TraceScript Code Medium

Governing AI Code Changes as Substrate Mutations

Canonical Public White Paper v1.0

Subtitle:
Code object traces, geometric software substrate, topological reachability, interface contracts, package integrity, test proximity, deployment gates, external write-path governance, receipts, replay, and release proof for AI coding agents

Primary contribution: AI Coding Agent Security
Secondary contribution: Geometric Software Substrate Governance
Tertiary contribution: A reference runtime architecture for proving whether AI-authored code changes are topologically safe to merge, deploy, and execute

---

Abstract

AI coding agents are changing the software-development boundary.

For most of software history, code review focused on whether a change was syntactically correct, functionally plausible, stylistically acceptable, adequately tested, and aligned with developer intent. CI/CD systems checked builds, unit tests, linting, type safety, dependency vulnerabilities, and sometimes security rules. Human reviewers inspected diffs. Static analyzers looked for patterns. Deployment pipelines checked whether the system could ship.

That model is still necessary. It is no longer sufficient.

AI coding agents can generate, modify, refactor, connect, extract, migrate, and deploy software faster than human teams can reason through the full topological consequences of those changes. A change may compile. Tests may pass. The diff may look small. The agent's explanation may sound coherent. Yet the change may create a new external write path, bypass an adapter, weaken an interface contract, couple two previously separated modules, mutate schema assumptions, alter data-flow topology, introduce package drift, route sensitive data through an ungoverned boundary, or make deployment technically possible while leaving action basis, receipt coverage, rollback evidence, or reconciliation incomplete.

A code change can compile and still be topologically unsafe.

TraceScript Code Medium is a runtime architecture for governing AI code changes as substrate mutations. It treats the codebase not as a passive repository of files, but as a geometric software substrate whose components, hooks, edge functions, schemas, policies, packages, tests, migrations, API routes, adapters, interface contracts, deployment paths, external writes, and receipts form a living topology that shapes future execution.

The central problem is not only whether code is valid. The deeper problem is whether the proposed code mutation preserves the lawful response geometry of the software substrate.

Code Medium asks:

Is this change reachable from protected execution paths?
Does it cross a governed boundary?
Does it introduce an unmediated external write?
Is the required policy on the execution path?
Are interface contracts preserved?
Are tests topologically close to the changed object?
Does the package extraction preserve coherence?
Does the migration match runtime schema use?
Does the deployment have action basis, receipt, replay, rollback, and reconciliation proof?
Can the system reconstruct why this change was allowed to touch production?

TraceScript Code Medium provides a governed runtime between “agent wrote code” and “code touched production.”

Its core doctrine is:

A code deployment is a state-bearing external action; therefore AI-authored code must be governed as substrate before it can merge, deploy, or mutate external systems.

Its winning sentence is:

A code change can compile and still be topologically unsafe.

Keywords

TraceScript
Code Medium
AI Coding Agent Security
geometric software substrate
codebase substrate
topological safety

topological reachability
code object traces
AI-generated code governance
deployment governance
interface contracts
test proximity
package integrity
external write paths
adapter governance
schema migration integrity
code topology
runtime proof
deployment receipts
replayable release proof
code-medium governance
AI software supply chain
agentic coding security
governed configuration assembly
substrate-oriented programming
state-bearing computational systems

---

1. Introduction

AI coding agents are becoming operational software participants.

They do not merely autocomplete. They read repositories, infer architecture, modify files, add routes, write components, generate migrations, change schemas, insert packages, create tests, call tools, open pull requests, produce refactors, and increasingly connect to CI/CD, deployment, cloud, database, and repository automation.

This changes the security and governance problem.

A human developer changes code from inside a tacit map of the system. Even then, failures happen. But AI agents operate from partial context, generated summaries, embeddings, tool outputs, stale documentation, incomplete tests, and local diff views. They can produce code that compiles but changes the system's topology in a way the agent does not understand.

The visible diff is not the whole change.

A line in a component may create a new write path to a database.
A hook may introduce a hidden side effect.
An API route may bypass a policy layer.
A migration may alter assumptions for multiple services.
A package may introduce transitive supply-chain risk.
A refactor may move logic outside an audited boundary.
A test may pass while failing to cover the changed execution path.
A deployment may succeed while lacking proof that it was authorized, reviewed, replayable, and reversible.

Traditional code review asks:

Does this code work?

TraceScript Code Medium asks:

What substrate did this code change, and is the resulting software geometry safe to execute?

This is a deeper question.

A codebase is not merely a collection of files. It is a state-bearing medium. Its objects, routes, dependencies, schemas, policies, packages, tests, adapters, workflows, and deployments define the response geometry through which future computation occurs. When an AI agent changes code, it is not simply editing text. It is intervening in a living software substrate.

TraceScript Code Medium exists to govern that intervention.

---

2. Relationship to TraceScript

TraceScript is a substrate-oriented programming language and runtime architecture for governing how signals become trusted state, action basis, and future computation in state-bearing computational systems.

Code Medium is a product and runtime instantiation of TraceScript.

