Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

SAGE: A Self-Adaptive Agentic Framework for Execution-Guided Code Generation and Self-Repair

Harshil Lodhiya

hlodhiya@slicedhealth.com

SlicedHealth, Inc

Research Article

Keywords: Large language models, Agentic code generation, Execution-guided self-repair, API version drift, Self-adaptive software systems, Correctness-aware oracle

Posted Date: August 11th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-10523195/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

SAGE: A Self-Adaptive Agentic Framework for Execution-Guided Code Generation and Self-Repair

Harshil Lodhiya

Affiliation: SlicedHealth, Inc., Woodstock, GA, United States

Corresponding author: Harshil Lodhiya, hlodhiya@slicedhealth.com

ORCID: 0009-0005-5798-9176

Abstract

Large Language Models (LLMs) generate individual functions well, but turning a natural-language goal into a working multi-file project — with resolved dependencies and an entry point that actually runs — remains unreliable: single-pass output frequently crashes on missing packages, deprecated APIs, and unobserved runtime errors. We present SAGE, an open-source multi-agent framework that treats project synthesis as a closed-loop, self-adaptive control problem, a MAPE-K control loop: a Planner decomposes the goal, a Coding Agent generates every file, a Validator executes the project inside an isolated virtual environment, and a Feedback Agent converts the resulting tracebacks into typed repairs, iterating until the project runs. We evaluate SAGE on *SAGE-Bench-v2*, a benchmark we designed to exercise the failure modes that actually break LLM-generated code — API version-drift, type and schema errors, and silent incompleteness — using an *independent* re-execution oracle and the unbiased Pass@k estimator, with significance assessed by a paired Wilcoxon signed-rank test and Vargha–Delaney effect sizes. On `gpt-4.1-mini`, execution-guided repair raises Pass@1 from 69.3% to 92.0% (+22.7 points; $p = 0.042$, medium effect), rescuing tasks that open-loop generation never solves — for example, code that emits a pandas frequency alias removed in a recent release. We further introduce and test a *correctness-aware oracle* that repairs toward planner-specified success criteria; it does not significantly beat the exit-code oracle (+2.7 points, negligible effect), because the extra repair churn regresses roughly as many tasks as it rescues. SAGE offers a practical framework for self-healing synthesis and evidence that *specifying* success helps more than aggressively *enforcing* it.

Keywords: Large language models; Agentic code generation; Execution-guided self-repair; API version drift; Self-adaptive software systems; Correctness-aware oracle

Introduction

Large Language Models (LLMs) trained on source code have rapidly become competent at function-level program synthesis, achieving strong results on benchmarks such as HumanEval and MBPP that ask a model to complete a single, self-contained function from a docstring (Chen et al. 2021; Austin et al. 2021; Nijkamp et al. 2023). This progress has driven the adoption of AI assistants throughout the software-development lifecycle. However, the unit of value in real software engineering is rarely an isolated function; it is a *working project*: multiple interacting files, a resolved set of third-party dependencies, an executable entry point, and an environment in which the whole thing actually runs to completion.

When an LLM is asked to produce such a project in a single forward pass, the output is frequently plausible but non-functional. The generated code may reference packages that are not declared, import symbols that do not exist, assume input files that are absent, or, often enough, call library APIs that have been *deprecated or removed* in current releases. A model whose training data over-represents older code will confidently emit `matplotlib.cm.get_cmap`, `pandas.DataFrame.applymap`, or a `'M'` month-frequency alias, all of which now raise at runtime. Because the model has no observation of the program's execution, it cannot know that any of this has happened. The result is an artifact that looks correct and parses, yet fails when actually run, requiring a human to read the traceback, diagnose the cause, and fix the code. This *open-loop* mode of generation places a hard ceiling on autonomy.

A natural response is to close the loop: execute the generated program, observe what actually happens, and feed that observation back into a subsequent repair step. This idea — using execution as ground truth — underlies a productive line of recent work on self-refinement, reflection, and self-debugging (Shinn et al. 2023; Madaan et al. 2023; Chen et al. 2023). Yet most of this work operates at the granularity of a single function or a competitive-programming solution graded against hidden unit tests. The distinct engineering problems of a *multi-file project* — declaring and installing the right dependencies, provisioning an isolated environment, generating sample data so the program can run at all, and driving the artifact to a zero-exit-code execution — are largely left unaddressed.

