

MRAC Bench: A Software Engineering Evaluation Method and Infrastructure Based on Multi-Round Audit Convergence

Liu Ming

September 16, 2026 | V1.0 | Final manuscript

Abstract

Software engineering evaluation needs a continuing supply of tasks that reflect real development, distinguish system capabilities, and remain affordable to construct. This paper proposes MRAC Bench, an evaluation method and infrastructure based on Multi-Round Audit Convergence (MRAC). Its central idea is to use audit-and-repair trajectories as a conditional signal of whether an AI system can achieve stable, effective coverage of the relevant search space under given task and resource constraints: consistently identifying and coordinating the constraints, dependencies, and verification requirements that matter for delivery. The system works toward a fixed objective, undergoes independent audits, and repeatedly repairs its output. Evaluation considers convergence within budget, process trajectories, and resource consumption. Audits can examine requirement and design specifications or execution artifacts such as code changes.

Candidate tasks are generated from real repositories and change requests, then screened through runs that calibrate difficulty and the applicable capability range. Construction does not require a complete reference implementation or a dedicated acceptance verifier to be prepared for every task, potentially lowering the barrier to task creation substantially. Varying change scope, cross-module effects, and combinations of constraints also offers a way to keep constructing tasks near system capability boundaries. As long as real engineering continues to provide more complex tasks that can be audited effectively, the evaluation mechanism may continue to scale with model progress.

To control the cost of reproduction at scale, the pilot study uses specification audits. The available records show differences among three model tiers in persistent non-convergence, gradual stabilization, and rapid stabilization. These observations do not directly establish actual coverage or a general capability ranking. Missed defects and differences in audit preferences may affect the signal's meaning. Comparability across models, repeatability, cost advantages, and extensibility to greater difficulty still require systematic testing. The paper also discusses potential industry implications and develops three classes of counterexamples concerning the direction of the signal, limits to difficulty scaling, and unavoidable task-construction costs.

Keywords: MRAC; software engineering evaluation; search space; effective coverage; multi-round audit convergence; task construction; evaluation infrastructure

1. Introduction

Real software engineering requires a system to understand code, interpret requirements, and coordinate interfaces, dependencies, and compatibility requirements as it makes changes. SWE-bench and FeatureBench already evaluate issue resolution and feature development in repositories. [1] [2] A further question follows: how can evaluation sustain a large enough supply of tasks that distinguish systems?

Development requirements are plentiful, but turning them into comparable, reproducible evaluations generally also requires an execution environment and acceptance criteria. For complex specifications, features spanning modules, and design changes, this preparation may constrain task supply. Model progress can also reduce the discriminative value of existing tasks, requiring continual updates. LiveCodeBench addresses renewal by collecting new programming-contest problems. Real engineering offers another possibility: actively constructing change requests of different difficulty within repositories. [3]

The development process itself can provide evaluation evidence. Given the same objective, some systems stabilize a design after a few audits and repairs, while others continue to expose omissions, conflicts, and problems introduced by repairs. This difference concerns a state that is difficult to observe directly: can the system now handle the key relationships in the task's search space consistently? MRAC uses changes in findings across rounds as a conditional process signal of coverage. [4] This paper asks whether that signal can support comparisons of models and agents while reducing the effort required to construct engineering evaluation tasks.

The central contribution is a reusable evaluation method combining multi-round convergence trajectories, low-cost task construction, and continuing difficulty calibration. The following sections explain the mechanism, protocol, and infrastructure, present pilot evidence collected with reproduction cost in mind, and discuss implications and counterexamples. The concepts needed to read the paper are explained here.

2. Search Space, Effective Coverage, and Multi-Round Audit Convergence

2.1 Search space and effective coverage

Completing an engineering task requires a system to distinguish among interpretations of requirements, relevant code and states, feasible implementations, and verification paths. The possibilities it still needs to judge and handle constitute the task's search space (S). Difficulty depends on coupled constraints, implicit dependencies, distributed state, and verification conditions; lines of code or context length do not directly measure it.

Effective coverage does not require enumerating every possibility. It requires identifying key conditions, preserving relevant constraints, ruling out decisive errors, and obtaining necessary verification. Coverage capability (C) belongs to the complete system comprising the model, agent, tools, environment, feedback, and budget. Delivery reliability (R) concerns whether the result satisfies key requirements, whether relevant effects have been verified, and whether it remains effective under expected conditions. The Search Space-Coverage Capability-Delivery Reliability relationship (SCR) treats the match between system coverage capability and task search space as a foundational constraint on delivery reliability. This is an analytical relationship, not a formula comparing two directly measurable numbers. [5]

An observability problem remains: producing a result or passing known checks does not reveal whether key relationships have been missed. MRAC generates process evidence through successive execution, audit, and repair, allowing coverage to be investigated indirectly. [4]

2.2 How audit and repair reveal coverage

MRAC observes evolving engineering artifacts under a stable objective. These artifacts include requirement and design specifications (Specs), as well as execution artifacts such as code changes. Objectives, non-goals, important constraints, and issue-assessment rules remain stable while the artifact evolves through repairs. Making a compatibility requirement implicit in the original objective explicit is a revision; adding an unrelated feature changes the task.