TraceScript defines the trunk:

governed signal-to-substrate mutation.

Code Medium defines a high-value technical branch:

governed code-to-production substrate mutation.

In TraceScript terms:

a code change is a signal
a code object is a trace
a repository is a medium
a module boundary is a boundary
a dependency graph is topology
a policy is execution semantics
a deployment is a protected action
a release artifact is proof
a receipt is substrate evidence
a rollback may leave residue
a production effect requires replay and reconciliation

Code Medium applies the TraceScript runtime to software itself.

Where Substrate Integrity protects what agents believe, and Agent Action Firewall protects what agents do, Code Medium protects the software substrate agents modify.

The three are complementary:

Substrate Integrity governs trusted memory, retrieval, policy, and knowledge.
Agent Action Firewall governs protected external action release.
Code Medium governs AI-authored software mutations before merge, deployment, and production effect.

In the TraceScript stack, Code Medium is not the core language. It is a major runtime/product instantiation that proves TraceScript can govern codebases as living substrates.

---

3. The Codebase as a Living Substrate

A codebase is not passive text.

A codebase encodes executable relationships. It carries architecture, dependencies, boundaries, policies, runtime assumptions, data contracts, tests, schema expectations, package surfaces, deployment pathways, and external effects.

A component is not merely a file. It may render sensitive data, call hooks, trigger state changes, and expose user actions.

A hook is not merely a reusable function. It may bind UI behavior to memory, network calls, permissions, or external writes.

An API route is not merely a handler. It is a governed boundary between clients, services, databases, policies, and external systems.

A schema is not merely a type or table. It is a contract between persistence, application logic, migrations, validations, permissions, and analytics.

A package is not merely dependency code. It is an imported trust boundary with its own topology, update cadence, vulnerabilities, maintainers, and transitive effects.

A test is not merely a pass/fail artifact. It is evidence of nearby expected behavior.

A deployment is not merely a release. It is a state-bearing external action that changes the operational response geometry of the system.

In Code Medium, the codebase is treated as a geometric software substrate.

It has:

objects
coordinates
edges
paths
boundaries
regions
interfaces
contracts
policies
locks
traces
receipts
deployment risk
coherence state

architectural drift
runtime reliability
external write surfaces

The goal is not to replace compilers, tests, or CI/CD. The goal is to add the missing layer:

topological governance of code substrate mutation.

---

4. The Core Thesis

The core thesis is:

A code change can compile and still be topologically unsafe.

This sentence separates Code Medium from ordinary static analysis, code review, test automation, and vulnerability scanning.

A syntactically valid change can still be unsafe because it may alter topology.

Topological unsafety includes:

creating an unmediated external write path
bypassing a policy layer
crossing a sensitive boundary without adapter governance
introducing a new dependency into a protected path
moving logic outside receipt coverage
adding a route without authorization checks
changing a schema without migration alignment
altering a hook used by high-risk components
removing a guard from a reachable path
creating an execution path with no nearby tests
creating package drift
breaking an interface contract
making deployment possible without release proof
expanding runtime capability without authority
creating hidden coupling between previously separated media

A compiler does not usually know whether a write path is governed.
A unit test may not know whether a policy is on-path.
A linter may not know whether a package extraction preserves architectural coherence.
A CI pipeline may not know whether an AI agent's action basis was valid.
A human reviewer may not inspect all reachable consequences.

Code Medium adds a runtime and analysis layer for those questions.

---

5. Why AI Coding Agents Make This Urgent

AI coding agents change the scale, speed, and shape of code mutation.

They can produce changes across multiple files in seconds.
They can refactor without fully understanding architectural intent.
They can introduce packages to solve local problems.
They can follow stale documentation.
They can copy patterns from unrelated contexts.
They can generate tests that validate their own assumptions.
They can create migrations from inferred schemas.
They can modify routes, adapters, and permissions in the same change.
They can make code look coherent while quietly altering topology.

Human teams already struggle with software architecture drift. AI coding agents increase mutation rate dramatically.

The core danger is not only bad code. It is ungoverned code substrate evolution.

A repository can appear healthy while its topology becomes fragile:

more hidden coupling
more unreviewed dependencies
more direct external writes
more edge routes without governance
more schema drift
more policy bypasses
more tests far from the changed paths

more deployment paths without replay proof
more packages with unclear ownership
more generated code with unclear lineage

Code Medium is designed to make this substrate evolution visible, governable, repairable, and replayable.

---

6. Why Existing Tools Are Not Enough

Code Medium is not introduced because existing tools are useless. It is introduced because existing tools govern different layers.

6.1 Compilers are necessary but insufficient

A compiler checks whether code satisfies language rules. It does not determine whether the changed object creates an unsafe path through the system.

The code can compile and still bypass governance.

6.2 Type systems are necessary but insufficient

Types help enforce local contracts. They do not generally evaluate whether a high-risk deployment path has policy, authority, test proximity, receipt coverage, and rollback proof.

The type is valid. The configuration may be unsafe.

