

Autopoietic Code Rewriting inside Isolated Execution Environments

Stability Conditions, Structural Invariants, and Controlled Runtime Transformation

Samir Baladi

Independent Interdisciplinary Researcher — Ronin Institute / Rite of Renaissance

Research Area: Computer Science and Software Engineering

Research Domain: Dynamic Software Systems, Program Transformation, and Runtime Verification

Project Codename: AUTOPOIETIC · Record ID: AUTO-PROTO-01

Version 0.2.0-implementation · Phase I complete. No pilot executed.

DOI: <https://doi.org/10.5281/zenodo.23475104>

ORCID: 0009-0003-8903-0029 · gitdeeper11@gmail.com

October 2026

Abstract

Autonomous software modification introduces a fundamental engineering challenge: how can a running software system modify its own program representation without compromising structural integrity, behavioral correctness, or operational continuity? This paper investigates controlled runtime code rewriting through Abstract Syntax Tree (AST) transformations performed within isolated execution environments. Rather than treating self-modification as an unrestricted capability, the proposed approach defines it as a constrained transformation process governed by explicit syntactic, structural, and behavioral invariants. The research introduces a two-phase rewriting architecture in which candidate program modifications are generated and evaluated independently before being considered for activation. The architecture combines source-code extraction, dependency analysis, controlled transformation, isolated execution, differential behavioral testing, invariant verification, atomic deployment, and rollback mechanisms. The initial experimental scope is deliberately restricted to individual top-level Python functions using Python 3.11 or later and the LibCST transformation framework. Four controlled transformation classes are considered: constant folding, algebraic simplification, dead branch elimination, and local variable renaming. The initial execution model uses stateless functions and multiprocessing-based logical isolation. A central contribution is the explicit separation of four operational properties: execution isolation, transformation validity, deployment integrity, and recoverability. Recoverability is treated independently from rollback success through the introduction of State Restoration Fidelity as a separate evaluation dimension. The paper establishes a reproducible pilot-study protocol for examining transformation success, verification effectiveness, runtime overhead, false acceptance, false rejection, rollback behavior, and service continuity. It also defines measurable transition criteria for extending the experimental scope from function-level rewriting to module-level transformations. The proposed architecture does not assume that syntactic validity guarantees semantic equivalence or that differential testing establishes universal program correctness. Instead, it establishes a layered verification strategy whose effectiveness can be evaluated empirically. As of the present version (0.2.0-implementation), Phase I is complete: all eight core, invariant, sandbox, and deployment modules are implemented and pass their full test suite (118/118). No pilot study has yet been executed and no empirical results are reported.

Keywords: *Autonomous Software Modification; Runtime Code Rewriting; Abstract Syntax Trees; Program Transformation; Structural Invariants; Differential Testing; Runtime Verification; Software Reliability; Dynamic Software Updating; Recoverability.*

Contents

Abstract	1
1. Introduction	4
2. Related Research and Theoretical Foundations	6
3. Formal Problem Definition	9
4. Proposed Architecture	11
5. Transformation Classes	13
6. Invariant Specification	15
7. Isolation, Deployment, and Recoverability	16
8. Experimental Methodology	18
9. Evaluation Metrics	20
10. Hypotheses and Experimental Acceptance Criteria	21
11. Transition Criteria: Function-Level to Module-Level	21
12. Four-Week Pilot Implementation Plan	22
13. Threats to Validity	24
14. Discussion	24
15. Limitations and Future Research	26
16. Implementation Status (Phase I)	27
17. Conclusion	29
References	30
Current Manuscript Status	30

Ronin Institute / Rite of Renaissance · gitdeeper11@gmail.com · ORCID: 0009-0003-8903-0029

1. Introduction

1.1 Background

Modern software systems increasingly incorporate mechanisms that modify their behavior during execution. Dynamic configuration, hot deployment, runtime patching, adaptive execution, and autonomous software agents represent different manifestations of this broader capability.

However, modifying an executing program introduces risks that are fundamentally different from those associated with conventional development and deployment.

In a traditional development process, modifications are generally constructed, tested, and deployed through controlled development pipelines. Runtime code rewriting changes this relationship by allowing the modification process to occur while the target system remains operational.

The principal engineering challenge is therefore not simply whether a program can modify its own representation, but whether the modification can be introduced under explicit conditions that preserve required system properties.

This research addresses that challenge through a constrained model of autonomous code rewriting.

