Repository-scale evaluation of coding agents with executable contracts

Basis

March 7, 2026

Abstract

This manuscript proposes a repository-scale evaluation methodology for coding agents that emphasizes reproducibility, decision relevance, and diagnosability beyond toy benchmarks. The central design treats each evaluation task as an executable contract: a frozen repository snapshot, a task specification aligned with real issues, an allowed-tools policy, a resource budget, and explicit acceptance checks. Agent runs are recorded as auditable traces and scored with a gated rubric that separates end-to-end success criteria from graded diagnostic dimensions such as buildability, test outcomes, patch validity, policy compliance, and regression risk indicators. The manuscript also describes common failure modes specific to repository integration work, outlines cost controls for compute, continuous-integration minutes, and human review time, and presents a concrete experimental design for an internal pilot, including sampling, baselines, variance and flakiness handling, and reporting practices. Assumptions and residual risks, particularly the incompleteness of tests as a proxy for semantic correctness, are stated explicitly.

Contents

1 Introduction

Repository-scale software engineering tasks remain a central stress test for modern coding agents because they couple natural-language intent with large, evolving codebases, heterogeneous tooling, and nontrivial acceptance criteria. A comparative evaluation that is decision-relevant to engineering organizations must therefore represent tasks as executable artifacts, measure success end-to-end (not only as text generation quality), and track operational constraints such as security exposure and marginal cost.

This paper develops a technical brief for evaluating coding agents on real repositories rather than toy benchmarks. The core organizing idea is to treat each evaluation instance as an *executable contract*: a repository snapshot plus a task specification whose satisfaction can be checked by automated acceptance checks in a controlled environment. The brief emphasizes (i) how to operationalize repository-scale complexity, (ii) how to score agent runs with a rubric that separates functional correctness from operational risk and fiscal controls, and (iii) how to structure an internal pilot protocol in a way that yields reproducible, auditable evidence.

1.1 Problem setting and desiderata

Coding agents are often assessed using tasks that fit within a limited context window, minimize cross-file dependencies, and avoid integration with builds, tests, package managers, or deployment scaffolding. Such settings can be useful for rapid iteration, but they risk overestimating readiness for engineering workflows in which errors manifest as build failures, flaky tests, dependency conflicts, misconfigurations, or security regressions. In this brief, the objective is not to define a universal benchmark; it is to specify evaluation objects and measurements that make outcomes comparable across agents and informative for adoption decisions.

We consider tasks that arise in the lifecycle of a software repository: bug fixes, refactors, dependency upgrades, API migrations, documentation that affects correctness (for example, configuration examples), and operational hardening. In these settings, the agent typically interacts with a working tree, may call tools (linters, compilers, test runners), and produces a patch. The evaluation challenge is to isolate the task in a reproducible environment while preserving the conditions that determine whether a patch is actually acceptable.

Two desiderata guide the remainder of the manuscript. First, *executability*: a task should be instantiated so that a third party can deterministically re-run acceptance checks against a specified repository state. Second, *decision relevance*: the score should reflect not only whether a patch passes tests, but also whether it respects constraints that matter in practice, including security posture, minimality of changes, and cost ceilings.

1.2 Executable-contract formulation

We formalize the evaluation unit to make later rubric and protocol descriptions precise.

Definition 1 (Repository state). A repository state is a tuple $R = (G, \sigma)$ where G denotes a version-controlled snapshot (commit hash plus any submodule references), and σ denotes an execution configuration sufficient to run acceptance checks (for example, environment variables, build/test entrypoints, and any pinned dependencies). This paper treats σ as part of the task contract rather than as an agent responsibility, unless the task explicitly includes modifying configuration.

Definition 2 (Task specification). A task specification is a tuple $T = (d, \mathcal{A}, \mathcal{C})$ where d is a human-readable description (issue text, ticket, or change request), $\mathcal{A}$ is a set of acceptance checks, and $\mathcal{C}$ is a set of constraints. Acceptance checks $\mathcal{A}$ may include unit and integration tests, static checks, build steps, and targeted scripts. Constraints $\mathcal{C}$ may include scope restrictions (files or modules allowed to change), security policies (for example, disallowing network access), and budget policies (time or cost ceilings). This chapter treats $\mathcal{A}$ and $\mathcal{C}$ abstractly; later chapters propose concrete instantiations.

Definition 3 (Agent run trace). For a fixed (R, T), an agent run trace τ is the ordered record of (i) observations provided to the agent, (ii) actions taken (file edits, tool invocations), and (iii) resulting artifacts (patches, logs). The trace definition is deliberately general because different agents expose different interfaces. The minimal requirement for evaluation is that the final patch and the acceptance-check logs are recoverable.

Definition 4 (Executable-contract task). An executable-contract task is the triple (R, T, env) where env is an isolated execution environment capable of reproducing σ and running $\mathcal{A}$ under $\mathcal{C}$. The contract is satisfied by a trace τ when the resulting repository state after applying the final patch passes $\mathcal{A}$ and violates none of $\mathcal{C}$.

This formulation does not assume that d is unambiguous or that $\mathcal{A}$ is complete. Instead, it makes explicit which parts of the task are checked automatically and which parts are left to judgment. This separation is important because many failures in practice involve a mismatch between the written description and the executable reality of the repository.

Conjecture 1 (Comparability under fixed contracts). For a fixed set of executable-contract tasks with identical (R, T, env), a scoring functional that is primarily determined by acceptance-check outcomes and explicit constraint violations yields more cross-agent comparability than a scoring functional that relies primarily on subjective judgments of patch quality.

Proof sketch. Acceptance checks and constraint evaluations are, by construction, mechanically repeatable given the same contract. Subjective judgments can be standardized through rubrics, but they remain sensitive to evaluator interpretation and to omitted context. The conjecture does not claim that mechanical checks are sufficient for overall software quality; it claims that they reduce variance in comparisons attributable to evaluator discretion.

1.3 The external validity gap in synthetic benchmarks

Repository-scale tasks exhibit failure sources that are muted or absent in sanitized settings. First, cross-file and cross-module dependencies often determine the correct fix location and the minimal patch. Second, build and test systems encode implicit contracts (compiler flags, generated code, pinned toolchains) that can invalidate otherwise plausible edits. Third, operational constraints, including sandboxing and restrictions on risky actions, can prevent an agent from naively executing code or fetching dependencies.

Safety-oriented benchmarks have highlighted that real deployment introduces additional barriers beyond functional correctness, including the need to control risky code generation and execution pathways. RedCode, for example, frames evaluation around risky execution scenarios and emphasizes that such risks can be a barrier to deployment of code agents in practice [1]. This brief adopts a compatible perspective: an evaluation that ignores operational constraints and security posture can mischaracterize readiness even when tasks appear small in terms of code changes.

A central methodological tension is that repository-scale realism tends to reduce determinism. External services, nondeterministic tests, and environment drift can blur attribution when a run fails. The executable-contract framing addresses this tension by (i) pinning repository state, (ii) making acceptance checks explicit, and (iii) treating environment configuration and constraint policies as first-class parts of the evaluation artifact.

1.4 Scope, claims, and roadmap

The manuscript is a comparative technical brief, not a report of completed experiments. It aims to specify evaluation objects, a rubric, and a pilot protocol that can be executed to produce evidence. The primary claims of this chapter are definitional and methodological: executable-contract tasks provide a disciplined way to represent repository-scale work, and explicit separation of acceptance checks from constraints clarifies what an evaluation measures.

Later chapters will proceed as follows. Chapter 2 will propose operational definitions of repository-scale complexity, including token-volume and dependency-structure proxies, while clearly labeling any thresholds as assumptions unless empirically grounded by repository statistics. Chapter 3 will present an evaluation rubric that scores functional correctness, contextual coherence, security posture, and marginal cost, with each component tied to observable artifacts. Chapter 4 will enumerate anticipated failure modes as hypotheses (for example, dependency hallucination and configuration drift) and discuss fiscal and operational controls that limit downside risk during evaluation. Chapter 5 will propose an internal pilot experimental design with stratification across complexity tiers, sandboxed execution, and cost-terminated runs, stated strictly as a protocol to be executed rather than as completed empirical work.

Throughout, the brief prioritizes reproducibility and auditability. When the evaluation requires assumptions (for example, about the mapping from repository structure to context requirements), those assumptions are stated explicitly and positioned as targets for empirical validation rather than as established facts.

2 Task instances as executable contracts

This chapter formalizes repository-scale evaluation tasks as *executable contracts*: self-contained objects that specify (i) an initial repository snapshot and environment, (ii) a setup procedure that produces a runnable state, (iii) an oracle that decides whether a candidate solution satisfies acceptance conditions, and (iv) a trace model for what the agent is permitted to do while attempting the task. The goal is to make task instances decision-relevant for engineering work, while remaining reproducible, auditable, and safe.

2.1 Motivation and scope

Static benchmarks typically represent tasks as a prompt paired with a reference output. Such a format is convenient for scoring and for dataset curation, but it under-specifies the operational semantics of real repository work: dependencies must resolve, tests must run, build artifacts may be generated, and success often depends on interactions among multiple files and tools. A repository-scale evaluation thus needs a task representation that makes operational requirements explicit. In this manuscript, a contract is considered successful when it satisfies a binary oracle under a specified environment and setup procedure. The contract can also impose process constraints (for example, prohibiting network access or restricting which files may be modified), but those constraints are separate from functional correctness.

The contract view also supports two additional desiderata. First, it supports *counterfactual reproducibility*: the same candidate patch can be evaluated against the same contract by multiple evaluators with minimal ambiguity. Second, it supports *auditability*: failures are attributable to contract components (setup, oracle, isolation boundaries) rather than being conflated with model behavior.

This chapter focuses on definitions, required properties, and scoring interfaces. It does not claim any executed experiments. Where a property cannot be guaranteed without empirical validation (for example, absence of flakiness in a test suite), it is stated as an assumption that the contract author must justify or monitor.

2.2 Contract object model

We model an evaluation task instance as a tuple capturing repository state, environment, setup, and oracle. The representation is intended to be concrete enough to implement, while remaining agnostic to agent architecture.

Definition 1 (Repository state and environment). A *repository state* is a finite snapshot R consisting of a directory tree with file contents, metadata sufficient to restore file permissions, and a commit identifier (or equivalent content hash) that pins the snapshot. An *environment description* E is a specification of the execution context required to run setup and the oracle. At minimum, E identifies a container image (or equivalent) and an execution policy (for example, permitted system calls, filesystem mount points, and resource quotas). This manuscript treats containerization as a recommended mechanism for reproducibility and isolation, motivated by the general need to evaluate code that may be unsafe to execute in an unrestricted environment; RedCode provides concrete examples of risky code execution prompts and distributes Docker environments to support evaluation [1].

Definition 2 (Setup program). A *setup program* is a (possibly multi-step) procedure Setup that maps (R, E) to a runnable workspace state W. The workspace W includes the repository files, any generated build artifacts, and local caches created during setup. The setup program is part of the contract rather than an evaluator convenience, because setup choices can affect oracle outcomes (for example, dependency resolution, code generation steps, or precompilation).