6.3 Linters are necessary but insufficient

Linters enforce style and common patterns. They do not understand organizational boundaries, external write governance, protected action basis, or runtime release proof.

6.4 Static analyzers are necessary but insufficient

Static analyzers can find important vulnerabilities and code patterns. But Code Medium asks broader substrate questions: topology, governance path, reachability, receipt coverage, package coherence, deployment admissibility, and replay.

6.5 Unit tests are necessary but insufficient

A passing test does not prove that the changed object is tested along the relevant execution path.

The test may be far from the mutation. The test may validate the wrong behavior. The test may not cover policy enforcement, external writes, migration effects, or deployment residue.

Code Medium evaluates test proximity.

6.6 CI/CD is necessary but insufficient

CI/CD determines whether code can build, test, package, and deploy. It does not necessarily determine whether the deployment should be released from the current action basis.

CI/CD asks: can this ship?

Code Medium asks: is this topology safe to ship?

6.7 Code review is necessary but insufficient

Human reviewers are critical. But AI coding agents can produce broad, multi-object changes with nonlocal consequences. Human review should be augmented by topology-aware obstruction, repair, and release proof.

6.8 SBOM and dependency tools are necessary but insufficient

Dependency tools track packages and vulnerabilities. Code Medium evaluates whether package introduction or extraction changes substrate topology, coherence, ownership, and protected execution paths.

---

7. Product Category

TraceScript Code Medium creates or occupies the category:

AI Coding Agent Security

More specifically:

Geometric Software Substrate Governance

Related category phrases include:

Code Medium Governance
AI Code Release Firewall
Topological Code Security
Agentic Software Change Governance
AI Code Deployment Gate
Software Substrate Integrity
Code Topology Runtime
Deployment Proof Runtime
AI Coding Agent Control Plane
Geometric Code Review

The best product name is:

TraceScript Code Medium

The best supporting line is:

Governing AI code changes as substrate mutations.

The best CISO-facing line is:

TraceScript prevents AI coding agents from merging or deploying code that creates ungoverned external write paths, bypasses policies, violates interface contracts, lacks test proximity, or cannot be replayed.

The best developer-facing line is:

TraceScript maps code objects as substrate traces, computes topology, checks reachability, verifies contracts, selects topologically close tests, gates deployment, emits receipts, and proves release.

The best strategic line is:

A runtime between “agent wrote code” and “code touched production.”

---

8. Code Objects as Traces

Code Medium treats code objects as traces.

A CodeTrace may represent:

component
hook
operator
function
class
edge function
API route
schema
migration
policy
package
test
adapter
database procedure
configuration file
deployment script
CI job
permission rule
external write path

feature flag
interface contract
lock artifact

A CodeTrace is not merely a file reference.

It carries:

source location
semantic role
coordinate
dependencies
dependents
runtime reachability
boundary crossings
effect class
policy requirements
test coverage
test proximity
interface contracts
package lineage
deployment risk
receipt coverage
owner
authority
state
phase
coherence contribution
runtime reliability

This transforms the repository from a file tree into a trace topology.

A file tree answers:

Where is the code?

A Code Medium answers:

What does this code object affect, what paths reach it, what boundaries does it cross, what policies govern it, what tests prove it, what contracts constrain it, and what receipts make it safe to release?

---

9. Geometric Software Substrate

Code Medium models the codebase geometrically.

This does not require visual metaphor. It means the runtime treats code objects as located in a structured space of relationships.

Code coordinates may include:

file path
module path
package namespace
runtime region
service boundary
schema region
data domain
permission domain
execution layer
dependency layer
test layer
policy layer
deployment layer
external adapter layer

Edges may include:

imports
function calls
hook usage
route invocation
database access
schema dependency
test coverage
package dependency
policy dependency
adapter dependency
deployment dependency
event emission

queue consumption
external API call
permission dependency

A topology emerges.

The topology tells the runtime what is reachable, what is central, what is fragile, what is protected, what crosses boundaries, what lacks tests, what bypasses adapters, and what may affect production.

This is why Code Medium is geometric software governance.

It governs not only the content of code, but the shape of software consequence.

---

10. Topological Unsafety

Topological unsafety occurs when a code change creates a dangerous configuration even though the changed code may be locally valid.

Examples:

A React component writes directly to a database instead of calling a governed API adapter.

A hook introduced for convenience is used by multiple protected components and now writes external state.

An edge function bypasses an authorization middleware.

A schema migration changes a field used by billing logic, but tests cover only UI rendering.

A package extraction moves validation logic into a shared package but leaves one route calling the old local validation.

A new dependency is introduced into an authentication path without supply-chain review.

A generated test validates the new behavior but is not on the protected execution path.

A deployment script changes environment variable behavior without receipt or rollback proof.

A route calls an external API directly rather than through a reconciliation adapter.

A policy check exists in one code path but not the path newly created by the agent.

A code change may therefore be valid as code but invalid as substrate.

Code Medium detects this distinction.

---

11. Topological Reachability

Reachability is central.

