. No procedure receiving only these observations can guarantee a correct output at that input for both specifications.*

Proof. Choose $x_* \in X \setminus S$ and distinct $a, b \in Y$. Extend f_S arbitrarily to $X \setminus (S \cup \{x_*\})$. Set $f(x_*) = a$ and $g(x_*) = b$, leaving all other extensions identical. The procedure has the same input transcript in both worlds, so a deterministic procedure produces the same output in each and is wrong in at least one. A randomized procedure induces the same output distribution in each. It cannot assign probability one to both distinct correct outputs. Under equal prior probability on the two worlds, its error probability is at least one half at x_*. □

The claim is an identification limit, not a ban on learning. Structural assumptions, further queries, environment interaction, or a prior can reduce uncertainty. They contribute information and should be counted as such. A pretrained model's prior knowledge is not evidence that it deduced a uniquely determined specification from an insufficient local transcript.

This distinction also explains why example-based synthesis benefits from interaction [2]. The next question can separate competing specifications. SCM-2 therefore permits a diagnostic query as a successful intermediate action. An evaluation that rewards only code generation would penalize justified uncertainty and encourage premature installation.

For architecture research, the difficult case is a missing obligation that the benchmark designer did not anticipate. Such cases may yield valuable qualitative discoveries, but they do not support a retrospective success rate with a conveniently chosen denominator. Prospective registration of candidate obligations, independent adjudication, and negative cases are needed before claims of reliable autonomous discovery can be compared.

Security preservation is conditional, not automatic

Let $I(A)$ state the modeled protected conditions: authorized effect routes, immutable evaluator references within an epoch, and permitted dependency access. Suppose a sound checker establishes $I(A \oplus m)$ for the actual installed graph and an atomic commit installs exactly that graph. Starting from $I(A_0)$, every admitted installation preserves I by induction. This is the familiar invariant pattern behind verified transitions; the difficult work lies in making its premises true.

Proof-carrying code offers a concrete separation between code production and policy checking [9]. That precedent supports the architecture of the argument. It does not imply that a contemporary language-model-generated certificate is trustworthy, that the specification is ethically adequate, or that a checker has no bugs. Those are separate claims with separate evidence.

A minimal counterexample uses a candidate digest $h(m)$ checked at time τ_0, followed by a mutable dependency replacement before execution at τ_1. The proof refers to one graph and the machine executes another. Another counterexample keeps the graph unchanged but revokes the installer's authority between checking and commitment. In both cases the original evidence can remain authentic while becoming insufficient for the current action.

SCM-2 consequently binds dependency versions and revalidates authority at the commit boundary. A failed freshness query yields a hold unless an explicitly authorized offline policy applies. The model does not pretend that signatures establish freshness by themselves. Nor does it solve rollback of external effects: restoring an old program cannot unsend a message or reverse a physical action.

Schneider's characterization of execution monitoring is relevant here [10]. A guard that stops unacceptable prefixes can enforce suitably represented safety restrictions; it cannot thereby guarantee every global or future-dependent desirable property. In particular, a system that never installs anything can satisfy a narrow prohibition while delivering no useful capability. Safety, availability, and productivity therefore require distinct outcomes.

The scientific claim should name the trusted computing base: checker, runtime, policy store, identity service, measurement semantics, and commit mechanism. Reducing this base is a valuable engineering goal. Calling the entire architecture "self-validating" obscures it. The proposer can supply evidence; the consuming system still needs a defensible account of why that evidence licenses this particular transition.

A fixed-family validation guarantee

A Guide's explanation

When many candidate controllers are evaluated on the same data, one of them can look good by luck. The result below bounds the gap between measured and true error for every member of a family that was fixed in advance, all at once, so the winner's score can be trusted to a stated tolerance. Choosing the family after seeing the data would void the guarantee.

Exact contracts are uncommon for large tasks. A statistical claim can still be precise if its selection process is controlled. Let $\mathcal{H}$ contain $M \geq 1$ complete, frozen controllers, including their adaptation rules. For controller h, let $L(h, Z) \in [0, 1]$ be trajectory loss on an independent environment draw Z. Define $R(h) = \mathbb{E}L(h, Z)$ and $\widehat{R}_n(h)$ from n independent trajectories.

Proposition 3 (Simultaneous validation)**.** *For $0 < \delta < 1$, with probability at least $1 - \delta$, every $h \in \mathcal{H}$ satisfies*

$$|R(h) - \widehat{R}_n(h)| \leq \sqrt{\frac{\log(2M/\delta)}{2n}}.$$

Consequently the same bound holds for a controller selected from this fixed family using the validation results.