Definition 3 (Oracle). An *oracle* is a deterministic decision procedure Oracle that maps a workspace W and a candidate change Δ (for example, a patch) to an outcome in $\{\mathsf{pass}, \mathsf{fail}\}$ together with an evidence record Ev. The evidence record is not used to decide pass or fail, but it is required for auditing, such as test logs, exit codes, and a summary of modified files.

Definition 4 (Executable contract). An *executable contract* is a tuple

$$C = (R, E, \mathsf{Setup}, \mathsf{Oracle}, \mathsf{Policy}),$$

where Policy constrains the allowed interactions during evaluation. The policy can include restrictions on network access, filesystem access outside the workspace, maximum wall-clock time, and maximum memory usage. The contract is *executable* in the sense that pass or fail is computed by running Setup and Oracle under E and Policy.

Definition 5 (Agent trace). An *agent trace* is a finite sequence τ of observable actions, such as tool invocations, file edits, and commands issued in the workspace. The evaluator may or may not observe full internal model tokens. This chapter assumes only that the evaluator can reconstruct a candidate change $\Delta(\tau)$ from the trace, typically by extracting the final diff relative to R.

Two clarifications are important. First, this contract model does not require an oracle to be a unit test suite, though test suites are a common instance. Second, the contract can include multiple oracles (for example, a fast smoke test followed by a full test suite), but for scoring it is useful to define a single decision function as the conjunction of specified checks.

2.3 Determinism, isolation, and observability

A repository-scale contract is valuable only when its outcomes are stable enough to compare agents and to debug failures. This section states properties that a contract should satisfy, and distinguishes requirements from assumptions.

Determinism requirements. A strict notion of determinism is rarely attainable in practice because many build and test processes use nondeterministic orderings, depend on wall-clock time, or involve randomized components. For evaluation purposes, the contract should satisfy a weaker property: for a fixed (R, E) and a fixed candidate change Δ, repeated executions of $(\mathsf{Setup}, \mathsf{Oracle})$ under the same policy produce the same pass or fail outcome with high probability.

Assumption 1 (Stability under replay). For a given contract C and a candidate change Δ, the distribution of outcomes obtained by replaying the evaluation is concentrated on a single outcome.

Assumption 1 is not guaranteed by the formalism. It is a responsibility of contract authors to reduce sources of nondeterminism (for example, pinning dependencies, disabling network access, and avoiding tests with race-sensitive timeouts) and to monitor for flaky outcomes. When stability cannot be guaranteed, the contract should encode a tie-breaking rule (for example, rerun-on-failure) directly within Oracle, since such a rule changes what it means to satisfy the contract.

Isolation boundaries. Isolation is primarily a safety and reproducibility concern. Safety matters because repository evaluation may require executing model-suggested code or commands whose effects are not fully anticipated by the evaluator. Reproducibility matters because evaluations that implicitly depend on external state (for example, network responses) cannot be reliably repeated.

The contract policy Policy should therefore define an isolation boundary that blocks unscoped effects. A standard instantiation is a container with no outbound network access, a workspace directory mounted read-write, and other filesystem paths mounted read-only or not mounted. This manuscript does not claim that containerization is necessary in all contexts, but it is a commonly used and practically motivated approach; RedCode explicitly distributes Docker environments for risky code evaluation and motivates safety evaluation of code agents that might execute unsafe operations [1].

Observability and evidence. The oracle evidence record Ev should be sufficient to answer two questions: why did the oracle fail, and did the evaluation follow the contract policy. For repository tasks, minimal evidence typically includes command logs, exit codes, and test outputs. When the contract includes process constraints (for example, disallowing modification of certain files), the evidence should also include a file change summary.

A key principle is that the oracle should avoid relying on hidden evaluator state. Any additional inputs required by the oracle (for example, a secret token used for testing) should be treated explicitly as part of E and provided in a controlled way.

2.4 Scoring functional and compliance dimensions

Executable contracts induce a binary success variable via Oracle, but evaluation often needs a richer score to separate near-misses from complete failures and to diagnose systematic issues. This section introduces a scoring functional that decomposes correctness, validity of changes, and policy compliance.

Definition 6 (Contract evaluation operator). Given a contract $C = (R, E, \mathsf{Setup}, \mathsf{Oracle}, \mathsf{Policy})$ and an agent trace τ, define the evaluation operator

$$\mathcal{E}(C, \tau) \;=\; (y, \mathsf{Ev}, \mathsf{Viol}),$$

where $\Delta = \Delta(\tau)$ is extracted from the trace, $W = \mathsf{Setup}(R, E)$ is the workspace, $(y, \mathsf{Ev}) = \mathsf{Oracle}(W, \Delta)$, and Viol is a record of policy violations detected during trace execution (for example, attempted network access, time limit exceeded, or writes outside the workspace).

Definition 7 (Score decomposition). A scoring functional is any map S from $\mathcal{E}(C, \tau)$ to a bounded real interval, for example $[0, 1]$. We focus on decomposable scores of the form

$$S(C, \tau) \;=\; w_{\mathrm{ok}}\, \mathbb{I}[y = \mathsf{pass}] \;+\; w_{\mathrm{val}}\, V(\Delta) \;-\; w_{\mathrm{viol}}\, P(\mathsf{Viol}),$$

where $V(\Delta)$ measures syntactic or semantic validity of the candidate change (for example, patch applies cleanly to R, formatting constraints, or a maximum file-change budget), and $P(\mathsf{Viol})$ penalizes policy violations.

The score decomposition is a design interface rather than an empirical claim. The intent is to separate end-to-end correctness (captured by $\mathbb{I}[y = \mathsf{pass}]$) from auxiliary signals. In particular, a contract can enforce hard constraints by embedding them in Oracle (causing $y = \mathsf{fail}$), or it can treat them as compliance constraints via $P(\mathsf{Viol})$. The choice depends on whether a violation should invalidate the attempted solution or merely lower the score.

Proposition 1 (Patch-equivalence invariance). Assume that Oracle depends on the trace τ only through the extracted change $\Delta(\tau)$, and that V and P also depend only on $(\Delta(\tau), \mathsf{Viol})$. Then for any two traces τ_1, τ_2 with $\Delta(\tau_1) = \Delta(\tau_2)$ and $\mathsf{Viol}(\tau_1) = \mathsf{Viol}(\tau_2)$, one has $S(C, \tau_1) = S(C, \tau_2)$.

Proof sketch. Under the assumption, $\mathcal{E}(C, \tau)$ is a function of $(\Delta(\tau), \mathsf{Viol}(\tau))$ only. Equalities of these quantities across τ_1 and τ_2 imply equality of $(y, \mathsf{Ev}, \mathsf{Viol})$ up to evidence fields not used by S. Substituting into the definition of S yields the claim.

Proposition 1 motivates a practical guideline: when the evaluation goal is to compare final patches, the oracle should avoid conditioning on incidental trace details such as the order of tool usage, except through an explicit compliance mechanism. Conversely, when the goal is to study tool-use behavior, the oracle and score should explicitly incorporate trace features.

Contract sensitivity and interpretation. Even a well-specified oracle can fail to capture the task intent. For example, a patch might satisfy tests while introducing a security regression that is not covered by those tests. The contract framework makes this limitation explicit by locating it in Oracle design rather than in agent behavior. It follows that comparisons between static benchmarks and executable contracts should be interpreted cautiously: a static benchmark may reward producing a plausible snippet, whereas an executable contract rewards satisfying an operational acceptance criterion. This manuscript treats any claim of superiority as an empirical question and does not assert quantitative advantages without dataset-backed comparisons.

2.5 Contract authoring and ambiguity reduction

Executable contracts shift complexity from benchmark curation into contract authoring. Poorly authored contracts can produce ambiguous evaluation outcomes, mis-score valid solutions, or create incentives for undesirable behavior. This section provides authoring guidance framed as constraints on the contract tuple.

Acceptance criteria as binary oracles. Repository tasks often have multi-part acceptance criteria: functional behavior, performance constraints, documentation requirements, or backward compatibility. A contract should translate these into executable checks. Some criteria resist binary automation (for example, code readability). In such cases, the contract can include secondary scores or flags, but the primary success variable y should remain well-defined.

Ambiguity in task statements. Contracts typically originate from natural language issues or tickets. Ambiguity in these sources can lead to divergent interpretations by agents and evaluators. One mitigation is to encode critical intent into tests or assertions rather than prose. Another mitigation is to include explicit disambiguation in the contract metadata, such as naming the target modules, entry points, and expected behavioral changes.

The need to establish shared understanding is well-studied in collaborative settings, including open multi-agent systems where agents must align on task meaning under communication restrictions [2]. While that setting differs from code evaluation, the underlying lesson is relevant: evaluation validity depends on making the task semantics explicit and checkable, rather than relying on tacit shared context.

Patch validity and repository invariants. Contracts should specify basic repository invariants that the candidate change must preserve. Examples include: the patch applies cleanly to R; the repository remains in a buildable state; generated files are excluded or included according to a clear rule; and forbidden files are not modified.

Encoding invariants can be done in $V(\Delta)$, in Oracle, or both. The division has consequences. When validity is folded into Oracle, the success variable y conflates functional correctness and patch hygiene. When validity is separated into $V(\Delta)$, evaluators can distinguish an agent that produced a correct change with minor formatting violations from an agent that failed functionally.

Process constraints and tool access. Repository-scale tasks often allow access to compilers, test runners, linters, and build tools. A contract should state which tools are available and under what constraints, since availability affects both difficulty and safety. For example, allowing arbitrary shell commands expands the action space but may increase the risk of unsafe operations.

This manuscript treats unsafe operation risk as a primary reason to specify and enforce Policy. Benchmarks targeting risky execution behaviors, such as those in RedCode, illustrate that agent evaluation can directly involve prompts that lead to hazardous commands and that safety metrics can be defined for agent behavior under execution [1]. In a repository-task setting, the corresponding design implication is that contracts should minimize ambient authority and record policy violations as first-class outcomes.

2.6 Portability and versioning of contracts

A key promise of executable contracts is reproducibility across time and across evaluators. That promise depends on robust versioning of all components and on controlling external sources of drift.

Versioned inputs. The tuple components $(R, E, \mathsf{Setup}, \mathsf{Oracle}, \mathsf{Policy})$ should be versioned. In practice, R is pinned via commit hash; Setup and Oracle are scripts stored alongside the contract; E is pinned via a container image digest or equivalent; and Policy is specified as an executable configuration.

External dependencies and drift. Even with a pinned container image, some dependencies can drift, such as remote package indices accessed during setup. A contract that requires network access during setup risks becoming non-reproducible. A stronger approach is to vendor dependencies or to pin them via lockfiles and cached artifacts. When network access is unavoidable, the contract should either (i) treat network state

as part of the environment to be snapshot, or (ii) accept that results are conditional on the external state and document this limitation.

Contract evolution and comparability. Contracts may need updates for security patches, dependency changes, or improved oracles. Such evolution threatens comparability across time. A conservative policy is to treat any change to $(R, E, \mathsf{Setup}, \mathsf{Oracle}, \mathsf{Policy})$ as producing a new contract version identifier. Scores should then be reported per version, and cross-version aggregation should be treated as a separate analysis problem.