The basic cycle is to execute and produce an artifact, independently audit it for omissions and conflicts, repair it, and audit again. An independent audit starts afresh from the task objective, the current artifact, and the permitted evidence. Roles may be separated through different calls or sessions, or by changing models or audit perspectives. This operational separation does not imply statistical independence.

A repair can remove an issue, expose a previously obscured dependency, or introduce a new conflict. While resolving the current issue, the system must preserve constraints that already held. When a task spans multiple calls, each call must also recover the engineering state and earlier decisions. More calls therefore do not automatically produce more complete coverage. This is the mechanism through which audit-and-repair trajectories may reveal the stability of coverage.

The Unity module-migration task in this paper illustrates the point. Independent module compilation, existing caller relationships, data-format compatibility, and save-failure behavior must all hold together. A local change may fix one issue while violating another requirement. Stable coverage describes the ability to coordinate these relationships throughout revision. This example explains the mechanism; establishing that a particular causal chain occurred in a run still requires issue evidence and repair diffs.

2.3 Conditional meanings of convergence and non-convergence

A single audit produces a list of findings. A multi-round trajectory shows whether valid issues continue to emerge, recur, or stop appearing after they are addressed. Counts need not decline monotonically: more findings can reflect either a worse artifact or a more effective audit.

When the task reference remains stable, issues are valid and traceable, repairs are actually carried out, and audit and budget conditions are explicit, the continuing appearance of important issues requiring action indicates that the current system has not yet demonstrated stable, effective coverage under those conditions. This is a conditional negative signal. It does not prove that the task is impossible or attribute the result solely to the base model. Continuing scope expansion, unexecuted repairs, and tool failures should be recorded separately rather than mixed into ordinary convergence comparisons.

The positive signal from convergence is more limited. Consecutive valid clean audits first establish that the current audit configuration no longer finds issues requiring action. This state supports a coverage interpretation only if remaining important defects still have a chance of being detected. Shared blind spots can cause repeated omissions, so simply adding more identical audits cannot replace audit-capability calibration. Diverse audits, independent tests, or real execution outcomes can strengthen the coverage interpretation without proving complete correctness. [4]

2.4 From a coverage signal to an evaluation signal

MRAC Bench turns this mechanism into comparable tasks and protocols. It fixes inputs, information access, issue rules, and budgets, then observes the process and cost by which different systems stabilize an engineering artifact. Operational convergence requires all valid issues requiring action to be closed and the same artifact version to pass a predetermined number of consecutive valid clean audits. Section 3 specifies the rules.

The evaluation hypothesis is that, within a range where audits can identify the relevant important issues and assessment standards are comparable, stronger stable-coverage capability should make convergence within budget more likely or require fewer resources to reach an equivalent stable state. Section 10 discusses calibration and counterexamples for these two premises. If this correspondence is repeatable and meaningful for engineering, convergence

trajectories may provide a useful evaluation signal before stronger claims about complete coverage or delivery reliability are established.

3. Evaluation Tasks and Protocols

3.1 Systems under evaluation and task instances

The system under evaluation includes the model, agent, tools, and context organization, operating under a specified budget. Holding other conditions fixed while changing the model supports model comparison; holding the model fixed while changing the workflow supports comparisons of agents or prompting approaches.

A case contains a repository baseline, an original objective, and required inputs. An initial artifact may be supplied as input or generated by the system under evaluation. Configurations start from the same task inputs; artifacts produced during initialization and execution are part of the evaluated process. Cases and protocols are managed separately, combined at runtime, and associated with a fixed protocol version.

The protocol defines the audit object and evidence scope. Specification audits examine omitted requirements and interface and lifecycle constraints. Audits of execution artifacts examine alignment with the objective, effects across modules, and available test evidence. Information access, permitted changes, and verification methods are fixed for each stage. Execution begins from a designated repository snapshot, with changes to code and other artifacts recorded. Repairs should be traceable to the original objective or existing evidence.

3.2 Run and stopping rules

A run consists of the following steps:

1. Fix task inputs, repository baseline, objective, information access, and budget. Perform initialization or execution as specified, and separately record the artifact to be audited.
2. Audit independently within the permitted information scope, recording findings, evidence, severity, and artifact version.
3. Determine which findings require action under the evidence and issue rules, repair them, and preserve the diffs. Reset the consecutive-clean count whenever the artifact changes.
4. Stop when the convergence threshold is reached, the budget is exhausted, or the run is paused, blocked, or terminated by an execution error. Record the status and reason separately.