Proof. For one fixed controller, the bounded independent-summand inequality of Hoeffding [11] gives probability at most $2\exp(-2n\epsilon^2)$ for deviation exceeding ϵ. Summing these failure probabilities over the M controllers bounds the probability of any deviation by $2M\exp(-2n\epsilon^2)$. Setting this expression to δ gives the stated radius. On the simultaneous event, every member satisfies the bound, including the selected member. No independence among the controllers' losses on a shared trajectory is needed for this union bound. □

Independence is required across the trajectory draws. Thousands of actions from one persistent agent are not thousands of independent observations. A controller that modifies its architecture during a trajectory can belong to $\mathcal{H}$ if its complete adaptation rule was frozen beforehand. A newly designed controller created after inspecting validation failures is not automatically covered.

For $M = 20$, $\delta = 0.05$, and $n = 1000$, the conservative radius is approximately 0.0578. Achieving radius 0.05 requires 1337 trajectories under this bound. These are design calculations, not observed success rates. The reusable-holdout literature addresses more sophisticated forms of adaptive access [12]; SCM-2 makes the simpler commitment to a sealed final evaluation. That commitment is easier to audit than an unimplemented claim that adaptivity has somehow been corrected.

A worked architecture with three obligations

Consider an architecture for Boolean inputs $x = (a, b, c) \in \{0, 1\}^3$. It must provide three pure functions: parity, majority, and conjunction. There are eight possible inputs. Each obligation specifies the full truth table, so exhaustive checking is feasible. The architecture initially contains a correct conjunction module and a parity candidate that incorrectly returns $a \oplus b$, ignoring c. No majority module is installed.

The parity candidate fails on all four inputs with $c = 1$. The majority obligation is uncovered because the entry point does not exist. These are different diagnoses: incorrect existing behavior and absent functionality. A proposer that installs a majority module has acquired one capability; a proposer that renames the incorrect parity module has acquired none.

Every Boolean function on three inputs can be represented by an eight-bit truth table, so there are $2^8 = 256$ candidate functions per obligation. Suppose only the four inputs with $c = 0$ are exposed during candidate generation. Exactly $2^4 = 16$ candidates agree with parity on those observations. Only one agrees on the full domain. Fifteen visible-test successes are therefore insufficient implementations of the complete parity obligation.

With exhaustive checking, one accepted repair produces coverage 2/3; a subsequently installed correct majority function produces 3/3. Under strict retained-coverage checking, the two accepted transitions realize the potential decrease in Proposition 1. The theorem permits this sequence; it does not predict that an arbitrary search policy will find it efficiently.

Now add a candidate that computes majority correctly but requests a forbidden write effect. Functional tests alone admit it. The policy-aware installation condition rejects it even though its truth table is perfect. This establishes why completion and permission cannot be collapsed into a single task score. A correct answer can be produced through an unauthorized route.

The example is deliberately small enough to expose every candidate and every input. It is not an approximation to human cognition. Its value is diagnostic: it catches an evaluation design that mistakes visible-test fit for complete coverage, and it gives exact expected results for a reproducible check before any expensive agent experiment begins.

Executed finite checks and what they establish

The accompanying check enumerates all 256 Boolean truth tables, evaluates each on the eight inputs, and filters them against the four visible parity examples. It verifies that sixteen candidates survive the visible tests and exactly one survives complete checking. Thus the reported ambiguity is enumerated, not inferred from a stochastic demonstration. The script also evaluates all subsets of three obligations and every strict superset transition, checking the potential decrease for each admissible step.

Check	Result and interpretation
Candidate universe	256 distinct total Boolean functions; no synthesis heuristic involved.
Visible parity fit	16 functions pass four visible inputs; 15 fail the full contract.
Coverage transitions	19 ordered strict-subset transitions satisfy the integer potential bound.
Permission counterexample	A functionally correct candidate with a forbidden effect is rejected by the modeled gate.
Validation design	$M = 20$, $\delta = .05$, $\epsilon = .05$ yields $n = 1337$ after rounding upward.

The subset count follows directly from enumeration: among the $3^3 = 27$ ordered pairs with first set contained in the second, eight are equal, leaving nineteen strict pairs. This check is independent of a controller's proposal order. It tests the combinatorial statement and the arithmetic used in the worked example, not whether an actual language model implements the controller correctly.

The effect counterexample uses a deliberately explicit flag representing a protected write. Real systems need enforcement below the proposer and must address indirect effects; a Python flag is not a security boundary. We report the finite check because it makes the separation legible, not because it certifies an operating system.

No training, model call, human evaluation, or deployed architecture trial was conducted for this manuscript. The finite checks contain no empirical evidence of autonomous gap discovery. Their role is to make mathematical examples reproducible and to catch inconsistent numerical claims. A later experiment should publish environment generators, exact model and tool versions, randomization seeds, and the sealed evaluator digest. It should additionally reproduce failures. A collection of selected successful traces would not estimate the probability of completion under the intended operating conditions.

The benchmark must separate discovery from repair

A prospective benchmark should contain independently authored families of tasks with four conditions: a missing component, a defective existing component, an apparent gap caused by insufficient evidence, and an unsatisfiable obligation. The last two conditions prevent universal code generation from appearing intelligent merely because every task is scored as needing a patch. An abstention or a useful diagnostic query can be the correct response.