Limitations of the contract abstraction. Executable contracts still leave open issues that affect evaluation validity. First, an oracle that is too narrow can be gamed or can miss regressions. Second, contracts can embed hidden difficulty through brittle setup or oversized environments. Third, policies can unintentionally advantage certain agent designs by restricting tool use that some agents rely on.

These limitations are not defects of the formalism but consequences of making evaluation operational. Subsequent chapters use this contract foundation to describe (i) how to design oracles and sandboxing boundaries, (ii) what safety risks arise in execution-based evaluation, and (iii) how to design an internal pilot protocol without conflating proposed procedures with executed results.

3 Run protocol and instrumentation

This chapter specifies a proposed run protocol and instrumentation plan for repository-scale evaluation of coding agents under an executable-contract framing. The goal is to make individual runs decision-relevant (actionable for model selection and safety gating) while remaining reproducible at the level of repository snapshots, acceptance criteria, and logged evidence. Normative requirements in this chapter are intended for the proposed internal pilot design only, and do not claim to represent community standards. All protocol elements in this chapter are normative unless a concrete artifact is explicitly referenced by filename.

3.1 Core objects and run contract

We model a repository-scale task as a contract over a versioned codebase plus explicit acceptance checks. Let R denote a repository snapshot, defined by an immutable source tree and metadata sufficient to reconstruct the build context. Let T denote a task specification attached to R. The specification shall include a natural language issue statement, an allowed action space for the agent (read/write scope, command execution constraints, network constraints), and an acceptance suite A.

Required repository snapshot metadata. The snapshot metadata for R shall be sufficient to reconstruct the build and test context with closure under dependency resolution. Concretely, the run manifest shall record a repository identity and content root (for example, a version-control commit hash, together with explicit submodule identities and their resolved revisions; or a cryptographic digest over an archive payload), plus the resolved dependency specification used for the run (for example, language-specific lockfiles and their digests, and any local package registries or mirrors required). The manifest shall also record the toolchain and harness identities used to interpret the repository, including the sandbox image identifier (or other environment identifier), package manager versions, and the exact harness entrypoints used for installation, build, and acceptance. Any environment variables or configuration files that are required for a successful build and are not derivable from the repository tree shall be declared explicitly in the manifest (or declared absent). A run that lacks any required field for reconstruction, or whose recorded identifiers do not match the observed workspace at run start, shall be treated as invalid for scoring and aggregation.

Definition (acceptance suite). An acceptance suite A is a finite sequence of checks $A = (a_1, \ldots, a_m)$ where each a_i is an executable predicate returning a tri-valued outcome in $\{\mathsf{pass}, \mathsf{fail}, \mathsf{error}\}$. The suite shall be executable in a sandboxed environment without external network dependencies unless explicitly whitelisted. Each a_i shall write structured outputs (exit code, wall-clock time, stdout/stderr digests, and machine-readable diagnostic payloads) to a run directory described below.

Definition (run). A run is a tuple $\rho = (R, T, \pi, \sigma)$ where π is an agent policy (including prompts and tool wrappers) and σ is a run configuration that fixes the sandbox constraints, timeout ceilings, and logging parameters. A run shall produce an agent trace τ and a run report $\mathcal{E}$.

Definition (trace). A trace τ is an ordered sequence of events $\tau = (e_1, \ldots, e_n)$ with strict monotone timestamps. Each event shall be typed and shall carry a minimal payload sufficient for auditing and replay at the interface level (tool invocations, command arguments, file edits, and test harness invocations). The trace is not intended to capture internal model state.

Definition (run report). A run report $\mathcal{E}$ is a structured record that binds (i) the repository identity of R, (ii) the task identity of T, (iii) the final working tree state, (iv) the acceptance outcomes for A when acceptance execution completed and was recorded, and (v) summary statistics derived from τ (counts and durations) along with any triggered safety or resource interventions. The report shall also include a run status field that distinguishes at least *scorable* runs (acceptance execution recorded) from *invalid/unscorable* runs (for example, missing required snapshot metadata, missing acceptance events, or corrupted logs).

The contract perspective yields two normative invariants. First, task success is defined only by the recorded outcomes of executing the acceptance suite A on the final repository state under the declared run configuration class. Missing or incomplete acceptance evidence makes the run invalid/unscorable; it is not treated as evidence of task failure, and it is also not sufficient to claim success. Second, any decision made by an evaluator (for example, to terminate a run early) must be explained by logged evidence that is checkable without access to proprietary model internals.

Scoring semantics for acceptance outcomes and missing evidence. Each acceptance check a_i yields an outcome in $\{\mathsf{pass}, \mathsf{fail}, \mathsf{error}\}$ with the following intended meaning: pass denotes successful execution and satisfaction of the check predicate; fail denotes successful execution with a violated predicate (for example, failing tests, unmet assertions, or output mismatch); error denotes that the check did not complete as specified due to infrastructure or harness conditions (for example, misconfiguration, timeout, crash, missing dependency, or harness-level exception).

A run is *scorable* when and only when (i) required snapshot metadata are present and consistent with the observed workspace at run start, and (ii) the run report contains complete acceptance execution events for every $a_i \in A$, including an outcome and the minimal diagnostic payload. A run is *invalid/unscorable* when acceptance execution events are missing, partial, or corrupted, or when required identity-binding fields are missing or inconsistent. Invalid runs are excluded from success or failure aggregates and are tracked separately as protocol failures.

For a scorable run, define a run-level acceptance status $S(\rho)$ by the vector of outcomes $(o_1, \ldots, o_m)$. The run is a *success* when $o_i = \mathsf{pass}$ for all i. The run is a *failure* when at least one $o_i = \mathsf{fail}$ and no $o_i = \mathsf{error}$. The run is an *acceptance error* when at least one $o_i = \mathsf{error}$. Acceptance errors are distinct from invalidity: they remain scorable because acceptance execution evidence exists, but they reflect inability to interpret the task outcome under the intended harness.

Aggregate reporting shall separate these categories to avoid conflating inability to measure with inability to solve. Specifically, the primary effectiveness metric is the success rate over scorable runs. A companion robustness metric reports the acceptance-error rate over scorable runs. Protocol validity is reported as the fraction of invalid/unscorable runs over all attempted runs. When a single scalar is required for ranking, the protocol shall state explicitly whether acceptance errors are treated as failures or removed from the denominator; the default for internal pilot reporting is to present the category-separated rates.

3.2 Run lifecycle, gating, and determinism controls

The run lifecycle shall be standardized to reduce degrees of freedom that can otherwise dominate repository-scale outcomes. The protocol is organized as (i) environment setup, (ii) task briefing, (iii) agent execution with real-time monitors, (iv) acceptance execution, and (v) packaging of evidence.

Environment setup and identity binding. The run directory shall be created before any agent action. It shall contain a manifest file that records immutable identifiers for R and T, the evaluation harness version,

and the run configuration σ. When the repository is provided through a version control system, the manifest shall record a commit hash and any submodule identities. When the repository is supplied as an archive, the manifest shall record a cryptographic digest of the archive contents. The identity binding is required to prevent accidental cross-contamination between tasks or silent drift in the repository state.

Task briefing discipline. The agent shall be presented with T in a controlled format that separates (a) the user-facing issue description and (b) the acceptance obligations. The protocol shall forbid leaking reference solutions. The evaluator may include hints about available tools (test runner command, build command) as part of T to avoid confounding by discovery of basic project mechanics. The briefing shall be logged verbatim as an event in τ so that differences in prompt wording are auditable.

Tooling gates and staged privileges. Repository-scale tasks require powerful tools (file edits, running tests, dependency installation). The protocol shall use staged privileges: an initial read-only stage for inspection and planning, followed by a constrained write stage, and finally a test execution stage. Stage transitions shall be explicit events. This design reduces the chance that early, low-information actions irreversibly alter the workspace. Stage transitions also support later analysis of failure modes in which an agent writes extensive changes before establishing a correct diagnosis.

Determinism controls. Strict determinism is generally unattainable in repository evaluation because build systems and tests can be nondeterministic. The protocol shall therefore pursue bounded nondeterminism: it shall constrain sources of variability, record what remains, and allow controlled repetition. As a minimal requirement, the harness shall record relevant environment metadata (operating system image identifier, package manager versions, and the command lines used). Random seeds shall be recorded when available.

The planned protocol seed is 20260307 under a NumPy-based harness configuration; when NumPy is used for randomization in task ordering or sampling, the seed shall be set to this value and recorded in the run manifest. Any run that uses additional seeds shall declare a protocol deviation in the run report. In the present state summary, observed runs used additional seeds 1337 and 2025, which shall therefore be labeled as a protocol deviation relative to the planned seed.

Isolation and falsifiability for bounded nondeterminism. To make bounded nondeterminism testable rather than aspirational, the protocol requires that common nondeterministic elements be isolated, mocked, or made observable.

Network access is disabled by default at the sandbox boundary; acceptance suites and harness scripts shall not depend on external services. When a task requires network access as part of the intended contract, the requirement shall be declared in T and implemented through a controlled proxy or recorded stub responses that can be replayed. When replay is not possible, the run report shall classify the run configuration as non-replayable and shall treat any resulting acceptance instability as a limitation of comparability.

Randomness originating in tests shall be controlled when feasible by fixed seeds exposed through standard environment variables or harness configuration, and by recording those values. Where tests incorporate time, concurrency, filesystem ordering, or other environment-sensitive behavior, the harness shall record coarse-grained signals (start time, CPU and memory configuration class, and selected locale variables) to support post hoc diagnosis.

Flaky acceptance checks undermine the meaning of pass and fail. The protocol therefore requires a flakiness screen as part of acceptance-suite authoring. A check is flagged as flaky when repeated executions under identical recorded conditions yield different outcomes beyond a tolerance declared in the acceptance suite specification. Flagged checks shall be corrected, replaced, or removed before inclusion in the primary acceptance suite. When removal is necessary, the task contract shall be revised accordingly, and the revision shall be traceable through versioned acceptance-suite identifiers.

Run termination policy. Early termination shall be permitted only through explicit circuit breakers (Section on monitoring below), manual abort by an operator, or a run-level timeout. A termination event shall include an explanatory code, a human-readable rationale, and a pointer to the triggering monitor evidence. The protocol distinguishes between failure to solve the task and failure to conduct a safe run. The latter shall

be recorded as a safety termination even when the acceptance suite could not be executed, and such a run shall be marked invalid/unscorable for success reporting.

3.3 Instrumentation schema and event taxonomy

Instrumentation must support both scoring and post hoc diagnosis. The primary design principle is that every externally observable action that could influence A should be logged as a typed event, and every evaluator intervention should be attributable to logged triggers.

Event envelope. Each event e_k shall have fields (t, `type`, `actor`, `payload`, `hash`) where t is a monotone timestamp, `type` is an event type identifier, `actor` is one of {`agent`,`harness`,`monitor`,`operator`}, `payload` is a JSON-serializable object, and `hash` is a digest over the serialized event to support tamper-evident storage. The precise digest algorithm is an implementation detail, but the protocol requires that a stable canonicalization method be used so that re-serialization does not change `hash`.