1.2 Research Problem

A program transformation may produce syntactically valid source code while introducing structural inconsistencies or behavioral regressions.

Similarly, a candidate modification may pass isolated tests but fail when activated within the original execution environment.

These observations establish that transformation correctness cannot be represented by a single validation operation.

The research problem is formulated as follows:

Under what formal and operational conditions can a software system transform an individual executable function during runtime while preserving specified structural invariants, observable behavioral properties, and deployment continuity?

This problem contains four distinct dimensions:

- 1 Isolation of the candidate execution.
- 2 Verification of the transformed program.
- 3 Integrity of the transition between program versions.
- 4 Recoverability following an unsuccessful transformation.

These dimensions must be independently specified and evaluated.

1.3 Research Objectives

The primary objective is to design and evaluate a controlled runtime rewriting architecture for individual software functions. The specific objectives are:

- Define a formal representation of admissible program transformations.

- Establish three distinct categories of program invariants.
- Implement controlled AST transformations.
- Introduce isolated candidate execution.
- Evaluate behavioral compatibility through differential testing.
- Design a controlled deployment and rollback mechanism.
- Measure recovery fidelity independently from rollback success.
- Establish explicit experimental conditions for extending the architecture to larger software units.

1.4 Research Questions

The study addresses the following research questions.

RQ1: Under what conditions can AST transformations preserve syntactic and structural validity during runtime modification?

RQ2: To what extent can isolated candidate execution reduce the propagation of transformation failures into the active program environment?

RQ3: How effectively can differential testing identify behavioral deviations between original and transformed functions?

RQ4: What operational overhead is introduced by the proposed verification and deployment architecture?

RQ5: Can rollback mechanisms restore the previous execution state with complete fidelity under controlled failure conditions?

RQ6: Which experimentally measurable conditions justify extending the transformation scope from individual functions to complete software modules?

1.5 Research Contributions

The paper proposes five principal contributions.

- 1 A layered invariant model separating syntactic, structural, and behavioral validity.
- 2 A two-phase runtime rewriting architecture based on candidate transformation and controlled activation.
- 3 A verification strategy combining structural analysis and differential behavioral testing.
- 4 A four-dimensional operational model separating isolation, verification, transition integrity, and recoverability.
- 5 A reproducible experimental protocol with explicit criteria for progression to larger transformation scopes.

The contributions are architectural and methodological at this stage. Their empirical effectiveness remains subject to implementation and experimental evaluation.

2. Related Research and Theoretical Foundations

2.1 Program Representation and AST Transformation

An Abstract Syntax Tree provides a structured representation of the syntactic organization of a program.

Unlike direct textual manipulation, AST-based transformation operates on program constructs such as function definitions, expressions, assignments, conditions, and control-flow statements.

This representation enables transformations to be expressed as operations on structured program elements.

However, syntactic representation alone does not encode every relevant semantic property. Two structurally different programs may exhibit equivalent observable behavior, while two structurally similar programs may produce different results.

Consequently, AST transformation requires complementary verification mechanisms.

LibCST is selected for the initial implementation because it supports concrete syntax preservation, including comments and formatting, while providing structured program transformation facilities.

2.2 Formal Verification and Program Invariants

Program invariants provide conditions that must remain valid across specified execution or transformation boundaries.

Hoare logic offers a formal foundation for expressing program correctness through triples:

$$\{ P \} \ C \ \{ Q \}$$

where P is a precondition, C is a program operation, and Q is a postcondition.

In the proposed architecture, invariants are organized into three categories.

2.2.1 Syntactic Invariants

Syntactic invariants concern the formal validity of the transformed program representation. Examples include:

- Successful parsing.
- Valid function definitions.
- Correct expression structures.
- Absence of malformed AST nodes.
- Preservation of required source-level constraints.

2.2.2 Structural Invariants

Structural invariants concern the relationships between program elements. Examples include:

- Preservation of the target function identity.
- Preservation of the function signature.
- Preservation of required symbol references.

- Absence of unresolved dependencies.
- Preservation of permitted scope relationships.

Not every dependency graph must be acyclic. Recursive programs may contain legitimate cycles. Structural validation must therefore distinguish permitted recursive relationships from invalid dependency references.

2.2.3 Behavioral Invariants

Behavioral invariants concern observable program properties. They may include:

- Return-value consistency.
- Exception behavior.
- State changes.
- Side effects.
- Execution traces.
- Specified resource or latency constraints.