We present **SAGE**, an open-source framework[1] that treats end-to-end project synthesis as a *closed-loop control problem*. A natural-language goal is decomposed into a structured plan; a coding agent materialises every file in that plan; a validator provisions an isolated Python virtual environment, installs the declared dependencies, syntax-checks the sources, and executes the entry point under a timeout; and a feedback agent reads the resulting logs and tracebacks and emits a *structured, typed* repair — a root-cause analysis,

[1] SAGE, its benchmark *SAGE-Bench-v2*, and the evaluation harness are released under the MIT license at https://github.com/harshillodhiya/SAGE.

a set of per-file edits, and a delta to the dependency set. An orchestrator drives this generate → validate → diagnose → repair cycle until the project runs successfully or an iteration budget is exhausted. We deliberately frame the architecture in the vocabulary of *self-adaptive software systems* (Kephart and Chess 2003; Salehie and Tahvildari 2009): the validator is the system's monitor, the feedback agent its analyser and planner, and the coding agent its executor, all operating over a shared knowledge base (the plan and code) in the spirit of the MAPE-K reference model for autonomic computing (Kephart and Chess 2003).

A framework alone, however, does not tell us *when* execution-guided repair actually pays off. Prior work has cautioned that self-repair is not a silver bullet (Olausson et al. 2024), and a fair evaluation must (i) target the failure modes that genuinely break LLM-generated projects rather than textbook exercises a strong model solves in one shot, (ii) grade success by *independent* re-execution rather than the system's own optimistic signal, and (iii) test differences for statistical significance. We therefore build *SAGE-Bench-v2*, a benchmark deliberately composed of tasks that induce three real failure modes — API version-drift, type/schema errors in multi-stage data flow, and silent incompleteness (a program that exits cleanly but omits a required output) — and evaluate SAGE against an open-loop baseline under this rigour.

This paper makes the following contributions:

1. **A closed-loop, multi-agent framework for project-level synthesis and self-repair.** SAGE couples an LLM with real execution inside isolated virtual environments, resolving dependencies and driving generated multi-file projects to functional execution without human intervention. The framework is open-source, provider-agnostic (any OpenAI-compatible endpoint), and configuration-driven.
2. **A control-theoretic formulation and a structured feedback interface.** We cast execution-guided repair as a MAPE-K feedback loop and define a typed feedback contract that converts unstructured runtime tracebacks into machine-actionable repair operations (analysis, per-file `modify`/`create`/`delete` edits, and requirement changes), rather than free-form regeneration.
3. **A failure-mode-targeted benchmark and a reproducible harness.** We release *SAGE-Bench-v2*, a suite of self-contained, offline software-generation tasks curated to exercise the failure modes that actually break LLM-generated code, together with a harness that grades success by an *independent* re-execution check (including completeness checks that catch programs which run but omit required outputs), estimates unbiased Pass@k (Chen et al. 2021), and reports paired Wilcoxon significance with Vargha–Delaney effect sizes. Because the tasks are authored for this work, they are *free of training-data contamination* by construction.
4. **An empirical study of when repair pays off.** On `gpt-4.1-mini`, execution-guided repair significantly improves project-level Pass@1 (69.3% → 92.0%, +22.7 points, $p = 0.042$, medium effect), rescuing tasks that open-loop generation *never* solves. We further introduce a *correctness-aware oracle* that repairs toward planner-specified success criteria and find, on a benchmark built to give it its best chance,

that it does *not* significantly beat an exit-code oracle (+2.7 points, negligible): it rescues the silent-incompleteness cases it targets but its extra repair churn regresses others. The lesson — *specifying* success helps more than *enforcing* it — is a rigorous, mechanistic negative result about the limits of self-repair.