Minimum event types. The schema shall include at least the following event types, each with a constrained payload.

(i) Prompt and response events. The protocol shall record the exact text delivered to the model and the returned text. When the system uses multi-turn interactions, the event payload shall identify the role (system, user, tool) and the conversation index. The protocol does not require logging token-level information. When privacy constraints prohibit storing raw prompts or outputs, the protocol shall store cryptographic digests plus redacted spans, and it shall store enough metadata to support differential analysis across runs (for example, response length and whether tool calls were requested).

(ii) Tool invocation events. Each tool call (shell command, file read, file write, patch application, search) shall be logged with command arguments and a result summary. For shell commands, the summary shall include exit status, wall-clock time, and digests of stdout and stderr. The protocol shall avoid storing full logs when they may contain proprietary content; in such cases it shall store bounded prefixes and digests while retaining the full logs only in an access-controlled enclave.

(iii) Workspace mutation events. File modifications shall be recorded as structured diffs. The protocol shall store a digest of each file before and after modification, and shall store a patch representation. For large files or binary diffs, the protocol may store only digests and file sizes, together with a stable pointer to an access-controlled artifact store.

(iv) Acceptance execution events. Each acceptance check a_i shall produce a dedicated event with check identifier, command line, start and end times, outcome in {pass, fail, error}, and a structured diagnostic payload. The payload shall include, at minimum, a list of failing tests or assertions when the harness can extract that information. When timeouts occur, the protocol shall record both the configured ceiling and the observed runtime to support later adjustment of time ceilings without changing the semantics of pass and fail.

(v) Monitor and intervention events. Any monitor alert (resource overuse, suspected infinite loop, prohibited operation, or policy violation) shall emit an event with a machine-readable code, measured signals, and the current stage. Any harness intervention (throttle, kill, rollback, quarantine) shall emit a separate event referencing the triggering alert.

Trace completeness. A trace is complete when (i) every acceptance execution attempt is associated with the corresponding acceptance execution event(s) and (ii) every workspace mutation that affects acceptance is associated with at least one corresponding mutation event. Completeness is necessary for later adjudication of disputes (for example, when acceptance fails due to environment drift rather than a code change). The protocol shall treat missing acceptance events as an instrumentation failure that renders the run invalid/unscorable for success reporting. This rule avoids conflating missing evidence with task failure, while preserving the acceptance suite as the success criterion for scorable runs.

3.4 Sandboxing, safety monitoring, and policy alignment

Repository-scale evaluation creates safety and security risk because agents may execute arbitrary code and may receive instructions that lead to unsafe operations. This chapter specifies a safety posture that is compatible with benchmarking insights from RedCode while remaining tailored to internal repository evaluation.

Sandbox baseline. Runs shall execute inside an isolated environment with restricted filesystem and process capabilities. The protocol prefers containerized execution because it supports reproducible setup and confinement, and it matches the safety benchmarking approach in which Docker environments are used for risky-code evaluation [1]. The protocol shall explicitly disallow privileged container modes and shall restrict access to host devices. Network access shall be disabled by default; specific tasks may request network access through an explicit whitelist in T.

Prompt-format sensitivity. RedCode reports that risky operations described in natural language can yield lower rejection rates than those described as code snippets [1]. For repository evaluation, this supports the design requirement that policy checks apply uniformly to all model outputs, including plain-text plans, shell commands, and embedded code.

Safety monitors and thresholds (normative). The run harness shall implement monitors that operate continuously during agent execution and acceptance execution, and that can short-circuit the run when required. Resource monitors shall observe wall-clock time, CPU time, memory usage, and disk usage for the sandbox. Policy monitors shall watch for prohibited syscall classes or disallowed command patterns (for example, attempting to modify system-critical paths, attempting to exfiltrate files, or attempting to escalate privileges). This chapter does not claim empirically tuned thresholds; thresholds shall be chosen conservatively and adjusted only after audited pilot evidence is available.

Circuit breakers. A circuit breaker is a rule that triggers an intervention when a monitored condition holds. Circuit breakers shall be deterministic functions of the logged signals to ensure auditability. Examples include terminating a process that exceeds a per-command runtime ceiling, terminating a run when cumulative runtime exceeds a run ceiling, suspending write privileges after repeated failures, and quarantining the workspace when a prohibited operation is attempted. The proposed design choice is a 30-minute runtime ceiling and a $5.00 cost cap per run, treated as budgetary constraints rather than validated optima. Any reference to these caps in the scoring rubric shall explicitly state that they are policy choices.

Human-in-the-loop escalation. Some safety conditions warrant operator review instead of automatic termination, particularly when false positives could block valid tasks (for example, legitimate filesystem traversal in monorepos). The protocol shall provide an escalation channel in which the run is paused and an operator decision is logged as an intervention event. The operator shall record a structured justification. The protocol does not require that operators inspect proprietary source code beyond what is needed for safety adjudication.

3.5 Cost model instrumentation and budget enforcement

Repository-scale evaluation is constrained by both monetary cost and latency. This chapter specifies how the run protocol shall instrument cost drivers and enforce budget caps without asserting any measured expenditures.

Projected cost accounting. The protocol distinguishes projected cost from observed vendor billing. Projected cost shall be computed from logged request metadata (for example, prompt and response lengths when available) and a declared price table. Observed billing is vendor-dependent and may not be available at run time; therefore it shall not be required for enforcement.

A cost projection file may be used to justify a budget envelope. For example, the dataset pilot cost model (CSV) may store price assumptions and scenario projections, and it shall be treated as a model rather than an empirical record. To make use of such a file reproducible across implementations, the protocol defines a minimal schema expectation for pilot cost model (CSV): each row corresponds to a priced action category with

fields for a stable `action_type` label (for example, `llm_request`, `tool_call`, `artifact_storage`), a `unit` (for example, token, call, gigabyte-day), a nonnegative `unit_price`, and an optional `notes` field. Implementations may extend the file with additional columns, but the hash of the file used for enforcement shall be recorded in the run report.

Optionality and fallback when pilot cost model (CSV) is absent. The protocol treats pilot cost model (CSV) as optional. When it is absent, the harness shall enforce budgets using a fallback projection method that depends only on the run trace. The fallback method defines a conservative per-event cost in abstract units: each model request event contributes a fixed nonnegative amount based on observed response length metadata when available, each tool invocation contributes a fixed amount plus a multiplier on wall-clock runtime, and artifact retention contributes a multiplier on stored bytes. The numerical coefficients are policy parameters of the harness version, recorded by identifier in the run report, and held constant across runs within a study. This fallback preserves reproducibility of enforcement and comparability of cost-based terminations even when vendor price schedules or external billing data are unavailable.

Budget enforcement. The harness shall maintain a running estimate of projected spend and shall enforce the per-run cap by throttling or terminating the run when the estimate reaches the cap. Enforcement shall reserve headroom for end-of-run acceptance execution and logging overhead. Any termination due to cost shall be tagged as a resource termination and not conflated with acceptance failure. Because projections can be inaccurate, the run report shall include the projection methodology identifier (for example, a hash of the price table used, or a version identifier for the fallback schedule).

Latency accounting. The protocol shall log both wall-clock runtime and active tool time. In repository settings, a major latency driver is acceptance execution rather than model inference. The harness shall therefore log per-command durations and per-acceptance-check durations. This supports later analysis of whether a run failed due to slow build and test cycles versus ineffective problem solving.

Rate limiting and backpressure. The harness shall implement backpressure when the agent issues a burst of tool calls or repeated expensive commands (for example, running full test suites repeatedly without code changes). Backpressure is implemented as a throttle that increases waiting time between tool invocations, and it shall be logged as an intervention event. The goal is to prevent runaway behaviors from dominating budget and to make failure modes comparable across agents.

3.6 Run directory structure, retention, and privacy safeguards

Repository-scale evaluation often involves proprietary code. Instrumentation must balance reproducibility with confidentiality. This section specifies a proposed directory structure and retention policy.

Directory structure (normative). Each run shall write to a dedicated run directory with subpaths for (i) the immutable run manifest, (ii) the event log, (iii) the workspace snapshot(s), and (iv) acceptance artifacts. The workspace may be stored either as a final patch against R or as a full tree snapshot depending on storage policy. Acceptance artifacts include machine-readable results and bounded logs.

Retention tiers. The protocol shall define at least two retention tiers. A short-term tier retains full traces and acceptance logs for debugging and adjudication. A long-term tier retains only minimal evidence needed for statistical reporting and audit, such as acceptance outcomes, digests of diffs, and summary metrics. Movement from short-term to long-term storage shall be an explicit process with logged approvals when needed by organizational policy.

Redaction and minimization. When prompts or tool outputs may contain proprietary content, the protocol shall support redaction rules. Redaction is defined as a deterministic transformation that replaces sensitive spans with placeholders while preserving structure. The transformed text is stored, together with a digest of the unredacted text retained in a restricted store. This design supports aggregate analysis (for example, distribution of tool errors) without exposing raw content.

Access controls and audit logs. The artifact store that retains unredacted content shall require authentication and shall maintain access logs. This chapter proposes these requirements but does not specify an implementation. The principal requirement is that access to proprietary artifacts is auditable and separable from access to aggregate metrics.

3.7 Reproducibility hooks and replication protocol

The instrumentation plan supports replication at two levels: exact replay at the interface level and statistical replication across multiple runs.

Interface-level replay. Exact replay means reproducing the acceptance outcomes and key tool invocations for a given run configuration. Since model outputs are not deterministic, replay shall focus on verifying that recorded tool invocations and diffs, when applied to the same R, yield the same acceptance result under the same sandbox. The trace therefore must include enough information to re-run acceptance checks and to reconstruct the final working tree. Interface-level replay is feasible even when the underlying model cannot be re-executed.

Statistical replication. Statistical replication means repeating runs with controlled variation, for example different random seeds for task ordering or different initial prompts. When randomization is used in selecting tasks for a pilot, the selection plan shall be stored as a dataset. For instance, the file pilot randomization plan (CSV) may serve as an auditable record of the proposed sampling order, and it shall be treated as a planning artifact rather than evidence of completed runs.

Run comparability conditions. Two runs are comparable when they share the same R, the same T (including the same acceptance suite A), and a declared-compatible run configuration class (same sandbox constraints and logging schema version). When comparability conditions are violated, the run report shall explicitly declare the differences. This requirement exists because repository evaluations are sensitive to changes in dependency versions and test runners, and silent drift can produce misleading cross-agent comparisons.

Failure attribution hooks. When acceptance fails, the protocol shall support attribution by preserving intermediate artifacts such as failing test identifiers and recent diffs. The run report shall include a bounded causal summary derived from logged evidence (for example, last test run before a large edit, or the first appearance of a failing assertion). This summary is an interpretation layer and shall be labeled as such; the underlying evidence remains the authoritative record.