Behavioral invariants require an explicit definition of the observation model.

2.3 Differential Testing

Differential testing compares multiple implementations or program versions using equivalent inputs.

For the proposed research, the original function and its transformed counterpart are evaluated against the same input set. Let $f_0(x)$ represent the original function and $f_1(x)$ represent the transformed function.

For an observation function O , empirical compatibility is defined as:

$$O(f_0, x) = O(f_1, x)$$

for all tested inputs x in a specified test set X .

This establishes empirical observational equivalence over X , not universal semantic equivalence.

Differential testing is therefore treated as an empirical verification mechanism rather than a formal proof of program equivalence.

2.4 Dynamic Software Updating

Dynamic software updating concerns the modification of software components without requiring conventional full-system shutdown and restart.

The proposed research shares this general objective but focuses specifically on constrained function-level transformations.

The distinction is important: successful transformation of an isolated function does not establish the correctness of module-wide or system-wide modification.

2.5 Process Isolation

Operating-system processes provide separate execution contexts and address spaces.

Multiprocessing is therefore suitable for the initial experimental model because candidate execution can be separated from the parent process that maintains the original program representation.

However, process isolation is not equivalent to a hardened security sandbox.

The initial study evaluates logical execution isolation under controlled conditions. It does not claim resistance to malicious code, operating-system compromise, or unrestricted external side effects.

3. Formal Problem Definition

3.1 Program Model

Let the original program representation be:

$$P_0 = (A_0, D_0, S_0)$$

where A_0 represents the original abstract syntax structure, D_0 represents its relevant dependency relationships, and S_0 represents the source-level representation.

A transformation operator T , parameterized by θ , produces a candidate program:

$$P_1 = T(P_0, \theta)$$

The candidate is not immediately activated. Instead, it passes through a sequence of verification conditions.

3.2 Admissibility Conditions

The transformation is admissible only if the following conditions are satisfied:

$$A(P_1) = \text{True} \quad D(P_1) = \text{True} \quad B(P_0, P_1, X) = \text{True}$$

where A represents syntactic validity, D represents structural validity, and B represents behavioral compatibility over the test set X .

The final deployment condition is:

$$\text{Deploy}(P_1) = A(P_1) \wedge D(P_1) \wedge B(P_0, P_1, X)$$

This expression represents the proposed experimental acceptance policy. It does not establish that every accepted transformation is universally correct.

3.3 Observation Model

The observation model is defined as:

$$O(f, x) = (R, E, \Delta S, \Sigma)$$

where R is the return value; E is the exception type and relevant exception properties; ΔS is observable state changes; and Σ is observable side effects or execution trace.

The initial stateless experimental scope restricts functions to cases where relevant state changes and external side effects can be excluded or explicitly controlled.

Latency and memory consumption are measured separately as performance characteristics rather than automatically treated as semantic equivalence conditions.

3.4 Equivalence Classification

The research distinguishes four verification levels.

Level	Definition	Verification mechanism
Structural equivalence	Compliance with specified structural constraints	AST analysis and dependency analysis
Observation equivalence	Matching observable results over tested inputs	Differential testing

Level	Definition	Verification mechanism
Empirical equivalence	Compatibility across generated and fuzzed test populations	Property-based testing and fuzzing
Semantic equivalence	Equivalence under a formal semantic definition	Formal proof or bounded symbolic reasoning

These levels are not interchangeable. Structural equivalence does not imply behavioral equivalence, and empirical equivalence does not automatically establish universal semantic equivalence.

4. Proposed Architecture

4.1 Architectural Overview

The proposed architecture consists of six operational stages.

Stage 1: Program Extraction

The target function is extracted from its source representation. The extraction process preserves the original source and establishes a reference version.

Stage 2: Structural Analysis

The extracted function is analyzed to identify function identity, parameters and signature, local variables, referenced symbols, relevant dependencies, and scope boundaries.

Stage 3: Candidate Transformation

A controlled transformation is applied to the extracted representation. The output is stored independently as candidate version P_1 .

Stage 4: Isolated Verification

The candidate is executed in an isolated process. Verification includes syntactic validation, structural validation, and differential behavioral testing.

Stage 5: Stability Assessment

The candidate is evaluated against the specified invariant set and acceptance policy. Any failed condition results in rejection.