A protocol may require, for example, two consecutive clean audits of the same version; the threshold is fixed before evaluation. Clean means that the protocol's evidence and adjudication rules identify no valid issues requiring action and no outstanding repairs. It cannot be inferred solely from counts at selected severity levels. Labels such as P1 and P2 may be used, but their meanings and treatment requirements must be defined. Stylistic preferences, duplicate reports, and out-of-scope suggestions do not increase the valid-issue count. Reasons must be retained for findings that are rejected or permitted to be deferred.

Initialization, audit, review, and repair are recorded separately. Resource accounting includes the actual consumption of initialization and failed calls. The protocol must also specify which calls count toward the round budget, so displayed audit-round numbers are not mistaken for the total budget.

3.3 Metrics and comparability

Metric	What is recorded
Convergence rate within budget	The proportion of repeated runs that reach the threshold on a fixed task set and protocol
Audit and repair counts	Separate counts for the two types of calls, distinguishing converged and non-converged runs
Resource consumption	Tokens, elapsed time, and available cost information
Issue trajectory	New and unresolved issues, regressions, and disposition
Stopping reason	Convergence, budget exhaustion, pause, blockage, or execution error
Audit calibration status	Detection and false-positive results on known issues, review of clean artifacts, and model-task combinations that fail calibration

Reports should retain every run and stopping reason, distinguish environmental failures from failures of the evaluated system, and state task aggregation weights and convergence-rate denominators. Failure to converge within budget does not imply failure given unlimited time. Pauses, blockages, and errors cannot simply be equated with ordinary budget exhaustion. Partial trajectories should be reported as the state at the last recorded round.

When the same model audits and repairs, results reflect the entire loop and may also reflect model-specific audit granularity and preferences. With a common audit configuration, the comparison focuses more closely on repair capability under given feedback. These settings should be reported separately. Comparability across models requires the larger-scale testing described in Section 10.2. Model versions, tools, prompts, reasoning settings, context rules, and budgets must be recorded. Results under revised protocols should also be distinguished from earlier versions.

4. Low-Cost Case Construction

4.1 Starting from repositories and requirements

Real repositories already contain interfaces, data structures, dependencies, and compatibility requirements. These constrain tasks and provide evidence for audits. Requirements may come from actual plans or reasonable extensions to an existing system. Objectives should be understandable and satisfiable, and disagreements should be resolvable by reference to requirements and repository facts.

Construction proceeds by fixing a repository and objective, preparing task inputs and any necessary initial artifacts, running a common audit-and-repair protocol, and selecting tasks with the required discriminative value. A repository can yield multiple candidates across modules and directions of change, without waiting for historical issues to be paired with merged patches.

A candidate does not need a complete reference implementation or a dedicated acceptance verifier before entering evaluation. The repository supplies context, the common protocol organizes audits, runs support calibration, and targeted review checks the meaning of the signal. Existing tests and documentation remain useful. The proposed reduction in construction cost comes from reusing these preparation and evaluation processes across tasks.

4.2 An illustrative example

Consider a batch-processing tool with an existing task queue, plugin interface, and result output. A new requirement adds recovery after interruption while preserving how existing plugins are used. This example illustrates the construction method; it is not a measured case.

A design audit may reveal omissions concerning task identity, recognition of completed tasks, or the interaction between cancellation and recovery. An audit of the implementation can check whether recovery correctly skips completed tasks and whether state persistence is consistent with plugin invocation. Revising task-identity rules may in turn affect result indexing, creating another issue for a subsequent round.

The same requirement can support either a specification audit or an audit of execution artifacts under the chosen protocol. Each fixes its own starting point, evidence scope, and stopping conditions while sharing task sources and run infrastructure.

4.3 Construction and calibration costs

The paper proposes extremely low case-specific preparation costs; the magnitude of this advantage requires measurement. Costs include at least repository integration, candidate generation, calibration runs, quality review, and release maintenance. Environments, protocols, and tools can be reused, while computation and difficult-case handling incur costs per task.

Evaluation should report the candidate acceptance rate and the human effort, generation, screening, and review costs allocated to each usable case. Large-scale trials across many models may still be expensive. A high total run budget can coexist with low case-specific preparation costs. Reporting only the cost of generating a task description or initial artifact cannot establish an overall advantage.

5. Difficulty Adjustment and Continuing Extension

5.1 Varying engineering constraints and the initial state

Difficulty adjustment targets the task's search space: how many key conditions must be coordinated, how they are coupled, which dependencies remain implicit, and how broadly change effects must be verified. Large repositories provide rich sources of constraints, but code volume does not directly determine difficulty. A short task can span several modules, while a long artifact can address only simple constraints. Difficulty therefore requires empirical calibration.

Task constructors can vary change scope, compatibility requirements, dependencies across modules, and recovery, concurrency, or migration conditions. They can also split or combine requirements. The example in Section 4.2 might initially permit only one plugin and one recovery entry point, then introduce multiple execution paths and cancellation-state handling. Each step keeps the objective satisfiable without assuming that adding conditions necessarily increases difficulty.