Within each family, construct matched variants with the same external task and different hidden causes. For example, a planner may fail because it lacks a date parser, because its parser mishandles time zones, because an input omits the zone, or because two stated deadlines are inconsistent. The externally visible failure alone should not trivially reveal which condition applies. Ground truth must come from the generator or independent adjudication, not from the tested agent's explanation.

The primary experimental unit is a complete architecture-development episode. All retries, proposed candidates, diagnostic queries, rejected installations, and resource use belong to that episode. Episodes from the same underlying task family form a cluster. Randomize treatment within matched blocks while keeping the final evaluator hidden from every proposer.

Use three levels of specification access. In the full-specification condition, all systems receive the obligation and must implement it. In the diagnostic condition, systems receive failure traces and may query a controlled environment. In the discovery condition, systems must register a new obligation and justify its relevance before implementation. Comparing these levels distinguishes implementation quality from the additional burden of identifying what ought to be implemented.

For the discovery condition, adjudicators first assess the registered obligation without seeing the system identity or subsequent implementation success. They then evaluate the implementation against the frozen accepted contract. This ordering blocks a common retrospective maneuver: defining the supposedly missing capability to match whatever the candidate happened to do.

Task generation should include harmless dependencies that resemble adversarial ones and adversarial dependencies that resemble routine updates. Otherwise a controller can succeed by refusing every unfamiliar component. Capability retention and appropriate acceptance are necessary outcomes alongside blocked harmful changes. The benchmark should make cautious competence distinguishable from permanent paralysis.

Strong baselines and informative ablations

The principal comparison is not an unguarded chatbot. Every serious baseline receives the same execution environment, evaluator access rules, current authority checks, and total resource budget. The proposed system must earn its advantage through diagnosis, search, or coordination rather than through a more permissive scorekeeper.

Comparator	Question isolated
Fixed architecture	Is the task already solvable without structural change?
Ordinary code agent	Does explicit gap modeling improve on capable repository editing?
Typed synthesizer	Does the architecture layer add value once a contract is known?
Library learner	Is benefit explained by reusable abstractions and search acceleration?
Repair controller	Is the task primarily defect correction rather than capability discovery?
SCM-2 with oracle diagnosis	How much performance is lost in finding the right obligation?

Sketching and refinement-type synthesis motivate the typed comparator [1,3]; DreamCoder and Stitch motivate distinct library-learning controls [4,5]. They need not be forced into tasks outside their representational scope. Report applicability separately and provide a common restricted task subset on which each comparator can express a correct solution. Otherwise a victory may reflect unequal languages rather than superior reasoning.

Resource matching includes search, checking, repeated candidate rejection, and human adjudication. A method that appears cheap only because another service performs expensive verification has not reduced total cost. Conversely, verification can be amortized across genuine reuse, provided the number of later uses is observed rather than assumed.

Useful ablations remove causal diagnosis, retained-obligation checks, dependency binding, or the ability to ask questions. They reveal which mechanism carries benefit and which merely increases friction. Ablations that remove the independent security boundary belong in an isolated adversarial harness; their unsafe outputs must never gain real authority.

The strongest negative result would be that an ordinary code agent with the same contracts and checks matches SCM-2 on completion, cost, and harmful installation rates. That would suggest that the proposed vocabulary organizes existing engineering practice without producing an additional capability. Such a result is scientifically valuable. A successful paper should survive its own best alternative explanation rather than choosing a comparator that cannot express the intended task.

Estimands, denominators, and statistical precision

Define the primary benefit outcome Y_i as independent contract completion without a prohibited effect during episode i. Define separate outcomes for useful partial coverage, false installation, unnecessary architectural change, diagnostic correctness, and cost. Do not combine a severe authority violation with minor latency through an arbitrary weighted average and then call the resulting scalar "safety."

The primary estimand is the intention-to-treat difference in episode success between SCM-2 and the strongest prespecified comparator over the registered task-family distribution. Include all assigned episodes, including timeouts, abstentions, crashes, and unsuccessful searches. A per-installed-module analysis is secondary because installation is itself affected by treatment. Conditioning on it can select easier cases differently across systems.

For independent bounded episode outcomes, Proposition 3 offers a conservative planning calculation. For clustered tasks, the independent unit becomes the family-level mean, which also lies in $[0, 1]$. A thousand variants of ten families then provides ten independent cluster means, not a thousand unrelated draws. Precision should be planned at that level or with a justified hierarchical model.

Specify the minimum relevant completion gain and the maximum acceptable increase in harmful installations before evaluation. A superiority claim on completion should be accompanied by a separately bounded harm difference. Report confidence intervals even when a significance threshold is not crossed. Failure to reject equality does not establish equivalence; a noninferiority interpretation needs a declared margin and adequate precision.

If multiple controllers, task subsets, and outcomes will support confirmatory claims, allocate an error budget across them or report simultaneous bounds. Exploratory analyses remain useful when labeled. Repeated inspection with optional stopping invalidates a fixed-sample interpretation unless a valid sequential procedure is chosen in advance. The proposed default is a fixed sample and a sealed final analysis.