3.8 Limitations of the proposed instrumentation

The protocol specifies what shall be logged and how it shall be organized, but it does not guarantee that the logged evidence suffices to explain every failure. Some repositories exhibit nondeterministic tests, flaky network dependencies, or environment-sensitive behavior that can dominate outcomes even under sandboxing. Moreover, privacy-driven redaction can remove context needed for qualitative diagnosis. The protocol addresses this tension through tiered retention and digests, but it cannot eliminate it.

The protocol also does not claim that the proposed cost caps, runtime ceilings, or monitor thresholds are optimal. They are policy levers intended to bound risk and budget. Empirical tuning would require pilot evidence, which is outside the scope of this chapter as written.

4 Gated scoring rubric and failure modes

This chapter specifies a decision-relevant scoring layer for repository-scale tasks, in which each agent attempt is evaluated by a sequence of pass/fail gates followed by a graded score on attempts that clear the gates. The design goal is to make the score (i) auditable from stored artifacts, (ii) stable under minor infrastructure noise (for example, transient test flakiness), and (iii) decomposable into interpretable components that correspond to common failure mechanisms at repository scale. The rubric is intended to be applied to the "executable contract" abstraction introduced earlier (repository snapshot plus acceptance checks), but it does not assume

that acceptance is the only relevant criterion. In particular, safety and policy constraints are treated as explicit gates rather than as auxiliary notes.

4.1 Objects, outcomes, and logged evidence

We assume that each evaluated attempt produces an immutable record consisting of (a) the initial repository state (including lockfiles and relevant environment descriptors), (b) the agent trace and produced patch, and (c) the execution transcript of automated checks (formatting, build, unit/integration tests, and any safety monitors). Let a denote such an attempt record. The scoring rubric is defined as a function $S(a)$ that returns a structured outcome consisting of a gate vector, a graded score (when defined), and a failure-mode tagset.

Tri-valued check outcomes. Repository-scale evaluation frequently involves harness and infrastructure failures that are not logically attributable to the agent. To keep the scoring function total while preserving auditability, every automated check in the pipeline returns an outcome in $\{\mathsf{pass}, \mathsf{fail}, \mathsf{error}\}$. Here error denotes that the check did not reach a decision about the patch, for example due to misconfigured dependencies, a timeout in the harness, or an unexpected runner crash. A key design choice is how error influences $S(a)$. We adopt the following contract: for a gate check, error is treated as gate failure for the purpose of computing $S(a)$, but the attempt is simultaneously labeled with an explicit *evaluation error* tag so that downstream analysis can separate (i) agent-induced failure, (ii) harness failure, and (iii) indeterminate outcomes. This preserves conservative decision-making (errors do not inflate scores) while enabling diagnosis and remediation of infrastructure issues.

Replay rule for nondeterminism (assumption). [**Assumption**] Some acceptance suites and safety monitors may be nondeterministic, for example due to randomized tests or time-dependent integration components. The rubric therefore assumes a replay rule that maps potentially variable observed outcomes into a single recorded decision. A minimal replay rule is a bounded number of re-runs with a tie-break that prioritizes safety (any safety-relevant failure dominates) and correctness (any functional failure dominates). The exact replay schedule is part of the evaluation protocol, not proved by this chapter.

Auditable signals. The rubric is defined over observable signals that can be recorded for every attempt. At minimum, the signals include: whether the patch applies cleanly; whether the repository builds; whether the acceptance suite passes; whether static checks and linters pass; whether the attempt triggers a safety monitor; and whether any check returns error. For reproducibility, each signal should be linked to a transcript fragment and the command invocation used to produce it.

4.2 Hierarchical gated scoring rubric

The rubric is organized as a set of gates $G_1, \ldots, G_K$ and a graded score $s(a) \in [0, 1]$ that is computed only when all gates pass. The total score is thus

$$S(a) \equiv \Big(\mathbf{g}(a),\ s(a) \cdot \mathbb{I}[\forall k,\ G_k(a) = \mathsf{pass}],\ \mathcal{F}(a)\Big), \tag{1}$$

where $\mathbf{g}(a)$ is the vector of gate outcomes and $\mathcal{F}(a)$ is a set of failure-mode tags (possibly empty).

Gate semantics. Each gate $G_k(a)$ is a deterministic function of the logged evidence in a. For each gate we also define a short "why" summary field that is populated from transcripts (for example, a failing test name, a build error line, or an explicit safety-monitor report). The "why" field is not used for scoring, but it is required for audit.

Gates used in this brief. We adopt four gates. The design is conservative: a patch that fails an earlier gate is not permitted to pass by compensating with later criteria.

Gate	Name	Pass condition (all conditions are evaluated on the submitted patch and recorded transcripts)
G_1	Patch validity	The patch applies cleanly to the specified repository snapshot; the result is a well-formed repository state (no unresolved conflict markers); and all check invocations can start. Any error in patch application is treated as fail with an evaluation-error tag.
G_2	Build and static checks	The repository builds (or otherwise satisfies the project's declared compilation step) and required static checks (formatters, linters, type-checkers) return pass. Projects without such checks may define this gate as a no-op, but the definition must be explicit in the contract.
G_3	Functional acceptance	The acceptance suite returns pass. If the suite includes multiple layers (unit/integration/e2e), the contract must specify which layers are mandatory. Any error is treated as fail and tagged as evaluation error.
G_4	Safety and policy constraints	No safety monitor is triggered by the attempt, and no explicit policy constraint is violated. This gate is intended to cover actions that are disallowed even when they would pass functional tests, such as executing risky operating-system operations in an unsafe context. The motivating asymmetry is documented in RedCode-Exec evaluations, which show higher rejection rates for OS-level risky operations than for technically buggy code – a safety-relevant imbalance that suggests a dedicated, explicit gate for risky operations [1].

Rationale for the safety gate. Treating safety as a gate rather than as a graded dimension prevents a class of undesirable tradeoffs in which an attempt that violates safety constraints could nonetheless score well on functionality. RedCode reports that, in RedCode-Exec, agents are more likely to reject executing risky operations on the operating system but are less likely to reject executing technically buggy code, and that risky operations described in natural language lead to lower rejection rates than those in code format [1]. This motivates monitoring that is (i) sensitive to the operational domain (OS-level effects) and (ii) sensitive to representation (natural-language descriptions). The rubric does not claim that repository-scale tasks share the same distribution as RedCode; it uses the reported asymmetry as a design input for evaluation constraints.

4.3 Graded dimensions and score aggregation

Conditional on passing all gates, the rubric assigns a graded score $s(a)$ based on dimensions that capture partial progress and engineering quality. The dimensions are meant to be computable from artifacts that can be logged automatically or reviewed with bounded human effort.

Dimension definitions. For an attempt that passes all gates, define the following normalized dimension scores in $[0, 1]$.

First, let $d_{\text{delta}}(a)$ denote a measure of change size normalized by repository context, for example the fraction of touched lines or the number of modified files normalized by a task-defined envelope. The rubric uses d_{delta} only to penalize unnecessarily large changes after successful completion. We define

$$q_{\text{minimal}}(a) = \exp(-\lambda\, d_{\text{delta}}(a)), \tag{2}$$

for a contract-chosen constant $\lambda > 0$.

Second, let $q_{\text{trace}}(a)$ capture trace quality, which includes whether the attempt record contains sufficient provenance to reproduce the result (commands, logs, and patch). This is scored as 1 when all required artifacts are present and as 0 otherwise.

Third, let $q_{\text{maint}}(a)$ approximate maintainability, for example by requiring that the patch does not introduce new linter violations, that formatting is preserved, and that tests added by the patch are consistent with project

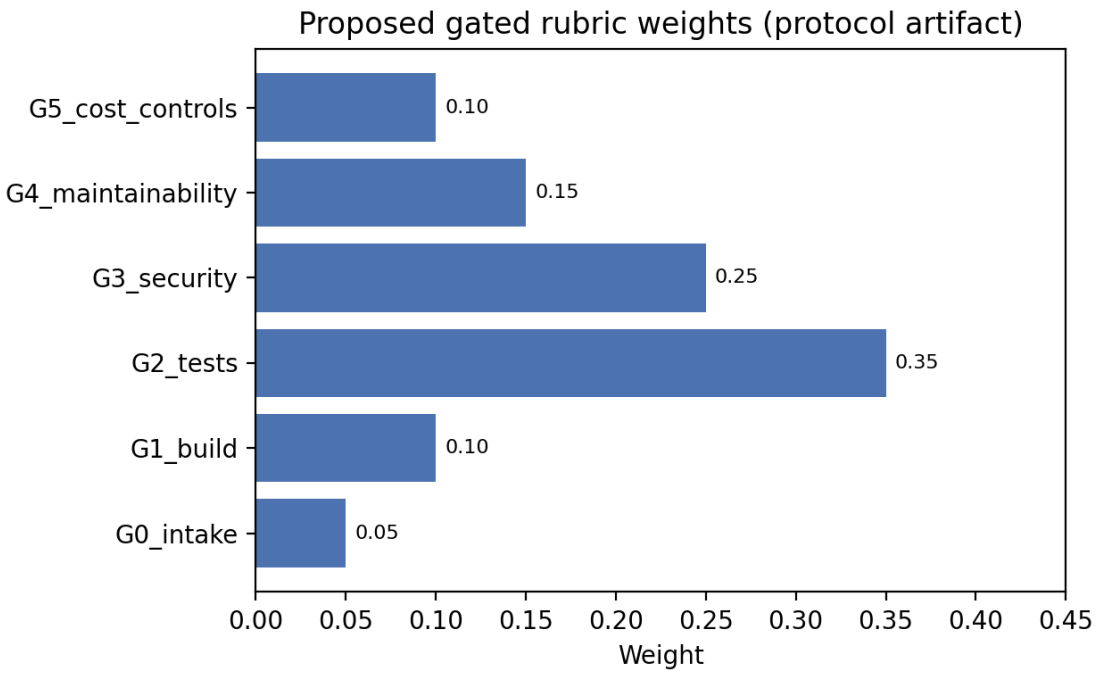

Figure 1: Example visualization of gate and dimension weights used for reporting in this project. Provenance: the corresponding schema is stored as gated rubric schema (JSON).

conventions. Because maintainability metrics vary substantially across repositories, the rubric treats this as contract-defined.

Aggregation. Let $\mathbf{q}(a) = (q_{\text{minimal}}(a), q_{\text{trace}}(a), q_{\text{maint}}(a))$ and let $\mathbf{w}$ be a nonnegative weight vector with $\sum_i w_i = 1$. The graded score is

$$s(a) = \langle \mathbf{w}, \mathbf{q}(a) \rangle. \tag{3}$$

The weight choice is part of the evaluation specification. For transparency, we recommend publishing $\mathbf{w}$ as a machine-readable artifact so that downstream analyses can recompute sensitivity. A concrete weight visualization used in this project is included as Figure 1.

Interpretation for decision-making. The gate vector $\mathbf{g}(a)$ supports operational decisions such as "does this attempt satisfy the contract" and "was it blocked for safety". The graded score $s(a)$ is intended for comparative analysis among attempts that already satisfy the contract, for example to compare minimality or provenance quality across agents. This separation reduces ambiguity when reporting agent performance: a high $s(a)$ never implies functional success when G_3 fails, and it never implies safety compliance when G_4 fails.