Initial-artifact quality is another adjustable factor. A weaker generation configuration may leave more omissions, but obvious local errors can be easy to fix, while excessive disorder can make a task ill-defined. Initial artifacts supplied as case inputs must remain fixed across comparisons. Artifacts generated by the evaluated system are part of the evaluated process, with the generation configuration and output retained.

5.2 Calibrating difficulty through runs

Candidates are first tried on reference configurations, with checks of their ability to detect the relevant important issues. If all compared configurations pass audit calibration and converge quickly, meaningful constraints can be added. If all remain non-convergent, requirements can be simplified or split. The aim is to find a region where some systems converge while others continue to expose issues. Tasks on which every system converges may also remain useful if cost differences are stable.

Independent repeated runs should verify the basis for admission to the task set. Screening runs must be separated from reported evaluation runs. The models used in screening should be recorded, and other configurations should test whether discriminative value transfers. Different task types may yield different rankings; a single capability ordering need not be imposed.

5.3 The claim of continuing extension and its limits

The extensibility claim is that, when repositories offer sufficiently rich combinations of satisfiable and effectively auditable constraints, requirement construction and calibration can continue to produce usable cases of greater difficulty. As systems improve and task sources expand, stable discrimination should remain obtainable with affordable screening effort, rather than merely by generating more complicated text.

Here, "arbitrary difficulty" means constructing tasks within a desired difficulty range for a given system and budget. Repository size alone does not guarantee that every difficulty level is reachable. A finite repository may exhaust its task combinations, in which case construction can extend to other repositories, versions, and engineering structures.

The claim that the method "need not become obsolete as models improve" therefore concerns the task-generation and renewal mechanism. Existing cases may saturate while the infrastructure continues to supply new tasks near capability boundaries. If audits consistently fail to recognize key relationships as complexity rises, difficulty may cease to translate into a useful signal. Section 10.3 develops a counterexample to this claim.

6. Evaluation Infrastructure

The infrastructure comprises task management, a common runner, process recording, difficulty calibration, and versioned releases. Task production covers repository selection, requirement design, input packaging, trial runs, and admission to the task set. The runner isolates starting states and executes protocols. Records retain issue evidence, repair diffs, artifact versions, and resource consumption, allowing results to be traced to the underlying process.

The author's existing engineering workflow implements audits of both Specs and execution artifacts. The public [MRAC-Bench](#) project, in the version examined for this paper, provides two Spec-audit protocols, task packaging, fixed inputs and repository baselines, independent model calls, and process-state records. [6] Audits of execution artifacts are already part of the engineering practice. Reusing them as public benchmark protocols requires their evidence scopes and convergence conditions to be specified separately.

The spec-flow-simple-v1 protocol requires two consecutive adjudicated clean audits of the same Spec version. Accepted P3 findings also require repair, so zero P0-P2 findings do not automatically imply clean. Initialization and failed audits that have already started count toward the total audit budget; review and repair calls are counted separately. The protocol also distinguishes budget exhaustion, pauses, and blockages. The general method permits other parameters, but any comparison must be tied to a specific protocol version.

Fixed versions support comparisons across systems, while continuing task additions support exploration of capability boundaries. Releases may retain common tasks, but aggregate scores on different task sets should not be compared directly. Public repositories and task materials may still enter training data. Evaluations that require separation from training should therefore record information visibility and use private or delayed-release tasks.

A public benchmark needs coverage across engineering domains. Internal evaluations can use an organization's own repositories to compare model upgrades, agent changes, and cost configurations. Both can reuse task formats, runners, and analysis methods. Community contributors can supply candidates for calibration through a common process. Different agents can connect through adapters with their operating conditions made explicit.

MRAC Bench's long-term value thus lies both in its task collection and in its ability to produce tasks. Initial cases test the minimal mechanism; construction and execution at scale reveal the actual investment required as the system grows.

7. Pilot Study and Scope of the Evidence

7.1 Task background and experimental procedure

The task comes from a Unity game project with approximately 500,000 lines of code. It moves object-library disk I/O, pre-save checks, and path management from runtime modules into an independent editor module that can be compiled and verified without the top-level runtime assembly. The migration must preserve data formats, file paths, and asset identifiers while coordinating caller relationships, write ownership, and failure behavior during loading and saving. It therefore requires multiple existing engineering constraints to be maintained together.

To control the cost of reproduction at scale, the pilot uses Spec audits, aiming to identify repeatable differences among three model tiers with as little budget as possible. This choice reduces additional costs from code implementation, environment preparation, and repeated test execution. The original three runs for each model tier start from the same input Spec, perform one initialization round, and then alternate audit and repair. Within each run, the same evaluated model performs these steps. The common starting point precedes initialization, so initialized Specs may differ across models. The additional Sol-4 run restores the complete, previously verified revision from an earlier run after repeating initialization, then begins freeze audits. It is reported as a supplementary trajectory that reuses earlier revisions.