Stage 6: Controlled Activation

An accepted candidate may be activated through a controlled deployment mechanism. The deployment mechanism retains the original version to support recovery.

4.2 Two-Phase Rewriting Model

The architecture separates rewriting into two principal phases.

Phase A — Candidate Construction

$$P_1 = T(P_0, \theta)$$

The candidate is generated without modifying the active program.

Phase B — Candidate Acceptance and Activation

$$P_{active} = P_1 \text{ if all acceptance conditions hold, otherwise } P_0$$

This separation prevents candidate generation from being treated as equivalent to successful deployment.

4.3 Transformation Lifecycle

The complete operational sequence is:

- 1 Extract the original function.
- 2 Preserve the reference source.
- 3 Analyze structural dependencies.
- 4 Generate a candidate transformation.

- 5 Validate syntactic integrity.
- 6 Validate structural invariants.
- 7 Execute the candidate in isolation.
- 8 Perform differential testing.
- 9 Evaluate behavioral acceptance conditions.
- 10 Activate the candidate using a controlled transition.
- 11 Monitor the resulting execution.
- 12 Restore the original version if a defined regression condition occurs.

5. Transformation Classes

The initial implementation supports four transformation classes.

5.1 Constant Folding

Constant folding evaluates expressions whose operands are known constants.

```
def calculate(): return 2 + 3
```

Transformed version:

```
def calculate(): return 5
```

The transformation is admissible when evaluation preserves the relevant language semantics, including applicable type behavior and exceptions.

5.2 Algebraic Simplification

Algebraic simplification replaces an expression with an equivalent expression under explicitly defined conditions.

```
def multiply(value): return value * 1
```

Candidate:

```
def multiply(value): return value
```

Such transformations require care because programming-language semantics may include overloaded operators, custom objects, or side effects.

Accordingly, the initial transformation engine must restrict the supported expression domain rather than assume universal algebraic equivalence.

5.3 Dead Branch Elimination

Dead branch elimination removes unreachable branches under a proven or explicitly restricted condition.

```
def evaluate(value): if False: return value * 10 return value
```

Candidate:

```
def evaluate(value): return value
```

The implementation must distinguish literal constant conditions from expressions whose values depend on runtime state.

5.4 Local Variable Renaming

Local variable renaming changes an identifier while preserving its binding relationships.

```
def calculate(value): temporary = value + 1 return temporary
```

Candidate:

```
def calculate(value): result = value + 1 return result
```

The transformation must be scope-aware and preserve all relevant references. It must not modify function parameters, global references, attributes, or identifiers belonging to nested scopes unintentionally.

5.5 Transformation Restrictions

The initial implementation excludes:

- Function signature modifications.
- Import insertion or removal.
- Closure-dependent transformations.
- Decorated functions.
- Cross-function modifications.
- Arbitrary changes to external state.
- Transformations requiring module-wide dependency updates.

These restrictions define the experimental boundary rather than a universal limitation of the architecture.

6. Invariant Specification

6.1 Syntactic Invariants

The candidate must:

- 1 Parse successfully.
- 2 Contain a valid top-level function definition.
- 3 Preserve the expected function identity.
- 4 Produce valid source code.
- 5 Contain no malformed transformation artifacts.

6.2 Structural Invariants

The candidate must:

- 1 Preserve the original function signature.
- 2 Preserve required symbol references.
- 3 Maintain valid local variable bindings.
- 4 Avoid introducing unresolved dependencies.
- 5 Respect the permitted transformation scope.

Dependency analysis must account for legitimate recursion and language-specific binding rules.

6.3 Behavioral Invariants

The candidate must preserve the specified observable behavior over the experimental input population. The initial comparison model includes:

- Return-value comparison.
- Exception comparison.
- Relevant side-effect comparison.
- Execution-trace comparison where applicable.

The observation model must be identical for both versions.

6.4 Invariant Ownership

Invariants are specified before transformation by the research protocol or the software owner.

Some structural properties can be inferred automatically, but inferred properties must not silently replace explicitly required behavioral constraints.

The architecture therefore distinguishes:

- Declared invariants.
- Automatically derived invariants.
- Experimentally observed properties.

7. Isolation, Deployment, and Recoverability

7.1 Execution Isolation

The initial pilot uses Python multiprocessing.

The parent process maintains the original reference program while the candidate is evaluated in a separate process.

The objective is to prevent ordinary candidate execution from modifying the parent process's in-memory program representation.