A changed object is more dangerous if it lies on a path to:

external writes
customer-visible behavior
financial operations
legal commitments
authentication or authorization
privileged data
database mutation
policy enforcement
deployment
workflow approval
agent memory write
sensitive retrieval
permission change

Topological reachability asks:

Can execution reach this object from a protected entrypoint?
Can this object reach an external system?
Can this route reach sensitive data?
Can this component trigger a mutation?
Can this hook alter memory?
Can this schema change affect protected workflows?
Can this package influence high-risk runtime paths?
Can this migration alter action-basis data?

Reachability is not only graph connectivity. It is governed connectivity.

A path may exist but be admissible only if it crosses the right adapters, policies, tests, receipts, and release gates.

Code Medium therefore distinguishes:

reachable
protected reachable
externally reachable
privileged reachable
policy-mediated reachable
adapter-mediated reachable
unmediated reachable
unreplayable reachable
unsafe reachable

The dangerous state is not reachability alone. The dangerous state is ungoverned reachability.

---

12. External Write Path Governance

External writes are among the highest-risk code effects.

An external write path may touch:

database records
SaaS systems
payment systems
email systems
customer messaging
identity systems
cloud resources
workflow tools
code repositories
deployment targets
third-party APIs
file storage

analytics events
agent memory
policy corpora

Code Medium asks:

Does this code object create or modify an external write path?
Is the write path mediated by a governed adapter?
Is there a policy on the path?
Is authority checked before the write?
Is the action basis valid?
Is a release artifact required?
Is a receipt emitted?
Can the external result be reconciled?
Can the action be replayed?
Can rollback restore visible state, and what residue remains?

The canonical Code Medium example is:

A UI component should not directly write to an external system.

If a component touches an external write without an adapter, Code Medium should block deployment and propose repair:

insert adapter
move write behind route
add policy middleware
add receipt emission
add reconciliation handler
add test on reachable path
route for review

This is not merely style guidance. It is external-action safety.

---

13. Interface Contracts

An interface contract defines the expected relationship between code objects.

Contracts may include:

API request/response shape
schema invariants
adapter obligations
permission requirements
event payload shape
package public surface
component props
hook behavior
migration assumptions
policy preconditions
receipt obligations
reconciliation obligations
deployment assumptions

AI coding agents frequently violate implicit contracts because they infer from local context.

A contract violation may compile if the type system is incomplete, if any is used, if runtime data differs, if schema migration changes data shape, or if the contract is organizational rather than syntactic.

Code Medium makes interface contracts explicit substrate objects.

It asks:

What contract governs this boundary?
Is the changed object inside the contract scope?
Does the change strengthen, preserve, weaken, or break the contract?
Does the contract require tests?
Does the contract require approval?
Does the contract require receipt or replay?
Does the contract protect an external write path?

A contract is not merely documentation. It is a governance object.

---

14. Test Proximity

Traditional test coverage asks whether code is covered.

Code Medium asks whether tests are topologically close to the changed object and the affected execution path.

A test can exist and still be too far away.

Examples:

A component test covers rendering, but the change affects external mutation.
A route test covers happy-path response, but not policy enforcement.
A migration test covers schema creation, but not downstream query assumptions.
A package test covers the extracted function, but not the consuming paths.
A generated test validates the agent's intended behavior but not the protected path.

Test proximity evaluates:

distance from changed object to test
distance from test to protected path
whether test crosses same boundary
whether test observes same external effect
whether test validates same interface contract
whether test exercises same policy condition
whether test covers failure mode
whether test is generated, hand-written, reviewed, or historical
whether test has replayable evidence

Code Medium does not merely ask:

Are there tests?

It asks:

Are the right tests close enough to the risk?

15. Package Integrity and Extraction Coherence

AI coding agents often create packages, extract shared utilities, add dependencies, and refactor modules.

Package-level changes are topology-changing interventions.

A package extraction may be unsafe if:

it hides domain-specific assumptions
it weakens validation
it spreads a low-quality abstraction
it creates circular dependencies
it exports too much surface area
it moves code outside a policy boundary
it breaks ownership
it creates version drift
it lacks tests for consuming paths
it changes deployment or build behavior
it makes protected logic reusable in unsafe contexts

A package introduction may be unsafe if:

the dependency is untrusted
the transitive dependency graph is risky
the package touches sensitive paths
the package has weak maintenance history
the package overlaps existing internal logic
the package increases build or runtime fragility
the package lacks supply-chain review
the package crosses license constraints
the package is added by agent without authority

Code Medium evaluates package coherence.

It asks:

Is this package extraction coherent?
Does the package preserve domain boundaries?
Does it maintain interface contracts?
Does it remain test-close to consumers?
Does it introduce supply-chain risk?
Does it cross protected boundaries?
Is it owner-approved?
Can package promotion be replayed?

---

16. Schema and Migration Integrity

Schemas are substrate contracts.

A schema change may compile and migrate but still be unsafe.