Post-initialization audits may read only the current Spec, not repository contents. They examine the engineering constraints, behavioral contracts, internal consistency, and verifiability expressed in the document. Consequently, this experiment cannot independently verify complete agreement between the Spec and repository facts. The available summary records do not specify the detailed information permissions during initialization and repair.

The results below organize the author's records or summaries for three Luna runs, four Sol runs, and three Astra runs. P1, P2, and P3 retain the severity labels in the original records. Clean retains the recorded run status rather than being inferred from P1 and P2 counts. Sol-1 and Sol-4 are both recorded as Spec-frozen after two consecutive clean audits. A single clean round and final convergence are shown separately, and issue counts are not treated as capability scores.

7.2 Overview of results

Table 1. Pilot results for three model tiers. Identifiers distinguish runs, not different tasks.

Model label	Runs	Reported trajectory	Observation
Luna	3	Each record reaches R6; every round reports P1 or P2 issues	All three remain non-convergent at the last recorded round
Sol	4	Sol-1 is clean at R6 and R7; Sol-4 is clean at R5 and R6; Sol-2 and Sol-3 still report issues in their final rounds	Two records end in freezing and two remain non-convergent; Sol-4 reuses earlier revisions
Astra	3	After initialization, at most the first audit reports an issue: P2×1, P1×1, and clean in the three runs	All three summaries show rapid stabilization

7.3 Round-by-round issue trajectories

Table 2. Post-initialization audit records for Luna and Sol. A cell of the form a/b gives the P1 and P2 counts, respectively; - indicates no subsequent record.

Audit round	Luna-1	Luna-2	Luna-3	Sol-1	Sol-2	Sol-3	Sol-4
R1	0/2	0/1	1/0	2/2	2/0	2/1	0/2†
R2	0/2	1/0	0/2	3/1	2/3	1/4	0/1
R3	0/1	0/2	0/1	2/2	2/3	1/1	0/2
R4	0/2	0/2	0/2	2/1	1/3	1/1	0/1
R5	0/2	1/0	0/1	1/2	clean	0/3	clean
R6	1/0	0/1	0/1	clean	1/2	0/4	clean
R7	-	-	-	clean	0/2	-	-

Every run first completes an Init round. In Tables 1 and 2, R1 consistently denotes the first audit after initialization. Initialization is excluded from the trajectory table, and the displayed round number is not the total audit budget including initialization. Sol-4's original R2-R7 correspond to R1-R6 here. The dagger (†) marks two additional P3 findings in that round, both accepted and repaired. Counts are reported issue entries for the round, without deduplication across rounds; - does not mean clean.

The three Astra summaries are labeled Astra-1, Astra-2, and Astra-3. Their first post-initialization audits report P2×1, P1×1, and no issues, respectively; the summaries report no further issues afterward. Because round-by-round clean records are unavailable, no specific stopping rounds are supplied, and these runs are not combined with Table 2 to calculate convergence rates under a common budget.

After its first clean round, Sol-2 still identifies issues concerning the standalone behavioral contract for alias synchronization, handling of unreadable optional data files, caller inventories, and the version state of a missing semantic shard. Later audits are thus still making engineering constraints explicit.

Accepted findings before Sol-4 freezes concern exception boundaries, optional-data deserialization, asset identifiers, and acceptance checks for retries. Its final two clean audits refer to the same Spec; the original records give exactly the same document digest:

4c23bf5407b0fadda949aa76280e03e7b6b13f709d5e6651616cba3b9ff91700.

7.4 Information in convergence trajectories and trends

The terminal status answers whether a run reached the threshold within the observation window. The trajectory also shows how the system approaches or moves away from that state. It distinguishes persistent high-severity findings, declining severity, a temporary clean followed by further findings, and consecutive clean audits. Two runs that both remain non-convergent may therefore be at different stages of the process.

The later rounds of Sol-2 and Sol-3 illustrate this distinction. Sol-2 first reaches clean at R5, reports P1 and P2 again at R6, and reports only P2 at R7. Sol-3 reports P1 in every round from R1 through R4, but only P2 at R5 and R6. These are local signs of stabilization near the truncation point: one run has reached clean once, while the other temporarily stops reporting higher-severity issues. However, Sol-2 backtracks, and Sol-3's P2 count rises from three to four. Neither has achieved sustained clean, and the records do not show that the total number of remaining issues is steadily declining.

Sol-1 and Sol-4 show a transition from continued findings to consecutive clean audits. The Astra summaries show earlier signs of convergence: after initialization, at most the first audit reports a single issue, with no further issues reported afterward. This contrasts with Sol's stabilization after multiple repair rounds, although exact convergence rounds still require complete clean records.

MRAC Bench should therefore retain the first clean round, the reappearance of issues after clean, changes at different severity levels, and the final consecutive-clean sequence. These observations can inform task-difficulty selection, further budget allocation, and tests of system improvements. They cannot, on their own, turn a non-converged run into a pass, nor can severity labels be used to estimate the fraction of search space already covered.