This arrangement does not provide a hardened security boundary.

7.2 Deployment Integrity

Deployment integrity concerns the transition between the original and candidate versions.

The implementation must prevent partial replacement and preserve a known-good reference.

Where a file-based deployment mechanism is used, atomic filesystem replacement can be employed under appropriate filesystem conditions.

For in-memory function activation, the implementation must define how references to the active callable are updated and how concurrent invocations are handled.

An atomic pointer or dispatch-table update is preferable to modifying a live function object in place.

7.3 Rollback

Rollback restores the original executable version when deployment or post-deployment monitoring detects an unacceptable condition.

The rollback operation must be independently observable and measurable.

7.4 Recoverability

Recoverability extends beyond successful version replacement.

A system may successfully restore the original code while failing to restore its associated execution state.

For this reason, the research introduces **State Restoration Fidelity (SRF)**.

For a reference state S_0 and recovered state S_r , define:

$$SRF = (\text{Number of restored state properties}) / (\text{Number of required state properties})$$

This ratio is meaningful only when the required state properties are explicitly enumerated and equally weighted, or when an appropriate weighting scheme is specified.

For exact recovery under a controlled test, $SRF = 1$ must hold for all required state properties.

In the initial stateless pilot, this metric is not treated as demonstrated recovery of a stateful application. Stateful recovery is introduced as a separate experimental extension.

7.5 Four-Dimensional Operational Model

The architecture distinguishes:

Dimension	Primary question	Evaluation
Isolation	Can candidate execution be separated from the original process?	Isolation tests
Verification	Does the candidate satisfy defined acceptance conditions?	Invariant and behavioral tests
Transition integrity	Can activation occur without partial or inconsistent deployment?	Deployment and continuity tests
Recoverability	Can the previous required state be restored?	Rollback and SRF

No dimension is assumed to guarantee another.

8. Experimental Methodology

8.1 Experimental Design

The initial research uses a controlled pilot study. The pilot compares three execution strategies:

- **Method A:** Conventional replacement.
- **Method B:** Runtime AST transformation without isolated candidate validation.
- **Method C:** Runtime AST transformation with isolated verification and controlled activation.

All three methods must use equivalent transformation operations and comparable workloads.

8.2 Experimental Environment

The initial implementation environment is:

- Python 3.11 or later.
- LibCST for concrete syntax representation and transformation.
- Python multiprocessing for candidate isolation.
- A controlled test runner.
- A consistent hardware and operating-system environment.
- Version-controlled source code and experiment configurations.

Exact operating-system, CPU, RAM, dependency, and software-version details must be recorded in the final experimental report.

8.3 Workload Design

Three workload categories are proposed.

Workload A — Micro-benchmark: simple computational functions used to measure transformation and verification overhead.

Workload B — Representative Function Workload: functions extracted from small application components, used to assess realistic transformation behavior.

Workload C — Stateful Extension: a stateful service introduced in a later experimental stage to examine recovery fidelity. This workload is not part of the initial stateless acceptance experiment.

8.4 Transformation Population

The pilot initially targets 50 eligible functions, four transformation classes, and three execution strategies. The proposed experimental population is therefore:

$$50 \times 4 \times 3 = 600$$

function-transformation-method combinations, before accounting for repeated trials.

Not every function will necessarily support every transformation. Eligibility must be determined before experimentation and reported transparently.

8.5 Failure Injection

Controlled failure injection will examine the detection and recovery mechanisms. The initial failure categories include:

- 1 Invalid transformed syntax.
- 2 Unresolved symbol references.
- 3 Invalid scope modifications.
- 4 Function signature incompatibility.
- 5 Semantically incorrect transformations.
- 6 Candidate execution exceptions.
- 7 Deployment failure.
- 8 Post-deployment regression.

Failure injection must distinguish invalid transformations from valid transformations that are simply unsupported by the current implementation.

8.6 Baseline Control

All comparative methods must use the same hardware, the same software environment, the same workload population, the same transformation definitions, the same input datasets, comparable repetition counts, and consistent measurement procedures.

Measurements should distinguish transformation time, validation time, deployment time, and total operational overhead.

9. Evaluation Metrics

9.1 Transformation Success Rate

$$TSR = (\text{Successfully generated valid candidates}) / (\text{Eligible transformation attempts})$$

9.2 Unsafe Transformation Detection Rate

$$UDR = (\text{Unsafe transformations correctly rejected}) / (\text{Total unsafe transformations introduced})$$

9.3 False Positive Rate

$$FPR = (\text{Valid transformations incorrectly rejected}) / (\text{Total valid transformations})$$

For this research, a false positive occurs when a valid, supported transformation is incorrectly rejected.