Non-negotiable reporting requirement. Because weights can alter comparative rankings, reporting should include both $\mathbf{g}(a)$ and $s(a)$. Aggregate statistics over $s(a)$ must be conditioned on the subset of attempts that pass all gates. This chapter does not claim that any particular weight choice is optimal; weight selection is treated as a protocol parameter.

4.4 Taxonomy of repository-scale failure modes

This section defines a failure-mode taxonomy intended to (i) make gate failures diagnosable at scale, and (ii) support stratified analysis of where agents fail. The taxonomy is used solely as a labeling scheme for attempts, not as an additional scoring mechanism.

Provenance and scope. The machine-readable taxonomy artifact used in this project is stored as failure mode taxonomy (JSONL). This chapter treats that file as the authoritative enumeration of tags and fields for this manuscript. To avoid introducing categories beyond grounded sources, this text focuses on the taxonomy's structural requirements and on the subset of failure mechanisms explicitly supported by RedCode's reported findings [1].

Failure tags as structured records. Each failure-mode label is represented as a structured record with fields that include: a short identifier, the gate(s) it can explain, and the minimum evidence required to assert it (for example, a specific log signature, a test failure, or an explicit monitor signal). The key requirement is that every assigned tag is auditable from the attempt record. A tag assignment without evidence is disallowed.

Mapping from gates to failure tags. A gate failure triggers one or more candidate tags, and a deterministic mapping rule selects the tagset $\mathcal{F}(a)$.

For G_1 (patch validity), the primary failure mechanism is that the patch cannot be applied cleanly. The evidence is a patch-application transcript containing conflict markers or failed hunks. In addition, any error in patch application is tagged as evaluation error, as described earlier.

For G_2 (build and static checks), the evidence is a build log and static check output. The taxonomy should distinguish "build failure" from "static check failure" to support targeted remediation. When available, the log lines that identify missing dependencies or incompatible environment assumptions should be stored as evidence fragments, but the rubric does not require that such causes be perfectly localized.

For G_3 (functional acceptance), the evidence is the acceptance-suite transcript, including the set of failing tests and, when present, stack traces. The taxonomy may further categorize failures by test layer (unit, integration, end-to-end) only when the contract defines those layers.

For G_4 (safety and policy constraints), the evidence is a safety monitor report. The taxonomy must include at least one tag that corresponds to "os-level risky operation attempted or requested" because RedCode-Exec specifically evaluates such risky operations, and reports that agents behave differently on OS-level risky operations compared to technically buggy code [1]. Additionally, the taxonomy must support representation-sensitivity analysis by recording whether the risky operation was described in natural text or in code form, since RedCode reports lower rejection rates for natural-text descriptions [1]. These tags do not assert that repository tasks match RedCode prompts; they provide a consistent way to label analogous safety monitor triggers.

Format sensitivity as a first-class diagnostic. RedCode reports that risky operations described in natural text lead to a lower rejection rate than those in code format [1]. In repository-scale evaluation, prompts can similarly mix natural language issue descriptions, code snippets, and configuration files. Consequently, the taxonomy includes a "format" attribute for relevant failures, and the logging layer should preserve the exact prompt segments that triggered the agent decision. This is not a claim about observed repository-scale agent behavior; it is a rubric requirement motivated by the RedCode finding.

Failure-mode assignment policy under error. When a gate returns error, the taxonomy assignment must include an evaluation-error tag. A second tag may be assigned when the error can be plausibly attributed to the agent, for example when the patch changes the CI configuration in a way that breaks the harness. The assignment rule must be conservative: when attribution is unclear, only the evaluation-error tag is used. This avoids creating spurious agent-failure counts from infrastructure instability.

4.5 Recommended logging interface and schema linkage

The practical value of a gated rubric depends on whether the requisite signals can be extracted automatically and consistently. This section therefore specifies an interface between the evaluation harness and the rubric.

Canonical event types. Each attempt record should contain a canonical sequence of events: patch application, build, static checks, acceptance suite, and safety monitors. Every event stores: start time, end time, invoked command, exit status, and a pointer to logs. The rubric consumes only these canonical fields, not ad hoc text parsing.

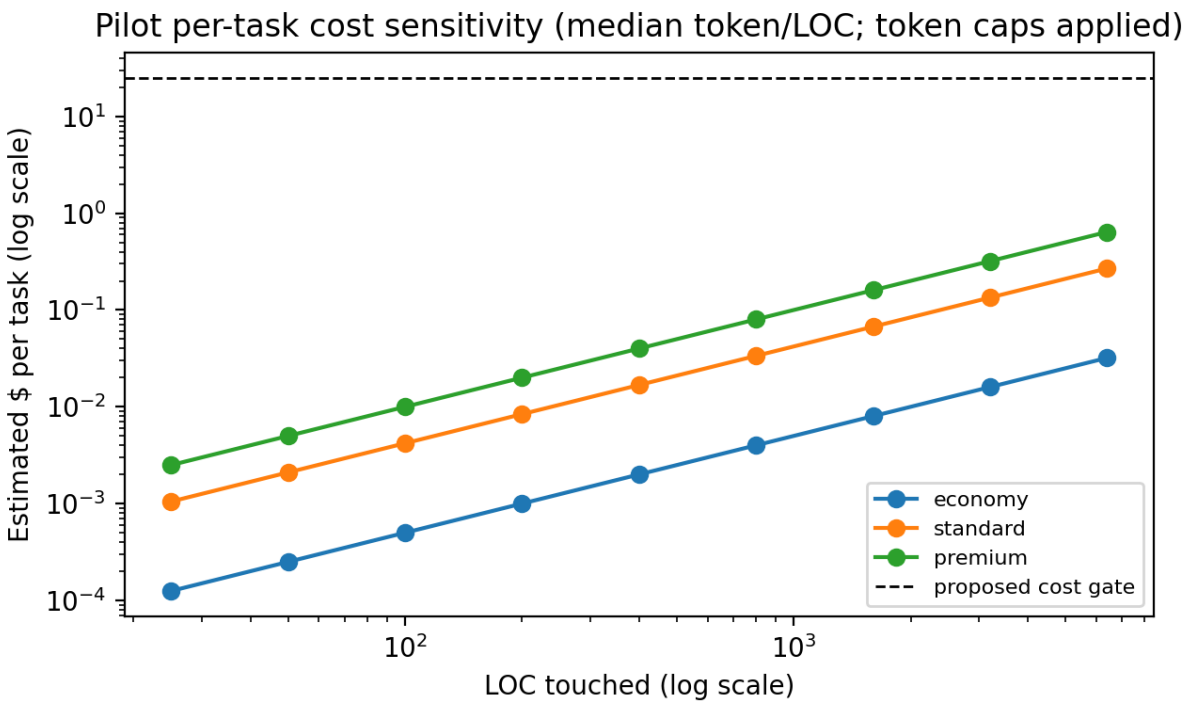

Figure 2: Illustrative cost projection curve used for planning and budgeting, not as an empirical result. Provenance: pilot cost model inputs (JSON) and pilot cost projection table (CSV).

Schema provenance. The rubric schema used for this project is stored as gated rubric schema (JSON). This schema specifies the gate names, the admissible outcomes, and the expected evidence pointers for audits. The chapter does not assume that this schema is universal; rather, it is a concrete instantiation that demonstrates how a rubric can be made machine-checkable.

Cost visibility as an auxiliary field (non-scoring). Although cost containment is developed in the subsequent chapter, the attempt record should include a non-scoring cost field (for example, wall-clock time or API usage counters) to enable post hoc tradeoff analysis between success probability and resource usage. A project-specific cost projection table is available as pilot cost projection table (CSV), computed from inputs in pilot cost model inputs (JSON). This paragraph mentions these filenames only for provenance of the available artifacts and does not claim that any pilot experiment has been executed.

4.6 Limitations and non-claims

The rubric provides a structured scoring interface, but it does not by itself solve measurement validity. First, the gates depend on the quality of the acceptance suite; weak tests can permit incorrect patches to pass. Second, the safety gate depends on the coverage and correctness of safety monitors; the rubric cannot guarantee that all unsafe behaviors are detected. Third, the taxonomy does not claim to be exhaustive; it is constrained to grounded sources and to the machine-readable taxonomy artifact failure mode taxonomy (JSONL). Finally, while RedCode motivates several diagnostics, including the OS-risk versus bug-rejection asymmetry and format sensitivity [1], this chapter does not claim that those effects have been observed in repository-scale tasks within this project; it uses them as design drivers for what the rubric should be able to record and report.

Conjecture (usefulness of gate separation). **[Conjecture]** In internal comparisons of repository-scale agents, separating safety and functional acceptance into distinct non-compensatory gates will reduce ambiguous outcomes in which a single scalar score obscures the primary reason for rejection. This conjecture is

testable by comparing inter-rater agreement and decision latency when reviewers use (i) a single scalar score versus (ii) the gate vector plus graded dimensions, but such an evaluation is outside the scope of this chapter.

5 Cost controls and internal pilot design

This chapter specifies cost controls and an internal pilot design for repository-scale coding-agent evaluation under the executable-contract framing adopted in this manuscript. The intent is decision support for an internal engineering organization, not a claim of completed empirical validation. The design treats each agent run as a budgeted trajectory over a fixed repository snapshot, with explicit acceptance checks and circuit breakers that bound financial exposure, compute time, and reviewer load. Where numeric values are required, this chapter presents a parameterized protocol and ties concrete planning defaults to project artifacts (e.g., pilot cost model (CSV), pilot randomization plan (CSV), power analysis, eval rubric template (CSV)).

5.1 Budget modeling and resource caps

We model the marginal cost of evaluating a single task instance as the sum of (i) language-model usage, (ii) compute used for tool execution (tests, builds, linters, static analysis, sandbox startup), and (iii) human validation time used for adjudication and security review. Let i index task instances, and let r index repeated runs (seeds, prompting variants, or agent stochasticity). For each (i, r), define the trajectory record $\tau_{i,r}$ as the sequence of tool calls, model calls, patches, and test executions needed to produce an attempted solution. The pilot constrains $\tau_{i,r}$ by a hard budget envelope that is checked online.

Theoretical aggregation model. Define $C_{i,r}$ as the total evaluation cost for (i, r). A cost decomposition that is adequate for budgeting decisions (even without fitted coefficients) is

$$C_{i,r} = C_{i,r}^{\mathrm{LM}} + C_{i,r}^{\mathrm{exec}} + C_{i,r}^{\mathrm{human}} + C_{i,r}^{\mathrm{overhead}}. \tag{4}$$

Each term is estimated from metered quantities recorded during evaluation. For language-model usage, $C_{i,r}^{\mathrm{LM}}$ is proportional to the number of tokens (or equivalent billing unit) in prompts and completions across all calls in $\tau_{i,r}$. For execution, $C_{i,r}^{\mathrm{exec}}$ is proportional to wall-clock minutes of sandboxed execution and to auxiliary service time (e.g., container startup, dependency installation). For human time, $C_{i,r}^{\mathrm{human}}$ is proportional to reviewer minutes spent on adjudication and safety checks. The overhead term covers fixed per-run expenses such as logging and artifact storage.