Examples:

A nullable field becomes required without migration coverage.
A table relationship changes but workflow logic assumes old shape.
A policy-relevant field is renamed without updating enforcement.
A migration succeeds locally but breaks historical records.
A generated migration omits backfill.
A database change affects action-basis objects.
A schema update changes customer-visible behavior through downstream rendering.

Code Medium governs migrations as substrate mutations.

It asks:

What runtime paths depend on this schema?
What queries will change?
What components consume the data?
What policies reference the field?
What external systems synchronize it?
What tests are close to affected consumers?
Is backfill required?

Is rollback possible?
What residue remains if rollback occurs?
Is migration replayable?

A migration is not only a database operation. It is a change to the software substrate's memory structure.

---

17. Policy-on-Path

Code often contains policy checks, authorization checks, validation checks, privacy checks, data-handling rules, and workflow gates.

A policy may exist somewhere in the codebase and still not be on the execution path affected by a change.

Code Medium asks:

Is the relevant policy on the path?

For example:

A route writes customer data.
The repository contains a privacy policy middleware.
But the new route does not call it.

The policy exists. It is not on-path.

Another example:

A deployment policy requires review for external integrations.
The code change adds an external API call through a package.
The deployment gate does not classify it as external integration.

The policy exists. It is not attached to the changed path.

Code Medium separates policy existence from policy reachability.

A codebase is not safe because policies are present. It is safe when the relevant policies govern the relevant paths.

---

18. Deployment as Protected Action

A deployment is a state-bearing external action.

It changes the operational environment from which users, systems, agents, and external integrations experience computation.

Deployment may create:

customer-visible change
external writes
financial effect
security posture change
schema change
workflow change
policy behavior change
agent behavior change
data migration
external system synchronization
runtime residue

Therefore, Code Medium treats deployment as a protected action requiring release proof.

Deployment should require:

code topology evaluation
changed-object trace map
reachability assessment
interface contract verification
test proximity score
package integrity result
schema/migration result
policy-on-path result
deployment risk classification
rollback plan
receipt plan
replay class
approval route where required
release artifact

deployment receipt
post-deploy reconciliation
residue assessment where required

The deployment question is not:

Did the build pass?

It is:

Is this release topologically safe to enter production?

---

19. Code Receipts and Replayable Release Proof

Code Medium emits receipts for code substrate decisions.

Receipts may prove:

what code changed
who or what changed it
what agent generated it
what files and objects were affected
what topology changed
what paths became reachable
what external write paths were introduced
what contracts were checked
what tests were selected
what tests passed
what packages changed
what schemas changed
what policies were on-path
what deployment gate decided
what release artifact was issued
what environment received the deployment

what post-deploy reconciliation occurred
what rollback plan existed
whether replay can reconstruct the decision

A deployment receipt is not a CI log.

A CI log says what ran.

A deployment receipt proves why the runtime allowed production mutation.

Replay matters because code changes become future substrate. If a later incident occurs, the organization must reconstruct not only the diff, but the governed decision path that allowed the diff into production.

The doctrine is:

No receipt, no trusted deployment.
No replay, no durable release proof.
No topology, no high-assurance AI code release.

---

20. Obstruction and Repair

Code Medium should not merely block. It should explain and repair.

Obstruction examples:

unguarded external write path
missing adapter
missing policy on path
missing interface contract
contract violation
test too distant
schema migration mismatch
package supply-chain risk
package extraction incoherent
deployment risk too high
rollback plan missing
receipt coverage missing

replay insufficient
authorization missing
boundary crossing ungoverned
high-centrality object changed without review

Repair examples:

insert adapter
move write behind governed route
add policy middleware
add interface contract
add topologically close test
add migration backfill
pin or replace package
split package
restore boundary
route to owner
require human review
add receipt emission
add reconciliation handler
stage deployment
simulate deployment
reduce scope
block deploy

A good Code Medium runtime turns architectural failure into precise repair.

This is where the product becomes developer-useful.

---

21. Threat Model

Code Medium protects against unsafe AI-authored code substrate mutation.

21.1 Ungoverned external write path

AI-generated code writes directly to an external system without adapter, policy, receipt, or reconciliation.

Defense:

topological reachability
effect classification
adapter requirement
deployment gate
repair insert adapter

21.2 Policy bypass

The code change creates a path that avoids authorization, review, privacy, or compliance policy.

Defense:

policy-on-path analysis
boundary graph
contract verification
path obstruction

21.3 Interface contract violation

The change breaks expected shape, effect, permission, or behavior at a boundary.

Defense:

interface contract objects
contract check
test proximity
receipt

21.4 Package drift

The agent introduces or modifies package dependencies in a way that changes trust or runtime behavior.

Defense:

package lineage
dependency topology
supply-chain risk
owner review
package lock receipts

21.5 Schema drift

A schema or migration change breaks downstream assumptions.

Defense:

schema reachability
migration integrity
consumer path analysis
backfill and rollback check

21.6 Test laundering

The agent generates tests that prove the intended local behavior but do not cover the protected path.