9.4 False Negative Rate

$$FNR = (\text{Unsafe transformations incorrectly accepted}) / (\text{Total unsafe transformations})$$

A false negative occurs when an unsafe transformation is accepted. The ground-truth classification must be established independently of the system being evaluated.

9.5 Runtime Overhead

$$OH = T_{\text{proposed}} / T_{\text{baseline}}$$

The report must distinguish total process latency from the latency experienced by ordinary application requests.

9.6 Service Disruption

Service disruption measures the period during which the active service cannot satisfy its specified operational contract during deployment. It must be measured separately from candidate verification time.

9.7 Mean Time to Recovery

$$MTTR = (\sum t_i) / n$$

where t_i represents the measured time required to restore the accepted previous version following a controlled failure.

9.8 State Restoration Fidelity

SRF is measured independently through explicitly enumerated state properties. It is not inferred from rollback success.

9.9 Verification Coverage

The proportion of declared invariants and observable behavioral properties that receive an actual verification procedure must also be reported. Coverage does not establish correctness, but it identifies the scope of the evidence supporting acceptance.

10. Hypotheses and Experimental Acceptance Criteria

The research begins with four principal hypotheses.

H1 — Isolation Hypothesis

Candidate execution in a separate process reduces the propagation of ordinary transformation failures into the original process compared with execution without process isolation.

H2 — Verification Hypothesis

Combining structural validation and differential testing detects more injected behavioral regressions than structural validation alone.

H3 — Deployment Hypothesis

Controlled activation and rollback reduce deployment disruption compared with conventional replacement under equivalent experimental conditions.

H4 — Recovery Hypothesis

Successful rollback does not necessarily imply complete restoration of required execution state.

These hypotheses will be evaluated through controlled experiments. Numerical acceptance thresholds will be established after baseline measurements and pilot calibration, rather than assumed to be universal performance requirements.

11. Transition Criteria: Function-Level to Module-Level

The initial architecture is intentionally restricted to individual functions.

Extending the transformation scope introduces new engineering conditions, including shared state, dependency interactions, cross-function effects, and module initialization behavior.

Consequently, progression to module-level rewriting requires four explicit conditions.

Criterion	Required condition	Measurement
Mechanism stability	Transformation success rate $\geq 90\%$ on eligible valid transformations	Controlled pilot
Unsafe-change detection	Detection rate $\geq 95\%$ on the defined injected failure population	Failure injection
Computational cost	Measured overhead $\leq 2\times$ baseline	Repeated benchmark
Recovery fidelity	SRF = 100% for all required properties in controlled recovery tests	Recovery experiments

These thresholds are proposed engineering gates for this research, not established scientific laws.

If any condition fails, the architecture must be revised and re-evaluated before expansion.

A successful function-level experiment does not automatically establish module-level correctness.

12. Four-Week Pilot Implementation Plan

Week 1: Program Extraction and Transformation

Objectives:

- Establish the Python and LibCST environment.
- Implement function extraction.
- Implement the four supported transformation classes.
- Create a controlled population of test functions.

Deliverables:

- Extractor implementation.
- Transformation implementation.
- Initial transformation test suite.

Week 2: Invariant Verification

Objectives:

- Implement syntactic validation.
- Implement structural validation.
- Define the behavioral observation model.
- Implement differential testing.

Deliverables:

- Invariant verification modules.
- Differential testing module.
- Initial failure-injection suite.

Week 3: Isolation and Deployment

Objectives:

- Implement multiprocessing-based candidate execution.
- Implement controlled activation.
- Implement rollback.
- Establish deployment and recovery instrumentation.

Deliverables:

- Isolated execution module.
- Deployment module.
- Rollback tests.
- Preliminary recovery measurements where applicable.

Week 4: Pilot Evaluation

Objectives:

- Execute the controlled experimental matrix.

- Collect baseline measurements.
- Analyze transformation failures.
- Evaluate verification performance.
- Determine whether the transition criteria are satisfied.

Deliverables:

- Pilot dataset.
- Experimental report.
- Revised implementation specification.
- Function-level progression decision.