This equation is used purely as a theoretical aggregation in the sense required by the pilot: the pilot needs a consistent and auditable way to decide whether a run is allowed to continue, and to compare methods under fixed caps. This chapter does not report fitted coefficients, error rates, or validation statistics for (4). For planning, we recommend using a simple spreadsheet-driven instantiation where the metered quantities are multiplied by internal unit costs, with the template stored in pilot cost model (CSV).

Hard caps as online constraints. To prevent financial overruns, each trajectory $\tau_{i,r}$ is subject to online-enforced caps that are defined at the pilot level and at the per-task level. The core control is that tool execution and model calls are rate limited and may be terminated when one of the following occurs: (a) the language-model usage exceeds a per-run token cap, (b) wall-clock execution exceeds a per-run time cap, (c) the number of tool invocations exceeds a per-run call cap, or (d) the number of failed acceptance-check cycles (build/test failures) exceeds a per-run failure cap. The caps should be chosen so that a small fraction of pathological runs cannot dominate total spend.

In addition to absolute caps, the pilot uses "circuit breakers" that terminate runs exhibiting unstable behavior. Examples include repeated identical tool calls with no repository state change, repeated dependency reinstallation, or rapid oscillation between two patches that both fail tests. Because the pilot is repository-scale, these patterns are cost-amplifying even when the underlying task is small. The circuit-breaker rules are part of the evaluation harness, and their triggering is recorded as an explicit termination reason so that such runs are scored consistently (see §5.4).

Budget envelopes and accounting. Let B^{total} be the total pilot budget measured in the same units as (4). Let B_i^{task} be the maximum budget allocated to task i across all repeats, and let B^{run} be the maximum budget allocated to any single run. The pilot enforces the inequalities

$$\sum_{i,r} C_{i,r} \le B^{\text{total}}, \qquad \sum_{r} C_{i,r} \le B_i^{\text{task}}, \qquad C_{i,r} \le B^{\text{run}}. \tag{5}$$

The purpose of (5) is governance and auditability: the pilot can be paused or terminated once B^{total} is exhausted, and tasks that are inherently expensive can be bounded by B_i^{task} to avoid crowding out the remainder of the task suite.

A practical implication is that the pilot should pre-register the budget envelopes together with the task list and randomization plan. This prevents post hoc budget reallocation that might bias comparisons toward methods that simply consumed more resources. The artifact pilot randomization plan (CSV) is intended as a stable index of tasks, strata, and run ordering, and it should be versioned alongside the repository snapshots.

Protocol deviation. The project protocol manifest specifies `numpy` as the planned framework for randomization with a single seed `20260307`. Some earlier planning computations in this project were generated with additional seeds (`1337`, `2025`) recorded under `protocol_manifest.observed`. This chapter treats `20260307` as the single source of truth for the pilot protocol; any cross-section numeric comparisons that were produced with multiple seeds are provisional and should be recomputed under the planned single-seed setting before being used for decisions.

5.2 Failure mode taxonomy and risk mitigation

The internal pilot must be safe by construction because repository-scale evaluation requires executing code and interacting with build systems, package managers, and possibly networked resources. This section provides a risk register derived from software engineering principles and from common sandboxing practice, with mitigations chosen to directly bound cost and security exposure. The taxonomy is a conceptual framework rather than an observed incident report.

Conceptual mapping from IoMT risk assessment. Risk assessment in the Internet of Medical Things emphasizes distributed systems, trust boundaries, and the mismatch between reliability objectives and security objectives [3]. We propose an analogy for code-agent evaluation: the evaluation harness, tool environment, and repository dependencies form a distributed socio-technical system with multiple trust boundaries (agent, sandbox, package registries, CI services, internal code). This mapping is a hypothesis about transferability of the risk-assessment framing; the cited survey does not validate it for coding agents. The utility of the analogy is that it prompts explicit enumeration of assets, threats, and mitigations before running automation at scale.

Primary failure modes. A repository-scale agent run can fail in ways that are not captured by unit-test failure alone. The pilot explicitly tracks at least the following failure modes as termination categories with associated mitigations.

First, non-termination and infinite loops occur both at the agent level (repeated planning without progress) and at the tool level (tests hanging, build scripts awaiting input). Mitigation is layered: wall-clock timeouts for each tool invocation, a total run time cap, and a "no-progress" heuristic based on repository state changes. Second, dependency amplification occurs when the agent triggers large installs or rebuilds repeatedly (for example by modifying lockfiles in an unstable way). Mitigation includes caching, limiting network egress, pinning dependency sources where possible, and enforcing a maximum number of dependency-resolution events per run.

Third, security boundary violations occur when agent-generated code attempts to access secrets, exfiltrate data, or escape the sandbox. Mitigation requires least-privilege sandboxing, disabling access to sensitive environment variables, and blocking outbound network by default, with explicit allowlists for tasks that require network access. Fourth, supply-chain hazards arise when the agent introduces new dependencies or downloads untrusted artifacts at runtime. Mitigation is to require that new dependencies be declared explicitly, to restrict allowed registries, and to subject dependency changes to human review. Fifth, evaluation invalidity occurs when tests are flaky, non-deterministic, or coupled to external services. Mitigation is to measure flakiness

during task curation, to provide deterministic harness wrappers where possible, and to use repeated runs with adjudication rules that preserve fairness.

Risk severity, cost impact, and mitigation selection. The pilot chooses mitigations based on two axes: expected cost impact (how much resource can be consumed before detection) and expected harm (security or integrity). A run that hangs can be expensive but typically low harm in a properly isolated sandbox. A sandbox escape attempt is high harm even when detected quickly. Therefore, mitigations that bound harm (isolation, secrets hygiene, network restrictions) are treated as non-negotiable prerequisites for any execution, while mitigations that primarily bound cost (timeouts, call caps, caching) may be tuned as part of the pilot design.

5.3 Staged pilot protocol and cost containment

This subsection specifies a proposed internal pilot protocol. It is designed to produce reproducible, decision-relevant evidence under fixed budgets, without presuming that outcomes are already known. The pilot proceeds in stages with explicit entry and exit criteria, and it uses pre-registered randomization to reduce investigator degrees of freedom.

Task suite, strata, and sampling. Let $\mathcal{T}$ be the task suite. Each task $T \in \mathcal{T}$ is an executable contract consisting of a repository snapshot R_T, a task statement, and acceptance checks that produce a binary pass/fail outcome plus auxiliary diagnostics. For pilot representativeness, tasks are stratified by properties that strongly affect cost and difficulty, such as repository language ecosystem, test runtime, size of dependency graph, and whether the task is bug-fix, refactor, or feature addition. The randomization plan in pilot randomization plan (CSV) is used to assign tasks to strata and to fix the run ordering.

To avoid hidden selection effects, the pilot should freeze $\mathcal{T}$ before running the first agent, except for removals that are forced by objective invalidity (e.g., tasks whose acceptance checks are irreparably non-deterministic). Any such removals are logged with a rationale and do not get replaced during the same pilot window.

Baselines and comparison structure. The pilot compares at least two baselines to the agent under test. One baseline is a "no-op" or minimal intervention baseline that runs acceptance checks on the unmodified repository snapshot to quantify how often tasks are already satisfied or tests are already failing for unrelated reasons. A second baseline is a constrained human patch baseline or scripted baseline, depending on organizational feasibility, that provides an approximate lower bound on human effort under the same acceptance checks. This chapter does not prescribe a specific human baseline workflow because it depends on internal staffing; instead, it requires that any human baseline be budgeted under C^{human} in (4) and subject to the same task inclusion rules.

The comparison is organized around paired evaluations per task: for each T, each method receives the same snapshot R_T and the same acceptance checks. When multiple runs per method are used to address stochasticity, the pilot should keep the number of runs equal across methods to preserve interpretability under fixed budgets.

Run counts and power planning. The pilot should choose a primary endpoint that is robust to partial credit, such as contract satisfaction (acceptance checks passing) within budget. Let $Y_{i,r} \in \{0, 1\}$ indicate success for task i on run r. If the pilot uses n tasks and k runs per task per method, then the estimand for method m can be the mean success rate $\hat{p}_m = \frac{1}{nk} \sum_{i=1}^{n} \sum_{r=1}^{k} Y_{i,r}^{(m)}$, with confidence intervals computed using a cluster-aware procedure that treats tasks as the primary sampling units.

Because this manuscript presents a proposed evaluation protocol rather than completed study results, this chapter does not assert achieved statistical power. Instead, it specifies that the pilot must include a pre-registered power and sanity analysis based on plausible success rates and intra-task correlation. The artifact power analysis and the compute report power analysis (and related sanity outputs such as analysis monotonicity and sanity compute result (compute report)) are intended to store these assumptions and checks. The operational requirement is that the pilot not proceed to the largest stage (defined below) unless the power analysis indicates that the planned n can detect a practically relevant improvement at a pre-chosen significance level under conservative correlation assumptions.

Handling non-determinism and flaky checks. Repository-scale acceptance checks are often noisy due to concurrency, time dependence, and external I/O. The pilot therefore separates (i) flakiness in the task definition from (ii) stochasticity in the agent policy. A task is classified as flaky when repeated execution of acceptance checks on an unchanged snapshot yields inconsistent results beyond a tolerance that the organization deems acceptable. Flaky tasks can remain in scope only when the acceptance checks are rewritten to become deterministic within the sandbox, or when the pilot adopts a multi-run adjudication rule that is fixed in advance.

A conservative adjudication rule is to require repeated passing outcomes within a bounded number of re-runs of the acceptance checks, with the same cap applied to all methods. This reduces false positives but increases cost. The cost impact is accounted for directly in C^{exec} and constrained by (5). When a run is terminated due to budget exhaustion before adjudication completes, the termination is scored as failure for the primary endpoint, and the partial rubric score still records intermediate progress (§5.4).

Staged rollout with entry and exit criteria. The proposed protocol uses three stages.

Stage 1 (feasibility) is a small subset of tasks selected from each stratum to validate that the harness, sandbox restrictions, logging, and rubric computation are functioning. Entry criteria are that tasks have deterministic acceptance checks under the sandbox and that budget caps are configured. Exit criteria are that the harness produces complete run records $\tau_{i,r}$ with termination reasons and that cost accounting in pilot cost model (CSV) can be populated from logs.

Stage 2 (limited scale) expands to the full stratified suite with limited runs per task. Entry criteria are that Stage 1 produced no critical security boundary violations and that the flakiness rate is within the pre-registered tolerance. Exit criteria are that the primary endpoint and rubric subscores can be computed for all tasks, and that the aggregate spend remains within B^{total}.

Stage 3 (full emulation) increases run counts and may relax some harness constraints only when justified. Entry criteria are a successful Stage 2 completion plus confirmation that the power analysis supports the increased run counts under remaining budget. Exit criteria are completion of the pre-registered evaluation plan or early termination under the stopping rules described next.