The remainder of the paper is organised as follows. Section 2 surveys related work. Section 3 describes the SAGE framework and formalises its feedback loop. Section 4 covers implementation details. Section 5 defines the evaluation methodology, the research questions, and the released benchmark and harness. Section 6 answers the research questions with the measured results. Section 7 discusses findings and lessons learnt, Section 8 states limitations and threats to validity, and Section 9 concludes.

Related Work

LLM-based code generation and its evaluation. Program synthesis with LLMs is commonly assessed on function-completion benchmarks — HumanEval (Chen et al. 2021) and MBPP (Austin et al. 2021) — using the Pass@k metric, which estimates the probability that at least one of *k* sampled generations passes the hidden tests. Models such as CodeGen (Nijkamp et al. 2023) and prompting strategies such as chain-of-thought (Wei et al. 2022) improved single-shot quality. Subsequent work showed that these benchmarks can *overstate* correctness because their test suites are shallow; EvalPlus (Liu et al. 2023) demonstrated that augmenting HumanEval with far more tests exposes failures missed by the original suite. Two implications shape our evaluation: we grade with an *independent execution-based acceptance check* rather than trusting the generator's own signal, and — because these public benchmarks are widely present in training data — we author a fresh task suite to eliminate contamination.

Execution feedback and self-repair. A prominent line of work closes the loop between generation and execution. Self-Refine (Madaan et al. 2023) iterates generation with self-produced natural-language critiques; Reflexion (Shinn et al. 2023) maintains an episodic memory of verbal self-reflections derived from environment feedback; and Self-Debug (Chen et al. 2023) teaches models to inspect execution results and explain their own code before repairing it. CodeRL (Le et al. 2022) frames the problem with reinforcement learning using unit-test signals, and AlphaCodium (Ridnik et al. 2024) uses an iterative test-based flow for competitive programming. Olausson et al. (Olausson et al. 2024) question whether self-repair is a "silver bullet," showing its benefit is conditional on the quality of the feedback and the model's diagnostic ability. SAGE differs from this body of work along two axes: it targets *multi-file projects with real dependency resolution* rather than single functions graded by pre-existing tests, and its feedback is a *typed, structured* repair contract rather than free-form text or whole-solution regeneration. Our correctness-aware-oracle result also sharpens the "not a silver bullet" thesis with a concrete mechanism: aggressive repair can regress as much as it rescues.

Agentic and multi-agent frameworks. ReAct (Yao et al. 2023) interleaves reasoning and tool-use actions, establishing the template for LLM agents. General frameworks such as AutoGen (Wu et al. 2023) orchestrate conversations among multiple agents, while CodeAct (Wang et al. 2024) argues for executable code as the action space. In the software-engineering domain specifically, ChatDev (Qian et al. 2024) and MetaGPT (Hong et al. 2024) simulate a software company with role-playing agents (product manager, engineer, tester) following a waterfall-style process. SAGE is narrower and more mechanistic by design: rather than emulating human organisational roles, it defines four functional stages tied directly to a control loop, and it measures success by *whether the artifact actually executes*, foregrounding dependency isolation and runtime repair that role-play pipelines typically underemphasise.

Repository-level and issue-resolution agents. SWE-bench (Jimenez et al. 2024) evaluates agents on resolving real GitHub issues within large existing repositories, and SWE-agent (Yang et al. 2024) introduces an agent–computer interface for that setting. This task is complementary to ours: SWE-bench agents *edit* a large existing codebase to satisfy tests, whereas SAGE *synthesises* a new project from a natural-language goal and drives it to first successful execution. The two settings share the principle that execution feedback is essential to autonomy.

Self-adaptive and autonomic software systems. The framing of software that observes and repairs itself predates LLMs. Kephart and Chess's vision of autonomic computing (Kephart and Chess 2003) introduced the MAPE-K loop (Monitor, Analyse, Plan, Execute over shared Knowledge), and the self-adaptive-systems literature (Salehie and Tahvildari 2009) formalises feedback-driven adaptation. We adopt this vocabulary explicitly: SAGE can be read as an autonomic loop in which the "managed element" is a program under construction and the adaptation goal is functional execution. To our knowledge, this control-theoretic framing is underused in LLM code-generation systems, and it clarifies both the design and its failure modes — including, as we show, exactly where a feedback controller becomes blind.