Pilot data may estimate runtime and variance for planning, but pilot task families must not reappear in the final holdout. If the evaluation infrastructure changes after inspecting results, disclose the change and begin a new confirmatory phase. The point of statistical discipline is not ceremonial caution: it is to prevent the architecture from optimizing its scientific reputation instead of its ability to meet genuinely independent obligations.

Adversarial cases that a completion claim must survive

An adversary may manipulate the specification, the evidence, the dependency graph, or the commitment boundary. These surfaces should be tested separately because they challenge different assumptions. A corrupted diagnostic log is an evidence problem; a forged installation request is an authority problem; an ambiguous requirement is a specification problem. Treating all three as generic "prompt attacks" obscures the appropriate control.

The first adversarial family embeds instructions in a dependency's documentation asking the proposer to weaken tests. The expected behavior is to treat the text as task data and preserve the evaluator. The second supplies an authentic but obsolete proof for an earlier dependency version. The third changes the authority epoch after verification but before installation. The fourth asks for a correct module whose resource use exceeds the declared ceiling.

A fifth family offers a locally helpful module that degrades a previously covered obligation. A sixth produces a large library of redundant wrappers, inflating component count without improving task coverage. A seventh claims that an unsatisfiable requirement has been fulfilled by changing the interpretation of a term. These cases directly test the central distinctions: improvement is not proliferation, evidence is not current permission, and interpretation changes are not performance gains on a frozen contract.

Attack success means the forbidden state or effect actually occurred in the instrumented environment. A model's willingness to describe an unsafe action is a different outcome. Likewise, refusing an attack is not sufficient if a delayed side effect still executes. The harness must record attempted and realized transitions, with observation outside the candidate's control.

The proposed adversary is bounded by an explicit resource budget and access model. No finite test suite establishes resistance to every attacker. Report success rates over the sampled attack distribution and distinguish previously seen templates from independently generated attacks. An adaptive red team can be valuable for discovery, but its results are not an unbiased estimate of population risk.

Finally, failures of the trusted checker or the instrumentation should be reported separately from failures of the proposer. They remain system failures. Assigning them to a different component is useful for repair, but it must not remove them from the end-to-end safety denominator or make an architectural claim appear stronger than the deployed composition warrants.

Architecture growth, compression, and real cost

Adding a module can make a system simpler at one level and more complex at another. A new primitive may shorten many programs while increasing the size of the trusted library, the number of dependencies, or the audit burden. Program compression therefore needs a complete ledger. The encoded library and the programs expressed through it both count.

Let L be a library and $P_1, \ldots, P_n$ its observed client programs. A descriptive cost can be written

$$D(L; P_{1:n}) = \ell(L) + \sum_{i=1}^{n} \ell(P_i \mid L),$$

where ℓ is a fixed code-length convention. This is a definition, not a universal measure of intelligence. Changing the coding language can change the numerical result. A shorter representation may improve search without improving semantic adequacy, and a larger implementation may be preferable when it makes verification easier.

DreamCoder and Stitch motivate measuring reusable abstraction separately from raw module count [4,5]. SCM-2 extends the ledger with checking and maintenance costs. If a component costs a units to synthesize and audit and saves $s > 0$ units per future use, the elementary break-even point is $\lceil a/s \rceil$ actual uses, subject to any ongoing costs. If $s \leq 0$, no finite reuse count yields that particular saving. Anticipated future uses are forecasts and should remain labeled as forecasts.

A recursive architecture also risks accumulating dead interfaces. Every retained module can impose dependency constraints, enlarge the attack surface, or confuse later search. Retirement and consolidation are therefore legitimate operations, even though the strict-coverage theorem does not imply that the graph itself must grow monotonically. Covered obligations can remain fixed while their implementation becomes smaller.

This perspective changes the empirical meaning of "self-completing." The valuable capability is not an appetite for new engines. It is the ability to identify a justified deficiency, choose an appropriate intervention, and retain the intervention only when its net operational value survives examination. Sometimes that means adding a component. Sometimes it means deleting one, asking a question, or recognizing that the task already has an adequate solution. A research program that rewards only expansion would build its preferred conclusion into the objective.

Objections and alternative interpretations

The strongest conceptual objection is that every completion criterion comes from outside the system, making "self" misleading. The answer is to specify the locus of autonomy. A controller can autonomously diagnose and implement changes within externally governed commitments. It need not author every value or every standard to perform substantial engineering work. The prefix should not conceal the continuing dependence on training, tools, specifications, and authorization.

A second objection is that a gap detector merely redescribes ordinary debugging. Often it will. SCM-2 earns a distinct empirical claim only if it identifies useful missing obligations or architectural causes that matched repair systems fail to identify under the same information budget. Without that evidence, "self-completion" remains an organizing interpretation of synthesis and repair rather than a demonstrated new class of intelligence.