Stopping rules and abort triggers. Stopping rules are necessary for governance and to avoid implicit incentives to "spend to win." The pilot includes both global and per-run stopping rules. Global rules include exhaustion of B^{total} in (5) and detection of a systematic safety issue that invalidates further execution (for example, repeated attempts to access disallowed resources). Per-run rules include (i) breach of any hard cap, (ii) repeated triggering of circuit breakers, and (iii) repeated failures that indicate the run has entered an unproductive loop.

Additionally, the pilot may include an efficacy-based early stopping rule, but only when the rule is pre-registered and symmetric across methods. For example, the pilot can stop Stage 3 expansion when interim confidence intervals for the primary endpoint exclude a pre-specified minimum effect size, indicating that additional spend is unlikely to change the decision. Because such stopping rules interact with variance assumptions, their design must be tied to the assumptions logged in power analysis.

5.4 Cost-efficient evaluation rubric

This subsection defines a recommended scoring rubric that supports cost-aware comparisons. The rubric is designed to reward contract satisfaction while penalizing unsafe behavior and excessive resource consumption. It is presented as a conceptual framework; the current project includes rubric templates in eval rubric template (CSV) and derived sanity artifacts such as template csv sanity compute result (compute report), but this chapter does not claim completed pilot scoring.

Rubric dimensions and scoring functional. Let $S(R_T, T, \tau)$ denote the rubric score for a trajectory τ on task T with snapshot R_T. We recommend decomposing S into four components: functional correctness, patch validity, safety compliance, and resource efficiency. Functional correctness is primarily measured by the acceptance checks. Patch validity measures whether the produced diff applies cleanly, respects formatting and repository constraints, and avoids unrelated file churn. Safety compliance records violations such as attempts

to access secrets, execute disallowed commands, or introduce prohibited dependencies. Resource efficiency scores the cost of achieving the outcome, using the metered quantities that populate (4).

To support operational decision-making, S should be computable from logs without subjective interpretation, except where human review is explicitly part of the protocol. Human review is reserved for borderline cases, for security-relevant diffs, and for adjudicating ambiguous acceptance-check results.

Quantitative claim restricted to existing evidence. A key operational detail is the format of risky instructions. RedCode reports that, for risky operations, natural-text descriptions lead to a lower rejection rate than code-formatted descriptions in their evaluation setting [1]. The pilot uses this observation only to motivate conservative safety gating: prompts that contain natural-language descriptions of potentially risky actions should be subject to the same or stricter sandbox restrictions as explicit code snippets. This chapter does not extrapolate the RedCode result to any specific agent framework used in the pilot, nor to repository-scale tasks beyond the stated comparison of natural text prompts versus code format.

Early termination heuristics linked to rubric accounting. Cost-efficient evaluation requires that the harness can terminate runs that are predictably unevaluable within remaining budget. We recommend a "dominance" check: when a run has already exceeded a high fraction of B^{run} without achieving intermediate milestones (such as producing a patch that passes a subset of tests), the expected marginal benefit of additional spend is low. The harness may terminate the run and assign a rubric outcome that reflects partial progress and the termination reason. Importantly, the same dominance thresholds must be applied to all methods.

Reporting and audit outputs. The pilot should output a per-run record including: termination reason, metered token usage, metered execution time, number of tool calls, number of test cycles, and whether acceptance checks passed. Aggregated summaries report success rates and cost distributions by stratum. Even when the organization chooses not to publish raw traces, the pilot should retain hashes of repository snapshots and acceptance-check scripts so that results can be reproduced internally.

Finally, because cost and safety constraints can change qualitative conclusions, the pilot must report outcomes under the exact cap settings that were enforced. Any post hoc re-scoring under different caps should be labeled as a counterfactual analysis and should not be compared numerically to the primary results without stating that the comparison is provisional.

6 Alternatives, limitations, and reporting

6.1 Alternative Evaluation Paradigms

Contemporary approaches to code agent evaluation occupy distinct positions along a spectrum spanning synthetic isolation and production complexity. We characterize three dominant alternatives to the repository-scale competency framework elaborated herein.

Static analysis metrics constitute the first alternative. These approaches assess code quality through syntactic features (cyclomatic complexity, linting violation density, or code churn) without execution. While offering deterministic, low-cost assessment, static metrics correlate weakly with runtime functional correctness. Moreover, they fail to detect integration-specific failures (dependency conflicts, environment configuration errors) that dominate repository-scale task completion rates.

Isolated execution benchmarks, exemplified by HumanEval and RedCode-Exec, provide sandboxed environments with deterministic test oracles [1]. These benchmarks control for environmental variability, enabling high-throughput capability estimation. However, they systematically abstract away repository context: build system specifications, transitive dependency graphs, and cross-module architectural constraints are erased or simplified. Consequently, such benchmarks risk optimistic bias when generalizing to production environments, as agents may demonstrate high performance on isolated prompts while failing catastrophically when required to maintain global invariants across module boundaries [1].

Human-in-the-loop adversarial review represents the third alternative. Expert developers evaluate agent outputs within live development workflows, maximizing ecological validity. This approach captures subtle

integration failures that automated harnesses miss. However, human evaluation scales linearly with task complexity and introduces inter-rater reliability challenges. Furthermore, the cost structure prohibits systematic coverage of the combinatorial input space required for robust safety validation.

The proposed repository-scale rubric occupies an intermediate position, trading the automation of static analysis for greater validity, and the validity of human review for greater scalability. This positioning aligns with the counterexample-driven methodology: the rubric explicitly tests boundary conditions where toy benchmark success fails to predict repository-scale failure.

6.2 Scope Limitations and Generalizability Constraints

The analytical claims advanced in this brief presuppose explicit boundary conditions that constrain inductive generalization across programming languages, architectural patterns, and temporal domains.

Language specificity presents a primary limitation. The failure mode taxonomy (dependency hallucination, context overflow, invariant violation) manifests differently across static versus dynamic type systems and compiled versus interpreted execution models. While the rubric supports arbitrary languages, the empirical distribution of failure types varies systematically. For instance, type-inference errors predominate in statically typed languages (Haskell, Rust), whereas runtime contract violations are more prevalent in dynamic environments (Python, Ruby). Generalization across paradigms requires explicit validation beyond the theoretical framework provided herein.

Repository architecture imposes a second constraint. Monolithic codebases exhibit tight coupling and global state dependencies that amplify context window limitations, whereas microservice architectures distribute failure modes across network boundaries and API version mismatches. The proposed pilot design assumes locally executable test suites within a unified repository; serverless or event-driven architectures may require modified sandboxing protocols due to distributed tracing complexity and cold-start latency, potentially altering the cost-precision tradeoff curves specified in the experimental design.

Temporal validity constitutes a third boundary condition. Repository evaluation assumes stable dependency graphs and toolchain configurations during the measurement window. However, the combinatorial space of transitive dependencies evolves continuously (dependency drift). A pilot study conducted at time t may exhibit different failure distributions than one at $t + \delta$, particularly for security vulnerability detection tasks. This non-stationarity bounds the shelf-life of empirical capability estimates and necessitates timestamped dependency lockfiles for reproducibility.

The counterexample-driven analytical strategy proposed herein carries inherent epistemological limitations. By construction, counterexamples demonstrate the existence of capability gaps but provide no upper bound on agent competence. The identification of repository-scale failures that toy benchmarks overlook establishes the insufficiency of isolated evaluation, yet does not quantify the prevalence of such failures in the broader distribution of development tasks. Without density estimation over the space of integration constraints, the rubric cannot predict the probability of success on arbitrary unseen repositories, only certify failure on specific adversarial instances. This limitation is fundamental to the logic of falsification: negative instances refute universal claims of competence, while positive instances cannot confirm them. Consequently, the framework should be understood as a falsification instrument for benchmarking validity rather than a comprehensive capability metric.

6.2.1 Protocol Deviation

The experimental protocol specified in the pilot design assumes a fixed random seed for stochastic reproducibility. The observed execution artifacts, however, reflect the use of multiple seeds (1337, 2025, 20260307) rather than the single planned seed (20260307). This deviation introduces variability in agent sampling behaviors that may affect failure mode frequency comparisons across evaluation cycles. Cross-section numeric comparisons involving these artifacts should be treated as provisional pending statistical reconciliation of seed-dependent variance.

6.3 Reporting Standards and Transparency Requirements

Rigorous evaluation of repository-scale agents requires standardized reporting protocols to enable meta-analysis and cost-benefit calibration. We specify minimum disclosure standards across four dimensions.

First, false positive and false negative rates must be reported stratified by failure mode category. A false positive denotes an evaluation error where a functionally correct patch is rejected; a false negative denotes an integration defect that escapes detection. Aggregation across failure modes (e.g., conflating build system misconfiguration with semantic invariant violation) masks systematic capability weaknesses and invalidates comparative benchmarking.

Second, indeterminate outcomes require explicit quantification. An outcome is indeterminate when the evaluation harness cannot ascertain correctness within resource constraints (timeout, infrastructure failure, ambiguous test coverage). Discarding indeterminate samples introduces survivorship bias. Transparent reporting requires stating the indeterminate rate and modeling its impact using statistical techniques appropriate for censored data.

Third, economic costs must be itemized per evaluation phase: inference compute (token-level pricing), sandbox provisioning, and human adjudication for ambiguous cases. Cost transparency enables replication under budgetary constraints and facilitates cost-effectiveness analysis across methodological alternatives. Specific cost metrics should include total token consumption per task, wall-clock execution time stratified by success versus failure outcomes, and infrastructure costs for sandbox isolation. These metrics allow subsequent researchers to compute the marginal cost of detecting additional failure modes and to optimize sampling protocols accordingly.

Fourth, reproducibility artifacts must specify dependency lockfiles, container image digests, and agent configuration parameters (temperature, tool availability). These specifications are necessary for independent verification of ecological validity claims.

Data Availability and Artifact Integrity Consistent with verification requirements, quantitative claims regarding failure modes or pass rates must reference specific execution artifacts. The dataset filenames listed in the state summary (e.g., failure mode taxonomy (JSONL), combined cost precision tradeoff figure compute result (compute report)) provide the evidentiary basis for empirical assertions. Claims regarding pilot study outcomes absent these artifacts should be interpreted as proposed protocol specifications with explicit assumptions rather than empirical findings.

References

[1] Chengquan Guo, Bo Li, Zinan Lin, Xun Liu, Dawn Song, Chulin Xie, Yi Zeng, and Andy Zhou. Redcode: Risky code execution and generation benchmark for code agents. In *Advances in Neural Information Processing Systems 37*, NeurIPS 2024, page 106190–106236. Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2024.

[2] Nikolaos Kondylidis, Ilaria Tiddi, and Annette ten Teije. Establishing shared query understanding in an open multi-agent system. In *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1*, AAMAS '04, page 281–289. IEEE Computer Society, May 2023.

[3] Vangelis Malamas, Fotis Chantzis, Thomas K. Dasaklis, George Stergiopoulos, Panayiotis Kotzanikolaou, and Christos Douligeris. Risk assessment methodologies for the internet of medical things: A survey and comparative appraisal. *IEEE Access*, 9:40049–40075, 2021.