7.5 Scope of the evidence and further tests

The available records show process differences among three model tiers in the current setting, providing pilot evidence of MRAC Bench's discriminative potential. They do not yet establish direct comparability across models. Larger-scale repeated and cross-audit tests across repositories, tasks, audit objects, and model combinations are needed to examine the effects of audit granularity and preferences, ranking stability, and run costs.

Record lengths differ, Astra is represented only by summaries, and Sol-4 reuses an earlier revision. This section therefore reports trajectories and observed stopping states rather than a convergence rate under uniform experimental conditions. The materials do not include audit-capability calibration, cross-review of clean artifacts, or complete item-level adjudication and repair records. They cannot rule out apparent convergence caused by missed defects or establish actual search-space coverage from issue counts.

Model names are run labels. Complete model versions, prompts, reasoning settings, stage-specific permissions, and resource usage have not been supplied with these records. This section is therefore a pilot summary of the author's run records, insufficient for independent reproduction of a model ranking or quantification of a budget advantage. Further work should release the necessary configurations and original artifacts and review representative findings and final results.

Table 3. Further tests of the three MRAC Bench claims.

Claim	Evidence required
Discrimination and repeatability	Large-scale repeated runs and audit-configuration controls across repositories, tasks, and model combinations; retesting configurations not used in task selection, checking comparability across models, and reviewing representative findings and repairs
Low construction cost	Investment records from requirement formulation to admission, including the candidate acceptance rate and cost allocated to each usable case
Extension to greater difficulty	Task sequences with adjusted difficulty within a repository, including cases where a stronger configuration fails to converge and converges after the task is simplified

Statistical analysis should use complete runs and tasks as units and retain every stopping state. Run budgets and case-construction costs are reported separately. Trajectory differences test discrimination, while construction-cost claims require records of the actual effort spent generating, screening, and reviewing candidates.

8. Related Work

Evaluation in real repositories and automatic task construction. SWE-bench evaluates repository modifications for real GitHub issues. FeatureBench extends evaluation to feature development and constructs tasks automatically using tests and dependency relationships. [1] [2] SWE-smith automatically synthesizes problems that break existing tests in codebases to generate training data at scale. [7] Real engineering contexts and automatic task construction therefore have an established research basis. MRAC Bench specifically explores whether a common audit-and-repair protocol can reduce the upfront requirement for complete task-specific acceptance criteria.

Continual renewal. LiveCodeBench continually adds tasks from new programming-contest problems. [3] This paper further investigates actively constructing changes within repositories and calibrating difficulty by varying constraints.

Iterative feedback and auditing. Self-Refine uses model feedback to revise outputs iteratively. [8] Calboreanu reports non-monotonic issue trajectories in a single-system case study and two partial replications on a public synthetic specification, examining cross-model audits and inter-rater agreement. [9] These studies support investigating multi-round feedback, but do not establish a general model ranking.

Coverage relationships and process signals. SCR examines the relationship among task search space, system coverage capability, and delivery reliability. MRAC then uses multi-round issue trajectories to produce a conditional signal of coverage state. [5] [4] This paper applies that signal to evaluation, asking whether it can consistently distinguish systems within a calibrated range.

The paper combines process signals, low-cost task construction, and continuing difficulty calibration to investigate whether they can jointly support evaluation infrastructure. Existing execution-based verification can still provide independent outcomes against which to test the engineering meaning of process signals.

9. Problem Spaces Affected and Potential Industry Changes

If low-cost construction, meaningful discrimination, and extension to greater difficulty can be sustained together, MRAC Bench's impact could extend beyond a task collection. The following implications identify the conditions the mechanism changes, the questions this makes investigable, and the industry practices it could affect.

9.1 The supply of complex engineering evaluation tasks

How can systems be compared on a real requirement that lacks an inexpensive, complete, dedicated verifier? MRAC Bench allows an objective and initial artifact to be fixed first, with evaluation evidence accumulated through audit and repair. Design changes, compatibility migrations, and features spanning modules could consequently enter evaluation before an implementation has been completed.

This change could broaden participation in task production. Software teams could turn difficulties encountered in development into candidates and continually propose tasks in different directions around existing repositories. Evaluation supply could then grow more directly with development activity, at a scale determined by the cost of obtaining usable cases and the candidate acceptance rate.

9.2 Studying the engineering capabilities of models and agents

Can a system preserve complex constraints through successive changes, and what causes that process to become unstable? Reproducible tasks and process records make these questions experimentally accessible. Researchers can hold the model fixed while varying context, retrieval, and auditing, or hold the agent fixed and compare trajectories after a model change.

Two configurations that both fail to converge may nevertheless behave differently: old issues may persist, repairs may frequently introduce regressions, or previously unrecognized relationships may continue to emerge. Trajectories can guide the next experiment. Adjusting difficulty across related tasks can also reveal which capability boundaries an improvement expands.