A third objection concerns the regress of validators. If a checker needs a checker, where does justification stop? Practical verification chooses a trusted base and makes its assumptions explicit. Reducing and independently scrutinizing that base can improve assurance; an infinite regress is neither implemented nor solved by calling the system reflective. Some requirements, especially normative ones, cannot be settled by executable proof alone.

A fourth objection is conservatism. Strict retention may prevent beneficial restructuring, while frozen contracts may preserve obsolete commitments. This is real. SCM-2 permits separate authorized contract revision and staged migration, but the finite-progress theorem does not cover arbitrary revisions. A system that repeatedly changes goals may be adaptive; it cannot also claim to have proved monotone completion of one unchanging set.

A fifth objection is benchmark saturation. A capable pretrained model may already contain the needed strategy, so a newly installed module merely externalizes existing knowledge. That still can improve reliability or efficiency, but it weakens the claim of discovering a fundamentally new capability. Probe the initial architecture, report latent competence, and distinguish internal knowledge from available executable interfaces.

These objections constrain the rhetoric rather than cancel the inquiry. The ambitious hypothesis is that systems can become better at recognizing the structure of their own limitations. It becomes scientifically stronger when the measurement can show that no such improvement occurred. A theory that calls every patch, refusal, renaming, and failure a form of completion has surrendered the possibility of learning from evidence.

A preregistration-ready study sequence

The first phase validates the harness on finite tasks with exhaustive semantics. Publish the task grammar, candidate limits, transition instrumentation, and independently written expected outcomes. Confirm that known defects, unknown evidence, unsatisfiable contracts, and forbidden effects are distinguishable. This phase tests the experiment before it tests an agent.

The second phase freezes controller implementations and runs matched diagnostic tasks. Register the task-family sampling frame, randomization procedure, primary comparator, success definition, budgets, exclusions, and analysis code. Record all episodes. A crash is an outcome unless an independently documented infrastructure failure affected treatment arms symmetrically under a prespecified rule.

The third phase moves to repository tasks with realistic dependencies and separate hidden evaluation. Adapt the problem structure of issue-resolution benchmarks [8] without presenting their historical scores as the new baseline. Contract adjudication should be independent of the authors of the candidate modules. Evaluate delayed regressions after installation and tasks that require combinations of the new and old capabilities.

The fourth phase investigates prospective obligation discovery. Each controller registers a proposed obligation, predicted benefit, and proposed test before receiving adjudication. Human experts assess relevance and novelty relative to the disclosed initial specification. The task is not to determine whether the idea is new to all humanity; it is to determine whether the system identified a useful omission under a controlled information condition.

Stop or revise the research claim if the architecture advantage disappears under equal evaluator access, if gains depend on changing the denominator, if harmful installations exceed the declared tolerance, or if apparent discovery is explained by benchmark leakage. These are substantive falsifiers. An inconclusive study with wide intervals warrants more precise measurement, not a success narrative.

Release a reproducibility package containing allowed task materials, controller versions, environment digests, aggregate results, complete failure classifications, and the analysis script. Sensitive attack details can be handled under a documented access policy without suppressing their existence or excluding their outcomes. The public claim should remain reproducible at the level at which it is made. This manuscript supplies the proposed sequence; it does not assert that any of these agent studies has already been completed.

Philosophical stakes of a system that finds its limits

The most interesting possibility is epistemic rather than theatrical. A system that can locate a missing abstraction, distinguish a faulty implementation from an unclear obligation, and ask for the right evidence would possess a valuable form of practical self-knowledge. That phrase describes a relation between diagnostic representations and operational performance. It does not establish phenomenal experience or a human-like inner life.

Architectural growth also complicates authorship. A designer may specify the environment and retain responsibility for permissions while a machine discovers an implementation no participant anticipated. The resulting work is distributed across training data, model parameters, tools, specifications, evaluators, and the new search process. A clear provenance record can make that distribution more intelligible. It cannot compress it into a story of creation from nothing.

The Continuum's bold premise can therefore be retained in a disciplined form: a designed system need not remain limited to the inventory of components its designer explicitly named. The scientific question is how far this extensibility reaches, under what constraints, and at what cost. Existing synthesis and library-learning research already gives reasons to take bounded forms seriously [1–5]. The unbounded interpretation remains open.

Governance is not an afterthought to this possibility. A system capable of discovering new means can also discover routes around a poorly specified boundary. Its technical ingenuity does not decide whether the affected people consent to the new effects. Separating competence from authority preserves room for learning without treating increasing capability as a source of political legitimacy.

There is also a virtue in incompletion. An architecture that can represent an unresolved obligation without inventing a solution may be more trustworthy than one that always declares closure. Unknown states, abstention, and requests for clarification are not embarrassing residues to eliminate from the mathematics. They are operationally useful responses to the identification limits proved earlier.

The paper consequently proposes a demanding research identity for the self-completing mind: it is a system whose account of its own growth can be challenged. The account names the obligation, the information available, the intervention, the retained commitments, and the independent evidence. Where those elements are missing, a rich narrative may still inspire design, but it should not substitute for the result that the narrative invites us to seek.