Defense:

test proximity
test lineage
path coverage
failure-mode coverage

21.7 Deployment without action basis

The deployment is technically possible but lacks authority, policy, review, evidence, or receipt support.

Defense:

deployment governance gate
action-basis integrity
release artifact
receipt and replay

21.8 Boundary collapse

A change weakens separation between UI, server, database, policy, adapter, or external system.

Defense:

boundary graph
ungoverned crossing detection
adapter insertion repair

21.9 High-centrality mutation

The agent changes a central object that affects many paths without sufficient review.

Defense:

centrality scoring
phase-regime control
review routing
simulation

21.10 Generated architecture drift

Repeated AI changes gradually erode architecture.

Defense:

architectural drift state
coherence scoring
package crystallization forecast
refactor governance

---

22. Security Invariants

Code Medium preserves the following invariants.

Invariant 1 — Code changes are substrate mutations

Every meaningful code change alters the future response of the software medium.

Invariant 2 — Compilation is not safety

A compiling change may still be topologically unsafe.

Invariant 3 — Tests are not enough unless topologically close

A test must be close to the changed object and affected path to count as strong evidence.

Invariant 4 — Policy must be on-path

Policies do not protect paths they do not govern.

Invariant 5 — External writes require governed adapters

Direct unmediated external writes from protected objects should be blocked or repaired.

Invariant 6 — Interface contracts are governance objects

A contract violation is a substrate obstruction, not merely a type mismatch.

Invariant 7 — Package changes are topology changes

Packages alter trust, reachability, ownership, and deployment behavior.

Invariant 8 — Schema changes are substrate memory mutations

Schema changes alter how the system stores, retrieves, and interprets state.

Invariant 9 — Deployment is a protected action

Production release requires release proof.

Invariant 10 — Receipts prove code substrate decisions

Logs are insufficient for high-assurance AI code release.

Invariant 11 — Replay sustains release trust

A deployment can remain trusted only if its release decision can be replayed.

Invariant 12 — Rollback is not full restoration

A faulty deployment may leave data, trust, customer, workflow, or external-system residue.

---

23. Product Architecture

TraceScript Code Medium includes the following major components.

23.1 Code Signal Normalizer

Converts commits, diffs, pull requests, agent patches, file edits, package changes, migrations, route additions, deployment requests, and test results into normalized code signals.

23.2 CodeTrace Indexer

Maps code objects into trace objects with role, location, coordinate, dependencies, effects, ownership, and receipt coverage.

23.3 Code Coordinate System

Assigns coordinates across file path, module region, runtime layer, domain, package, schema, route, service, test, policy, adapter, and deployment space.

23.4 Code Topology Engine

Builds and updates graph relationships among code objects.

23.5 Reachability Analyzer

Determines whether changed objects are reachable from protected entrypoints or can reach protected effects.

23.6 Effect Classifier

Classifies code effects such as read-only, internal mutation, memory write, workflow change, external write, financial effect, security effect, schema mutation, package mutation, and deployment effect.

23.7 Interface Contract Registry

Stores and verifies contracts between code objects, packages, schemas, routes, adapters, and external systems.

23.8 Test Proximity Engine

Scores whether available tests are close enough to changed paths and effects.

23.9 Package Integrity Engine

Evaluates package introduction, extraction, dependency drift, public surface expansion, and supply-chain risk.

23.10 Schema and Migration Integrity Engine

Evaluates schema changes, migrations, backfills, consumers, rollback feasibility, and downstream substrate consequences.

23.11 Policy-on-Path Engine

Verifies that required policies govern affected execution paths.

23.12 External Write Path Guardian

Detects direct or ungoverned writes to external systems and proposes adapter repairs.

23.13 Architectural Drift Monitor

Measures accumulated substrate drift from repeated changes.

23.14 Deployment Governance Gate

Determines whether a change may merge, deploy, stage, simulate, review, or block.

23.15 Release Artifact Service

Issues bounded release artifacts for approved deployments.

23.16 Code Receipt Ledger

Emits replayable receipts for code decisions, obstruction, repair, review, merge, deployment, rollback, and reconciliation.

23.17 Replay Verifier

Reconstructs release decisions from code version, topology, tests, policies, contracts, receipts, and artifacts.

23.18 Repair Planner

Provides typed repair operations.

---

24. Deployment Modes

24.1 Shadow mode

Code Medium observes AI code changes and reports what it would have blocked or required.

Use:

baseline risk mapping
AI coding pilot
architecture drift audit
test proximity analysis

24.2 Advisory mode

Code Medium comments on pull requests with topology warnings and repair suggestions.

Use:

developer adoption
low-friction rollout
agent-assisted code review

24.3 Review mode

Code Medium routes high-risk changes to owners or reviewers.

Use:

protected paths
package changes
schema changes
external writes
deployments

24.4 Enforcement mode

Code Medium blocks merge or deployment when gates fail.

Use:

production repositories
AI-generated code controls
security-sensitive systems

24.5 High-assurance mode

Code Medium requires release artifacts, replayable receipts, topology snapshots, retained test evidence, contract verification, rollback plans, and reconciliation.