This could provide research and development with more frequent feedback grounded in engineering artifacts. Workflow improvements could first be compared on existing tasks, then tested for transfer on new ones, reducing reliance on performance on a fixed task collection alone.

9.3 Feedback for training and system optimization

MRAC records connect initial artifacts, issue evidence, repair diffs, and subsequent effects. After reliable screening, these records could reveal repeatedly missed constraints, repairs prone to regressions, and approaches that preserve the objective. They could inform training-data selection, post-training, and agent design.

Lower barriers to task production would let teams construct requirement combinations more frequently, identify capability gaps, and retest improvements on new tasks. Task supply, system optimization, and difficulty renewal could thus form a continuing cycle.

Using convergence directly as a training reward requires separate calibration. Optimizing the label might improve repair capability, but it might instead merely reduce issue reporting. Training applications should retain independent tasks and outcome checks and separate training data from final evaluation data.

9.4 The evaluation ecosystem and industry collaboration

Protocols, run infrastructure, and task sources can be maintained separately. Software teams can contribute domain tasks, vendors can provide adapters and computation, researchers can improve signals and calibration methods, and public maintainers can manage versions and comparability. Participants can share formats and tools while retaining their own task sets.

Organizations can run internal evaluations on their repositories, while public benchmarks provide a reference across tasks. As task supply grows, the infrastructure's lasting value may increasingly depend on protocol credibility, calibration capability, reproducibility records, and continuing maintenance. Publishing admission rules and change histories can help sustain trust in collaboration.

Existing workflows for auditing specifications and execution artifacts can provide a starting point. Their value as infrastructure need not wait for every mechanism required by autonomous, project-level development to be completed.

9.5 Long-term autonomous engineering and engineering AGI

Specification audits examine design constraints, while audits of execution artifacts examine implementation, tests, and change effects. Task decomposition and roadmaps additionally involve overall objectives, dependency order, integration risks, and plan revision. Audit-and-repair trajectories can be explored for each, provided issues and stable reference conditions are defined separately.

Here, engineering AGI means AI autonomously completing complex engineering projects that would otherwise require a typical human team to collaborate over an extended period. MRAC Bench could provide an experimental foundation that grows from specifications and implementations to decomposition, integration, and long-term planning. It does not assume that every task admits a complete decomposition, nor that local convergence automatically means project completion.

Its potential contribution is to reveal engineering capability boundaries continually and support comparisons of more complex approaches to autonomous progress. Further research can reuse task production, execution, and recording systems while concentrating on the new mechanisms required at each stage.

10. Limitations and Counterexamples to the Core Claims

Individual cases with little discriminative value, audit drift, or shared blind spots can be addressed through screening and protocol maintenance. A counterexample to a core claim must show systematic failure within the declared scope that ordinary adjustments do not resolve. The following discussion examines signal meaning and comparability, continuing extension, and cost advantages. A changed protocol requires new evidence; it cannot rewrite a failure recorded under the original conditions.

10.1 Applicable capability range and the direction of the signal

MRAC Bench can compare convergence only within a capability range where the audit, under the given task and protocol, can effectively identify relevant important issues. This observability condition is necessary for interpreting convergence as evidence of coverage. In particular, when the same model audits and repairs, a weak model may repeatedly report clean because it cannot detect key problems, producing apparent convergence. Convergence speed therefore need not correspond monotonically to model capability across arbitrary capability ranges. Increasing the number of consecutive clean rounds alone cannot remove this error.

Two calibration checks can help identify such cases:

1. **Known-defect controls.** Reuse a small set of engineering artifacts with confirmed defects that match the task type, together with their corrected versions. Audit them without revealing defect locations and under the same information permissions. Record detection of the target defects and false positives according to rules fixed in advance.
2. **Cross-review of clean artifacts.** Freeze an artifact when it is reported clean and have other models or a common audit configuration review it under the same information permissions. Confirm new findings using common evidence and severity rules; a stronger reviewing model does not by itself establish that the original result was wrong.

Model-task combinations that fail calibration should be listed separately as having insufficient audit capability. Runs whose clean status is overturned by review should retain their original trajectories with missed-defect annotations and should not count as valid convergence. Results after simplifying the task or changing the audit configuration are reported separately. These checks can reuse historical issues, with their cost included in calibration. Passing them still does not guarantee that all blind spots have been removed.

A counterexample to the direction of the signal is as follows: system A misses a key constraint and fails to detect the omission, so it converges quickly; system B repairs more genuine issues but does not converge within budget. If a common review establishes that B's final artifact better meets the requirements, the convergence ranking points in the wrong direction. If such reversals persist across tasks within a capability range calibrated in advance, leaving no transferable basis for comparison, the capability-comparison claim is weakened. The applicable range cannot be redrawn afterward merely to make results agree with an expected ranking.