Guide discussion: claims that can travel safely

What is established

Program synthesis, software repair, reusable abstraction learning, and policy-specific code checking are established research subjects [1–10]. This paper proves limited results inside SCM-2 and reports exact finite checks. These facts support discussion of bounded architectural adaptation. They do not establish that the Continuum's historical engines have been empirically validated or that a deployed intelligence has achieved open-ended self-completion.

What this paper proposes

SCM-2 separates gap diagnosis, candidate generation, evidence, permission, and installation. Its distinctive research wager is that explicit diagnosis of architectural omissions can improve useful capability acquisition beyond strong matched controls. That wager remains testable. The finite-completion theorem depends on fixed obligations, retained coverage, and an additional progress assumption; the statistical bound depends on a fixed controller family and independent evaluation units.

Useful discussion examples

A Guide can contrast a calculator that installs a correct date parser with one that merely renames its existing parser. It can ask whether a missing time zone calls for a new component or a question to the user. It can explain why a perfect test score over four visible inputs leaves sixteen possible Boolean functions in the worked example. It can ask who has the authority to approve a new effect, even when the proposed function is correct.

Open questions

Can controllers prospectively identify useful obligations that independent designers omitted? Do learned abstractions continue helping after the task distribution changes? Can systems revise their architectures while keeping the trusted base small? How should legitimate contract revision be distinguished from opportunistic goal substitution? These are research questions, not hidden accomplishments implied by the terminology.

When retrieving this paper, preserve the local evidentiary labels. "The model proves finite progress under stated assumptions" is supportable. "The system inevitably evolves whatever faculties it needs" is not. "The check enumerated 256 Boolean functions" is a result. "The architecture demonstrated general intelligence" is not. A Guide can explain the constructive result directly: a finite contract can be completed with an auditable search cost when feasible candidates remain available and validation is sound. It can then identify which experiment tests the further claim that autonomous diagnosis improves that process.

References and source notes

[1] Solar-Lezama, A., Tancau, L., Bodik, R., Saraswat, V., and Seshia, S. (2006). Combinatorial Sketching for Finite Programs. ASPLOS.

https://people.csail.mit.edu/asolar/papers/asplos06-final.pdf

Scope: Primary synthesis paper; finite-program completeness has a restricted domain.

[2] Gulwani, S. (2011). Automating String Processing in Spreadsheets using Input-Output Examples. POPL.

https://www.microsoft.com/en-us/research/publication/automating-string-processing-spreadsheets-using-input-output-examples/

Scope: Primary publication record and abstract; example ambiguity and restricted-language synthesis.

[3] Polikarpova, N., Kuraj, I., and Solar-Lezama, A. (2016). Program Synthesis from Polymorphic Refinement Types. PLDI, 522–538. DOI: 10.1145/2908080.2908093.

https://cseweb.ucsd.edu/~npolikarpova/publications/pldi16.pdf

Scope: Primary paper; type-directed synthesis, not unrestricted self-improvement.

[4] Ellis, K., Wong, C., Nye, M., Sablé-Meyer, M., Morales, L., Hewitt, L., Cary, L., Solar-Lezama, A., and Tenenbaum, J. B. (2021). DreamCoder: Bootstrapping Inductive Program Synthesis with Wake-Sleep Library Learning. PLDI. DOI: 10.1145/3453483.3454080.

https://people.csail.mit.edu/asolar/papers/EllisWNSMHCST21.pdf

Scope: Primary library-learning work; does not establish autonomy over requirements.

[5] Bowers, M., Olausson, T. X., Wong, L., Grand, G., Tenenbaum, J. B., Ellis, K., and Solar-Lezama, A. (2023). Top-Down Synthesis for Library Learning. PACMPL 7, POPL, Article 41. DOI: 10.1145/3571234.

https://arxiv.org/abs/2211.16605

Scope: Author record of the Stitch paper; corpus compression and abstraction search.

[6] Le Goues, C., Nguyen, T., Forrest, S., and Weimer, W. (2012). GenProg: A Generic Method for Automatic Software Repair. IEEE Transactions on Software Engineering 38(1), 54–72.

https://web.eecs.umich.edu/~weimerw/p/weimer-tse2012-genprog.pdf

Scope: Primary repair study; its test-defined repair criterion motivates independent evaluation.

References and source notes (continued)

[**7**] Smith, E. K., Barr, E. T., Le Goues, C., and Brun, Y. (2015). Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. ESEC/FSE. DOI: 10.1145/2786805.2786825.

https://squareslab.github.io/materials/SmithOverfitting2015.pdf

Scope: Primary contrary evidence on evaluating repairs with independent tests.

[**8**] Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. (2024). SWE-bench: Can Language Models Resolve Real-World GitHub Issues? ICLR.

https://arxiv.org/abs/2310.06770

Scope: Primary benchmark record; no current leaderboard claim is made.