Agent tooling. SAGE is implemented on top of the `smolagents` library (Roucher et al. 2024), which provides model abstractions and a typed tool interface; we use it as the LLM-access and tool layer.

The SAGE Framework

Overview

SAGE transforms a natural-language goal into a validated, runnable Python project through a pipeline of four LLM-driven stages coordinated by a deterministic orchestrator. Figure 1 shows the data flow. The design principle is *execution as ground truth*: no artifact is considered complete until it has been run to a successful (zero-exit-code) execution inside an isolated environment.

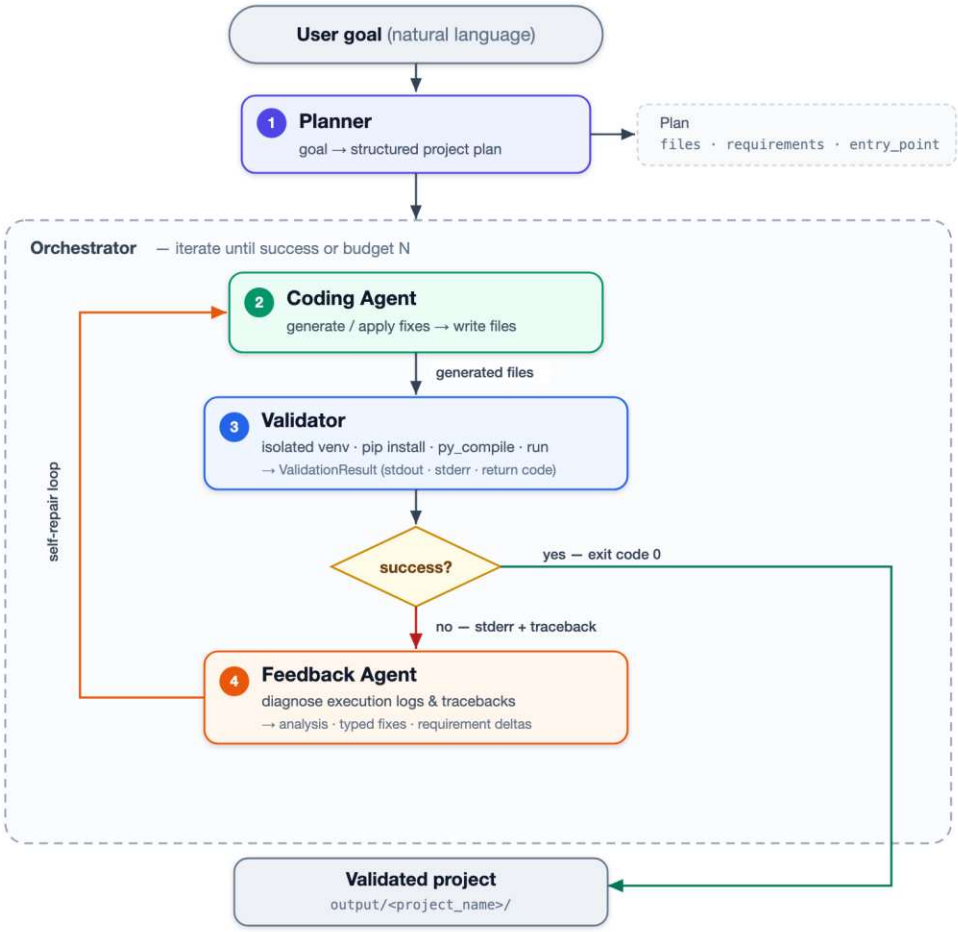


Figure 1. *SAGE's closed-loop architecture. A Planner decomposes the natural-language goal into a structured plan; the Orchestrator then drives the generate → validate → diagnose → repair cycle until the Validator reports a successful execution or the iteration budget N is exhausted. Because validation runs inside an isolated virtual environment, dependency and runtime errors become observable and are converted by the Feedback Agent into typed repairs.*

Mapping to a MAPE-K feedback loop

We formalise SAGE as a self-adaptive control loop over a program under construction. Let the *managed element* be the project state $P_t = (F_t, R_t)$ at iteration t, where F_t is the set of files (path → content) and R_t is the declared dependency set. A fixed *plan* π (produced once by the Planner) and the goal g form the shared **Knowledge**. Each iteration realises one turn of the MAPE-K loop:

- **Monitor** — the Validator executes P_t in an isolated environment and returns an observation $o_t = \langle$syntax_errors,runtime_errors,stdout,stderr,return_code$\rangle$.