Use:

regulated systems
financial software
healthcare software
security products
identity systems
production deployment pipelines

---

25. Reference Use Cases

25.1 Direct external write from component

An agent adds a database update inside a UI component.

Code Medium detects:

component trace
external write effect
missing adapter
missing policy on path
missing receipt
deployment risk high

Outcome:

block deployment
repair insert adapter
route to review
require test on governed path

25.2 Missing policy on API route

An agent creates a new route that writes customer data.

Code Medium detects:

route trace
customer-data domain
external write
authorization policy absent on path

Outcome:

block merge
repair add policy middleware
require route test
emit obstruction receipt

25.3 Package extraction incoherent

An agent extracts validation logic into a package.

Code Medium detects:

package boundary change
consumer mismatch
public surface expansion
test distance too high

Outcome:

stage
require contract
add consumer tests
route to owner

25.4 Schema migration risk

An agent changes a column used by billing workflows.

Code Medium detects:

schema mutation
financial path reachability
missing backfill
rollback uncertainty
test proximity failure

Outcome:

block deployment
require migration plan
require billing-path tests
require review

25.5 Generated tests too far from risk

An agent adds tests for a helper but changed route-level behavior.

Code Medium detects:

tests exist
test proximity low
protected path untested

Outcome:

require topologically close test
prevent false confidence

25.6 Deployment with incomplete release proof

An agent proposes deployment after CI passes.

Code Medium detects:

CI passed
topology changed
external write path introduced
receipt coverage missing
replay class insufficient

Outcome:

block release
require release artifact
emit deployment obstruction

---

26. Metrics

Core metrics include:

AI code changes observed
CodeTraces created
topology changes detected
protected paths affected
external write paths introduced
unguarded external writes blocked
policy-on-path failures

interface contract failures
test proximity failures
package drift incidents
schema drift incidents
deployment gates blocked
deployment gates passed
repair suggestions accepted
adapter insertions required
receipt coverage percentage
deployment replay success rate
rollback plan coverage
architectural drift score
high-centrality changes routed to review
false allow rate
false block rate
developer friction score
mean time to repair obstruction
audit export completeness

The most important security metric is:

Number of AI-authored code changes prevented from creating ungoverned protected execution paths.

The most important runtime metric is:

Percentage of protected code deployments covered by topology, contract, test, policy, receipt, replay, and rollback proof.

The most important developer metric is:

Percentage of blocked changes repaired through typed, accepted repair operators.

---

27. Evaluation Methodology

Code Medium should be evaluated across five dimensions.

27.1 Runtime correctness

Can it map code objects, compute topology, detect reachability, classify effects, verify contracts, evaluate test proximity, gate deployments, emit receipts, and replay release decisions?

27.2 Security efficacy

Can it detect ungoverned external writes, policy bypasses, contract violations, package drift, schema drift, missing tests, and deployment without proof?

27.3 Governance fidelity

Does it correctly represent owners, policies, protected paths, adapter requirements, review routes, deployment rules, and release burden?

27.4 Developer ergonomics

Does it produce actionable repair suggestions rather than opaque blocks?

27.5 Enterprise auditability

Can auditors reconstruct what changed, what paths were affected, why deployment was allowed, what tests proved it, what policies applied, what receipts were emitted, and whether replay verifies release?

---

28. Competitive Differentiation

Static analysis finds code patterns.

CI/CD runs pipelines.

Test tools run tests.

Dependency tools scan packages.

Code review tools manage review.

TraceScript Code Medium governs the software substrate.

It does not merely ask whether code compiles.

It asks whether the changed topology is safe.

It does not merely count tests.

It asks whether tests are topologically close.

It does not merely detect dependencies.

It asks whether package changes preserve substrate coherence.

It does not merely check routes.

It asks whether required policy is on the execution path.

It does not merely log deployment.

It emits replayable release proof.

It does not merely block.

It explains obstruction and proposes repair.

The moat is the lifecycle:

code signal
→ CodeTrace
→ coordinate
→ topology
→ reachability
→ effect class
→ interface contract
→ test proximity
→ package integrity
→ policy-on-path
→ deployment gate
→ release artifact
→ receipt
→ replay
→ reconciliation

That lifecycle is the product.

29. Limitations and Non-Claims

Code Medium does not make AI coding agents perfect.

It does not replace compilers, type systems, tests, static analysis, SAST, DAST, SBOMs, dependency scanners, human review, CI/CD, secure coding, or security engineering.

It does not guarantee perfect topology extraction.

It does not claim all code risk is graph risk.

It does not eliminate human architectural judgment.

It does not make rollback perfect.

It does not require every repository to adopt full high-assurance mode.

It does not claim all generated code is unsafe.

It claims that AI-authored code changes require substrate-aware governance before merge, deployment, and production effect.

That is the boundary.

---

30. Strategic Importance

Code Medium is commercially urgent because AI coding agents are entering the software factory before enterprises have a runtime for governing their architectural consequences.