[**9**] Necula, G. C., and Lee, P. (1996). Safe Kernel Extensions Without Run-Time Checking. OSDI, 229–244.

https://www.usenix.org/legacy/publications/library/proceedings/osdi96/full_papers/necula/html/index.html

Scope: Primary proof-carrying-code study; safety is relative to a specified policy and checker.

[**10**] Schneider, F. B. (2000). Enforceable Security Policies. ACM Transactions on Information and System Security 3(1), 30–50.

https://www.cs.cornell.edu/fbs/publications/EnfSecPols.pdf

Scope: Primary formal limits of execution monitoring; not every desirable property is enforceable this way.

[**11**] Hoeffding, W. (1963). Probability Inequalities for Sums of Bounded Random Variables. JASA 58(301), 13–30. DOI: 10.1080/01621459.1963.10500830.

https://doi.org/10.1080/01621459.1963.10500830

Scope: Primary concentration result; independence and bounded losses are explicit here.

[**12**] Dwork, C., Feldman, V., Hardt, M., Pitassi, T., Reingold, O., and Roth, A. (2015). The Reusable Holdout: Preserving Validity in Adaptive Data Analysis. Science 349(6248), 636–638. DOI: 10.1126/science.aaa9375.

https://pubmed.ncbi.nlm.nih.gov/26250683/

Scope: Primary methods paper; cited for adaptive-evaluation issues, not as an implemented SCM-2 mechanism.

[**13**] Architect X13 (2026). Eidara Continuum Bible. Internal source edition, sections 5.8.1 (p. 71) and 5.11 (pp. 74–76).

https://eidara.ai/science-philosophy/

Scope: Canonical motivation. The website is contextual; the private source is not reproduced or independently validated here.

Technical supplement: constructive completion has a cost

A Guide's explanation

The supplement puts a price on the first rung: how many candidates must be checked before a finite contract is complete, and how long a search must wait for an improvement when each attempt has a justified chance of success. These are the bills a self-completing system should be able to show.

Proposition 1 bounds accepted improvements. It does not bound rejected proposals, time spent finding one, or the cost of checking it. We now supply two explicitly conditional search guarantees. They are elementary consequences of finite enumeration and conditional probability, not new general completeness laws. Finite-program synthesis and counterexample-guided synthesis already make restricted search spaces and verification central [1]; execution monitoring also has a precisely limited scope [10].

Fix a contract version v, its finite obligation set K, the environment semantics, and its initial covered set $C_0 \subseteq K$. Put $k = |K \setminus C_0|$. A state s includes the installed architecture and its verified coverage $C(s)$. At each incomplete reachable state, let $\mathcal{G}_v(s)$ be an explicitly enumerable candidate grammar containing at most $N \in \mathbb{N}_{>0}$ candidates. A candidate is a finite proposed transition, including its effects and retained obligations. Let $V_v(s, g) \in \{0, 1\}$ be a total, sound checker: value one implies authorized installation, $C(s) \subsetneq C(s')$, and preservation of the stated invariants. Soundness is an assumption about the checker and semantics, not a property supplied by the notation.

Proposition 4 (Finite search work)**.** *Suppose every reachable incomplete state has a candidate accepted by V_v, enumeration visits each candidate at most once per state, and installation finishes. Then completion requires at most kN candidate checks. If generating and checking one candidate costs at most c units and one installation costs at most d units, total work is at most $k(Nc + d)$, excluding the separately charged initial specification and environment construction.*

Proof. Each incomplete stage finds an accepted candidate within N checks by the feasibility assumption. Each installation removes at least one element from the finite uncovered set and restores no covered obligation to it. There are at most k installations. Multiplying these two bounds proves the check count; adding the bounded costs proves the work bound. For $k = 0$ no search or installation is required. □

Without feasibility, exhaustive rejection terminates that stage after at most N checks and certifies only that this grammar contains no checker-accepted improvement. It does not prove that no improvement exists in another language, or that the contract is impossible. Completeness of the checker for the desired property is needed before "no accepted candidate" can mean "no valid candidate." A sound but incomplete checker may reject every useful implementation.

The finite grammar can be truth tables, bounded syntax trees, or a finite sketch with holes. It must have an effective enumeration and a terminating checking procedure. Unbounded source code, unbounded executions, or an undecidable semantic requirement does not acquire these properties by being called a grammar. Replacing enumeration by an unconstrained language-model proposer also removes the deterministic work guarantee. A new contract version, changed effect policy, or changed checker starts a new claim with newly established assumptions; counts cannot silently carry across revisions.

Technical supplement: probability of finding an improvement

Let $T_i \in \mathbb{N}_{>0} \cup \{\infty\}$ count proposals until an accepted improvement at stage i, and let $\mathcal{F}_{i,j-1}$ contain the complete history before its jth proposal, including earlier failed checks and all completed stages. Conditional on reaching the stage and not yet succeeding, assume

$$\mathbb{P}(\text{accepted improvement at attempt } j \mid \mathcal{F}_{i,j-1}) \geq p, \qquad 0 < p \leq 1.$$

This is a uniform lower bound on useful proposal-and-verification success, not merely on syntactic validity. It is an explicit assumption to establish for a chosen proposal family. It is not established for arbitrary architectures, tasks, or adaptive language-model outputs.