13. Threats to Validity

13.1 Construct Validity

The definition of observable behavior determines what differential testing can detect. An incomplete observation model may classify behaviorally different functions as equivalent. The research therefore requires explicit documentation of observed return values, exceptions, state changes, and relevant side effects.

13.2 Internal Validity

Differences between experimental methods may result from implementation details rather than the architectural approach itself. Controlled hardware, software versions, workloads, and measurement procedures are therefore required.

13.3 External Validity

Results obtained from individual Python functions cannot automatically be generalized to other languages, execution environments, or complete software systems. The initial conclusions will remain restricted to the evaluated environment.

13.4 Statistical Validity

A small pilot population may not provide sufficient statistical power to establish general performance claims. The pilot is primarily intended to establish feasibility, identify implementation limitations, and estimate the variance needed for a subsequent confirmatory study.

13.5 Security Validity

Multiprocessing provides process separation but is not a hardened security sandbox. The study does not claim protection against adversarial code or privileged operating-system attacks. Security-oriented evaluation would require a separate experimental configuration using stronger isolation technologies.

14. Discussion

The proposed architecture treats runtime self-modification as a controlled engineering operation rather than an unrestricted autonomous capability.

This distinction has several implications.

First, the ability to generate a modified program is separated from the ability to establish its admissibility.

Second, the verification process is layered. Syntactic validity, structural integrity, and behavioral compatibility represent different properties and require different evidence.

Third, candidate isolation is separated from deployment integrity. A candidate can execute safely in an isolated process while still being deployed incorrectly.

Fourth, rollback is separated from recoverability. Restoring an earlier executable version does not necessarily restore the complete operational state.

Finally, the experimental scope is progressive. Function-level validation provides evidence about the transformation mechanism, but expansion to modules requires additional analysis of dependencies and shared state.

The architecture therefore provides a foundation for investigating autonomous software modification under explicit operational constraints.

15. Limitations and Future Research

The present study is restricted to a controlled Python environment and a limited collection of transformation types. Several extensions remain open for subsequent investigation.

15.1 Module-Level Rewriting

Future work should examine cross-function dependencies, module initialization, shared namespaces, and import behavior.

15.2 Stateful Recovery

A dedicated experimental study should evaluate state snapshots, restoration fidelity, concurrent execution, and consistency following rollback.

15.3 Stronger Execution Isolation

Docker, gVisor, Firecracker, and WebAssembly-based environments may provide stronger isolation properties under different operational assumptions. Their effectiveness and overhead require direct comparative evaluation.

15.4 Formal Semantic Verification

Differential testing provides empirical evidence but cannot establish universal equivalence. Future research may investigate bounded symbolic execution, SMT-based verification, and formal equivalence techniques for restricted transformation classes.

15.5 Concurrency and Live Deployment

Function replacement in concurrent applications requires explicit treatment of active references, in-flight calls, synchronization, and version consistency.

15.6 Cross-Language Generalization

The architecture could be evaluated using JavaScript, Rust, or JVM-based systems, although each language requires its own transformation and runtime semantics.

16. Implementation Status (Phase I)

Subsequent to the initial manuscript (Version 1.0), the architecture specified in Sections 4–7 was implemented in full for the function-level, stateless scope. This section reports the implementation status as of Version 0.2.0-implementation; it does not report pilot or empirical results, which remain pending (Sections 8–12 are unchanged as a prospective protocol).

Layer	Module	Spec	Status	Tests
Core — Extraction	core/extractor.py	—	Complete	3
Core — Transformation	core/transformer.py	4 transforms	Complete	4
Syntactic (I _s)	invariants/syntactic.py	S1–S6	Complete	22
Structural (I _t)	invariants/structural.py	T1–T5	Complete	23
Behavioral (I _b)	invariants/behavioral.py	B1–B4	Complete	18
Sandbox — Isolation	sandbox/isolated_runner.py	multiprocessing + side-effect enforcement	Complete	17
Sandbox — Differential	sandbox/differential.py	P0 vs P1 on corpus	Complete	12
Deployer	core/deployer.py	atomic swap + rollback	Complete	19
Total			118/118 passing	118

Current VERSION: 0.2.0-implementation · **Scope:** Function-level, stateless, Python 3.11+ · **Test environment:** Python 3.13.13, libcst 1.9.0, pytest 9.0.3.

Phase I status: Complete — all eight modules implemented and tested. No pilot study has been executed; no empirical results (TSR, UDR, FPR, FNR, OH, MTTR, SRF) are reported at this stage.