- **Analyse + Plan** — the Feedback Agent maps the observation to a structured repair $\Delta_t = \langle \text{analysis,fixes,requirements_changes} \rangle$, where each fix is a typed operation over a single file.
- **Execute** — the Coding Agent applies Δ_t to obtain P_{t+1}.

The loop terminates successfully at the first t for which the return code is zero and no syntax or runtime errors are observed; i.e. P_t is a *fixed point* of the adaptation with respect to the goal. Algorithm 1 makes this precise.

```
Algorithm 1: SAGE orchestration loop
──────────────────────────────────────────────────────────────
Input : goal g, plan π, iteration budget N
Output: (success, project_dir, iterations, last_error)

P ← ∅                                    # current files
for t ← 1 .. N do
    if t = 1 then
        P ← CodingAgent.generate(π)      # first synthesis
    else
        P ← CodingAgent.fix(π, P, Δ)     # apply prior repair
    o ← Validator.validate(π, P)         # venv → pip → py_compile → run
    if o.success then
        return (true, dir, t, "")
    Δ ← FeedbackAgent.analyse(π, P, o)   # typed repair
    if Δ.requirements_changes ≠ ∅ then
        π.requirements ← apply(π.requirements, Δ.requirements_changes)
return (false, dir, N, summarise(o))
──────────────────────────────────────────────────────────────
```

Stage 1: Planner

The Planner issues a single constrained LLM call that decomposes the goal into a `Plan`: a project name, a natural-language description, a list of `FileSpec` entries (each with a relative path, a description, and intra-project dependencies), a list of PyPI `requirements`, an `entry_point`, optional `execution_args`, and a `success_criteria` block. The last is the observable evidence that the program actually accomplished the goal, expressed as expected output artifacts (glob patterns such as `*.pdf`) and expected stdout markers; it is the specification consumed by the *correctness-aware oracle* (Stage 3) that we evaluate in Section 6.2. The system prompt enforces several invariants that are essential for autonomous execution: the project must be runnable with *only* `python <entry_point>` and no additional command-line arguments; if the task would ordinarily require an input file (e.g. a spreadsheet), the entry point must *synthesise sample data automatically* so that it runs without user-supplied files; and the program must exit with code 0 on success. The Planner infers `success_criteria` from the natural-language goal alone; it never sees any external test suite or the benchmark's acceptance checks, so the mechanism remains general and self-contained rather than a leak of the evaluation. The response is parsed and

validated against a Pydantic schema, with bounded retries that feed the previous parse error back to the model.

Stage 2: Coding Agent

The Coding Agent materialises the plan. On the first iteration it emits the complete contents of every file as a JSON array of `{path, content}` objects, writes them to the project directory, and guarantees a `requirements.txt` consistent with the plan. On subsequent iterations it applies the Feedback Agent's repair. The agent supports two repair modes: (i) a *direct* mode, used when the feedback already contains complete replacement content for each affected file, which applies `modify`/`create`/`delete` operations deterministically to the file set; and (ii) a *regenerative* mode, used otherwise, in which the model is re-prompted with the plan, the current sources, the root-cause analysis, and the fix instructions, and asked to return the complete corrected file set. Both modes are bounded by a retry budget that returns parse errors to the model.