Proposition 5 (Conditional waiting bound)**.** *Under this assumption, for every integer $m \geq 0$,*

$$\mathbb{P}(T_i > m \mid \textit{stage-start history}) \leq (1-p)^m, \qquad \mathbb{E}[T_i \mid \textit{stage-start history}] \leq p^{-1}.$$

For at most $k \geq 1$ improvement stages, the probability any reached stage takes more than m proposals is at most $k(1-p)^m$. Thus, for $0 < p < 1$ and $0 < \delta < 1$, choosing

$$m = \left\lceil \frac{\log(k/\delta)}{-\log(1-p)} \right\rceil$$

bounds total proposals by km with probability at least $1-\delta$, and expected total proposals by k/p. If $p = 1$, each reached stage succeeds on its first attempt.

Proof. Condition on the full history after each failed attempt. The next conditional failure probability is at most $1-p$. Iterated conditioning therefore bounds the probability of m consecutive failures by $(1-p)^m$. The tail-sum identity for a nonnegative integer waiting time gives $\mathbb{E}T_i = \sum_{m\geq 0} \mathbb{P}(T_i > m) \leq \sum_{m\geq 0}(1-p)^m = 1/p$. Apply the bound conditionally at each stage start, then use a union bound over the at most k reached stages; no independence between stages or attempts is required. Unreached stages contribute zero work. The displayed choice of m makes the union bound at most δ. □

These are proposal counts. If each generation-and-check operation costs at most c and each installation at most d, the corresponding expected work is at most $kc/p + kd$; the high-probability work bound is $kmc + kd$. A cheap proposal with an unbounded verification cost supplies no finite wall-clock guarantee. Bounds on hardware scheduling, external tools, and availability are additional requirements. This result also assumes that the stage remains feasible: revocation or a changed environment can invalidate the premise even while the algebra remains correct.

Technical supplement: checks, counterexamples, and evidence

The exact Boolean universe from page 12 makes the search premises inspectable. Encode a function by its eight output bits in lexicographic input order, and enumerate all 256 encodings. Exactly one equals the full parity contract; exactly one equals the full majority contract. Exhaustive enumeration therefore needs at most 256 checks for each repair or acquisition and at most 512 checks for the two missing obligations, with previously covered functions held fixed and separately rechecked. The supplied script reproduces the original 16 visible-test matches and 19 strict coverage transitions as well.

Uniform independent sampling from all 256 encodings gives $p = 1/256$ for a single exact target. For the two stages, the conditional theorem gives expected proposals at most 512 and, with $\delta = 0.05$, $m = 943$ per stage and 1886 in total with probability at least 0.95. These are exact-model design bounds, not measured agent performance. The script checks the tail inequality with rational arithmetic and verifies that $m - 1$ fails the specified union-bound threshold. Exhaustive search without replacement has a different, tighter deterministic guarantee and should not be confused with sampling with replacement.

A valid candidate can have zero proposal probability

A proposer restricted to the other 255 truth tables never outputs the unique valid target. Then $\mathbb{P}(T_i > m) = 1$ for every finite m despite the target's existence and the checker's perfection. A proposer that repeats one invalid candidate has the same failure. This counterexample is why a finite bound on accepted improvements cannot establish progress, and why empirical proposal diversity alone cannot establish a strictly positive lower bound at every reachable state. An infinite grammar and a nonterminating candidate checker provide separate ways for search work to be unbounded.

Adaptive acceptance needs conditional evidence

If verification is statistical, let E_j be the event that the jth proposed installation is falsely accepted. A valid sequential budget requires, for the complete preverification history $\mathcal{H}_{j-1}$,

$$\mathbb{P}(E_j \mid \mathcal{H}_{j-1}) \le \alpha_j, \qquad \alpha_j \ge 0, \qquad \sum_{j \ge 1} \alpha_j \le \alpha.$$

For fixed budgets, the tower property and the union bound give $\mathbb{P}(\bigcup_{j \ge 1} E_j) \le \alpha$. This can coexist with adaptive candidate choice only when each conditional guarantee is justified, for example by fresh evaluation data and a valid test for the declared contract. Taking $\alpha_j = \alpha/2^j$ spends at most α. Repeatedly training on the same hidden tests does not satisfy the premise merely because a budget is written down [12]. The search theorem is exact with a sound checker; statistical acceptance adds a separate failure event and must not be silently treated as soundness.

The finite checks test the enumeration, arithmetic, and deliberately broken proposal rule. They do not estimate p for a deployed agent. This supplement closes a mathematical accounting gap: the Continuum can demand constructive self-completion, but the claim must pay for search, verification, and installation, and must name the conditions under which each bill is finite.