16.1 Syntactic Invariants Implemented (I_s)

#	Condition	Check
S1	P ₁ parses as valid Module	cst.parse_module
S2	Contains one FunctionDef node	count
S3	Function name unchanged	name compare
S4	Signature unchanged	params compare
S5	No new decorators	decorator set
S6	No new imports	Import scan

16.2 Structural Invariants Implemented (I_t)

#	Condition	Check
T1	Every Name has a binding	scope analysis
T2	No invalid dependency cycles	DFS cycle detection
T3	No orphan variables	liveness diff
T4	No new global statements	AST scan
T5	All local calls resolve	symbol table

16.3 Behavioral Invariants Implemented (I_b)

#	Condition	Check
B1	Return values match on corpus	differential
B2	Exceptions match (type + message)	differential
B3	stdout trace matches	differential
B4	No new side effects	sandbox enforcement

Timeout → Inconclusive (never treated as Accept).

17. Conclusion

This paper presented a controlled architecture for runtime code rewriting based on structured program transformation, layered invariant verification, isolated candidate execution, differential behavioral testing, and controlled deployment.

The central research proposition is that autonomous software modification should be evaluated through independently specified conditions of structural validity, behavioral compatibility, deployment integrity, and recoverability.

The initial implementation is restricted to individual top-level Python functions, using LibCST and multiprocessing-based logical isolation.

Four transformation classes provide a controlled starting population for experimental evaluation.

The study also introduces State Restoration Fidelity as a separate measure, recognizing that successful rollback does not necessarily establish complete recovery.

Rather than claiming that the proposed architecture already guarantees safe self-modification, the paper establishes a falsifiable research program through which the relevant engineering properties can be measured.

Progression to module-level and system-level rewriting is conditional on experimentally demonstrated performance and recovery criteria.

The resulting approach provides a structured foundation for studying runtime software transformation without conflating the ability to modify code with the ability to modify it reliably.

As of Version 0.2.0-implementation, Phase I — the full function-level architecture — is implemented and passes its complete test suite (118/118). The next research stage is pilot experimentation and empirical evaluation against the criteria of Sections 9–11.

References

1. Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Communications of the ACM*, 12(10), 576–580.
2. Knuth, D. E. (1969). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley.
3. Fraser, C. W., & Hanson, D. R. (1995). *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings.
4. Leavens, G. T., Baker, A. L., & Ruby, C. (2006). JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*.
5. McKeeman, W. M. (1998). Differential testing for software. *Digital Technical Journal*, 10(1), 100–107.
6. Tichy, W. F. (1998). Should computer scientists experiment more? *Computer*, 31(5), 32–40.
7. Harman, M., & Jones, B. F. (2001). Search-based software engineering. *Information and Software Technology*, 43(14), 833–839.
8. LibCST Project. *LibCST Documentation*. <https://libcst.readthedocs.io/>
9. Python Software Foundation. *Python 3 Documentation: multiprocessing*. <https://docs.python.org/3/library/multiprocessing.html>
10. Python Software Foundation. *Python 3 Documentation: ast*. <https://docs.python.org/3/library/ast.html>
11. Hicks, M., & Nettles, S. (2005). Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6), 1049–1096.
12. Field, A. (2013). *Discovering Statistics Using IBM SPSS Statistics*. SAGE Publications.

Current Manuscript Status

Dimension	Status
Formal model	Defined (§3)
Architecture	Specified (§4)
Invariant layers	Specified (§6)
Implementation (Phase I)	Complete — 118/118 tests passing (§16)
Pilot data	Not yet collected
Empirical results	Not yet available
Module-level rewriting	Out of scope (Phase II)
Stateful recovery	Deferred (Phase III)
Adversarial isolation	Out of scope

Record ID: AUTO-PROTO-01 · **Version:** 0.2.0-implementation · **DOI:** <https://doi.org/10.5281/zenodo.23475104> · **Scope:** Function-Level Runtime Code Rewriting · **Implementation Baseline:** Python 3.11+ / LibCST / multiprocessing.

Next Research Stage: Pilot experimentation and empirical evaluation against the criteria of Sections 9–11.

Samir Baladi · Ronin Institute / Rite of Renaissance · gitdeeper11@gmail.com · ORCID: 0009-0003-8903-0029 · DOI: 10.5281/zenodo.23475104