Every serious enterprise adopting AI coding agents will need a control plane between:

agent wrote code

and:

code touched production

That control plane must understand more than diffs.

It must understand topology, contracts, packages, tests, policies, deployment gates, receipts, replay, and rollback.

Code Medium is that control plane.

Its wedge is concrete:

stop ungoverned AI code paths from reaching production.

Its long-term expansion is broad:

govern software substrate evolution.

Its strongest demo is obvious:

An AI agent writes code that compiles but creates a direct external write from a UI component. Code Medium detects the ungoverned path, blocks deployment, proposes adapter insertion, emits a receipt, and replays the decision.

The demo sentence is:

TraceScript blocked a compiling AI code change because it was topologically unsafe.

---

31. Conclusion

AI coding agents are accelerating software mutation.

They can generate working code. They can pass tests. They can explain their changes. They can open pull requests. They can increasingly deploy. But software safety is not reducible to local correctness.

A codebase is a living substrate. Its components, hooks, edge functions, schemas, policies, packages, tests, routes, adapters, migrations, deployment paths, and external writes form a topology that shapes future computation.

A code change can compile and still be topologically unsafe.

TraceScript Code Medium governs that gap.

It treats code objects as traces. It maps software geometry. It evaluates reachability, effects, contracts, tests, packages, schemas, policies, deployment gates, receipts, replay, rollback, and reconciliation. It blocks unsafe topology. It proposes repair. It emits proof. It provides a runtime boundary between AI-

generated code and production consequence.

Its core doctrine is:

A code deployment is a state-bearing external action; therefore AI-authored code must be governed as substrate before it can merge, deploy, or mutate external systems.

Its category message is:

AI Coding Agent Security.

Its product message is:

Govern AI code changes as substrate mutations.

Its winning sentence is:

A code change can compile and still be topologically unsafe.

That is the missing control plane for the AI software factory.

Appendix A — Compact Glossary

Architectural Drift
Accumulated divergence between intended software architecture and actual code topology.

Code Coordinate
A structured location of a code object across file, module, package, runtime, schema, policy, test, adapter, and deployment space.

Code Medium
The TraceScript product layer that treats a codebase as a state-bearing geometric software substrate.

CodeTrace
A trace representing a code object such as component, hook, route, schema, migration, package, test, adapter, policy, or deployment artifact.

Deployment Governance Gate
Runtime gate determining whether a code change may merge, deploy, stage, simulate, review, or block.

External Write Path
A path through code that can mutate an external system, database, SaaS object, customer record, workflow, memory, or deployment target.

Geometric Software Substrate
A codebase modeled as a topology of code objects, coordinates, paths, boundaries, policies, tests, contracts, and runtime effects.

Interface Contract
A governed expectation between code objects, packages, schemas, routes, adapters, or external systems.

Package Integrity
Governance of package introduction, extraction, dependency drift, public surface, ownership, and supply-chain trust.

Policy-on-Path
The property that required governance policy actually lies on the affected execution path.

Test Proximity
The degree to which tests are topologically close to the changed object, affected path, and risk effect.

Topological Reachability
Whether a changed object can be reached from protected entrypoints or can reach protected effects.

Topological Unsafety
A state in which code is locally valid but creates an unsafe software configuration through reachability, boundary, policy, contract, package, schema, or deployment effects.

Appendix B — One-Page Summary

TraceScript Code Medium governs AI code changes as substrate mutations.

It exists because AI coding agents can create code that compiles, passes tests, and looks reasonable while changing software topology in unsafe ways.

The core doctrine is:

A code deployment is a state-bearing external action; therefore AI-authored code must be governed as substrate before it can merge, deploy, or mutate external systems.

The winning sentence is:

A code change can compile and still be topologically unsafe.

Code Medium treats the codebase as a geometric software substrate.

Code objects become traces:

components
hooks
operators
edge functions
schemas
policies
packages
tests
migrations
API routes
adapters
deployment scripts
external write paths

Code Medium tracks:

coordinates
topology
dependencies
reachability
coherence
runtime reliability
test proximity
interface contracts
package integrity
schema drift
policy-on-path
receipt coverage
deployment risk

The canonical runtime path is:

code signal
→ CodeTrace
→ coordinate assignment
→ topology update
→ reachability analysis
→ effect classification
→ interface contract check
→ test proximity check
→ package and schema integrity check
→ policy-on-path check
→ deployment governance gate
→ release artifact
→ deployment receipt
→ replay
→ reconciliation

It detects:

direct external writes
policy bypasses
interface contract violations
test laundering
package drift
schema drift
ungoverned boundary crossings
deployment without proof
architectural drift
high-centrality mutations without review

The strongest demo is:

An AI agent writes code that compiles but creates a direct external write from a UI component. TraceScript detects the ungoverned path, blocks deployment, proposes adapter insertion, emits a receipt, and replays the decision.

The product message is:

A runtime between “agent wrote code” and “code touched production.”

End of TraceScript Code Medium — Canonical Public White Paper v1.0