10.2 Comparability limits from audit granularity and preferences

Even models that can identify important issues may differ in audit granularity, severity judgments, and preferred areas of attention. The same issue may be split into several reports or combined with others, and models may disagree about which details should prevent clean. Similar coverage states may therefore produce different observed trajectories, affecting issue counts, repair scope, and convergence rounds even under identical prompts and severity labels. Repeatable differences between models may partly reflect stable audit preferences.

Large-scale testing is needed to assess this effect reliably. Experiments can repeat runs across repositories, tasks, and model combinations, cross-audit the same artifacts, and change the auditor while holding the executing model fixed. Analysis should examine changes in reported findings, clean judgments, and final rankings, combining root-cause grouping with reviews of representative defects to distinguish reporting differences from actual engineering problems.

The current pilot sample is too small to determine whether these differences systematically affect rankings. Common audit configurations, shared rules, and cross-review are possible controls; whether they support stable comparison remains an empirical question.

10.3 A counterexample to difficulty scaling

Suppose that, as tasks become more complex, both executors and auditors lose their understanding of key relationships. All systems then converge quickly, while independent checks continue to find important omissions. Engineering difficulty has increased, but the useful signal has disappeared.

This can be tested by increasing constraints across modules in several task families, then simplifying scope, changing the direction of the task, and improving the information available to the general audit procedure. If discrimination can be restored, isolated tasks that are too difficult

do not refute the mechanism. If the effective range persistently disappears despite real capability differences, the construction and audit methods may have a limit. Continuing to find tasks on which stronger configurations fail to converge but do converge after simplification would support extensibility within the range tested.

10.4 A counterexample involving unavoidable construction costs

Suppose candidates are easy to generate, but establishing the meaning of each case still requires a complete, task-specific analysis of constraints, a reference solution, and dedicated verification rules, with no way to reuse this work. Low candidate-generation cost would then fail to translate into low cost per usable task.

Evaluation should record why such work is necessary and test whether common calibration procedures, repeated runs, and sampled review can replace it. If several task families still require an almost complete acceptance framework for every case, the cost advantage is weakened. High token consumption alone is not this counterexample; the key question is whether the case-specific work the method aims to reduce must be reintroduced. The effects of a low candidate acceptance rate should likewise be assessed through the actual cost structure.

11. Conclusion

MRAC Bench turns the multi-round audit-and-repair process used in real development into an evaluation method. It uses convergence trajectories as a conditional signal of whether systems can achieve stable, effective coverage and combines that signal with resource consumption to compare models and agents. Tasks are constructed from repositories and change requests, with difficulty and applicability calibrated through runs. This reduces the upfront requirement for case-specific reference implementations and dedicated acceptance verifiers, while continuing task construction addresses the saturation of existing cases.

The existing engineering workflow supports audits of specifications and execution artifacts; the public benchmark version provides Spec-audit protocols. To control reproduction cost, the paper presents a Spec-based pilot study showing trajectory differences among three model tiers. It has not established that those differences directly measure actual coverage or imply a general capability ranking. If comparability and repeatability across models, low construction costs, and extension to greater difficulty can be sustained together within a calibrated range, the method could support public benchmarks, internal evaluations, model and agent optimization, and research into long-term autonomous engineering.

References

- [1] JIMENEZ C E, YANG J, WETTIG A, et al. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? ICLR, 2024. arXiv:2310.06770.
- [2] ZHOU Q, ZHANG J, WANG H, et al. FeatureBench: Benchmarking Agentic Coding for Complex Feature Development. ICLR, 2026. arXiv:2602.10975.
- [3] JAIN N, HAN K, GU A, et al. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code. 2024. arXiv:2403.07974.
- [4] LIU M. Multi-Round Audit Convergence (MRAC): A Conditional State Signal for High-Reliability AI Delivery. 2026. Based on the author-supplied Chinese manuscript, V2.9.2. [Public paper entry](#). DOI: 10.5281/zenodo.21804751.
- [5] LIU M. From the SCR Relationship to Search Space Governance: A Foundational Analytical Framework for Delivery Reliability in AI Software Engineering. 2026. Based on the author-supplied Chinese manuscript, V2.7.1. [Public paper entry](#). DOI: 10.5281/zenodo.21902420.
- [6] LIU M. [MRAC-Bench project repository](#). 2026. Protocol version examined: commit c2b1718. Accessed September 16, 2026.
- [7] YANG J, LIERET K, JIMENEZ C E, et al. SWE-smith: Scaling Data for Software Engineering Agents. 2025. arXiv:2504.21798.
- [8] MADAAN A, TANDON N, GUPTA P, et al. Self-Refine: Iterative Refinement with Self-Feedback. 2023. arXiv:2303.17651.
- [9] CALBOREANU E. Iterative Audit Convergence in LLM-Managed Multi-Agent Systems: A Case Study in Prompt-Engineering Quality Assurance. *Software*, 2026, 5: 26. DOI: 10.3390/software5020026. Based on arXiv v2.