Stage 3: Validator

The Validator is the source of ground truth and executes up to four phases against the resolved, absolute project path:

1. **Syntax check.** Every `.py` file (excluding any virtual-environment directory) is compiled with `py_compile`. Any failure short-circuits the pipeline with a structured list of syntax errors, avoiding the cost of environment provisioning for code that cannot even parse.
2. **Dependency isolation and installation.** If `requirements.txt` is non-empty, the Validator creates a dedicated `.venv` for the project and installs the requirements with `pip` under a timeout. Isolation keeps the host environment clean and makes dependency errors (missing or unresolvable packages) observable rather than silently satisfied by ambient global packages.
3. **Execution.** The entry point is executed with the virtual environment's interpreter as a subprocess, with the project as the working directory, under a configurable wall-clock timeout (default 60 s). Standard output, standard error, and the return code are captured into a `ValidationResult`. A non-zero return code, a timeout, or a launch failure is recorded as a runtime error.
4. **Correctness-aware oracle (optional).** A zero exit code is necessary but not always sufficient. After a clean run, the Validator can additionally evaluate the plan's `success_criteria`: it checks that the run produced the expected artifacts (files matching the specified globs) and printed the expected stdout markers. A program that runs to completion yet fails these criteria — for instance, one that exits 0 without ever writing the report it was asked to generate — is then recorded as a failure and handed to the repair loop, exactly as a crash would be. This closes a gap in principle: without it, the loop optimises for *runs without error* rather than *accomplishes the task*. Whether closing that gap actually improves net success is an empirical question we answer in Section 6.2.

A configuration switch (`syntax_check_only`) allows the execution phase to be skipped for fast, static-only runs, and `correctness_aware` toggles the fourth phase, recovering the exit-code-only oracle used as the primary condition in Section 6.

Stage 4: Feedback Agent

When validation fails, the Feedback Agent receives the plan, the current sources, and the full `ValidationResult` (including tracebacks) and returns a `FeedbackResult` with three parts: a natural-language `analysis` of the root cause; a list of `fixes`, each a typed operation (`modify`, `create`, or `delete`) over one file, optionally carrying the complete new file content; and `requirements_changes`, a list of dependency additions or removals (the latter encoded as `remove:<package>`). The prompt requires that at least one concrete fix with full corrected content be returned — never an empty repair — and re-asserts the runnable-with-`python main.py` and auto-sample-data invariants so that repairs do not regress the project into a non-runnable state. If, after bounded retries, the response cannot be parsed into the schema, the agent degrades gracefully to an empty repair rather than crashing the loop. This *typed* interface is a deliberate design choice: it turns an unstructured traceback into a machine-checkable, directly-applicable set of edits and dependency deltas, and it lets the orchestrator update the plan's dependency set between iterations.

Orchestrator

The Orchestrator owns the loop of Algorithm 1. It prepares a per-project output directory derived from the plan name, then iterates up to `max_iterations` (default 5): generate on the first pass, repair thereafter, validate, and either return success or obtain and apply feedback. It also reconciles the plan's dependency set with any `requirements_changes` proposed by the Feedback Agent, so that the next installation phase reflects the diagnosis. On exhausting the budget it returns a failure result carrying a summary of the last observed error.

Implementation

SAGE is implemented in Python (3.11+) in roughly 1,000 lines across a small, modular package. LLM access goes through the `smolagents` `OpenAIServerModel` abstraction (Roucher et al. 2024), which makes the framework provider-agnostic: any OpenAI-compatible endpoint works, including hosted OpenAI models, the Hugging Face Inference API, and local servers such as Ollama or vLLM via a custom base URL. All behaviour is controlled by a single `config.ini` (LLM provider/model/temperature/token limits; planner, orchestrator, coding-agent, validator, and feedback budgets; venv and timeout policy; output location), so no code changes are needed to change models or loop parameters.

Data contracts are defined as Pydantic models (`Plan`, `FileSpec`, `ValidationResult`, `FixInstruction`, `FeedbackResult`, `OrchestratorResult`), which gives each inter-agent

message a validated schema and makes malformed LLM output a caught, retried error rather than a downstream crash. The framework's side-effecting capabilities — writing files, creating directories, executing commands, installing dependencies, and syntax-checking a project — are additionally packaged as typed tools conforming to the `smolagents` `Tool` interface, providing a reusable capability layer that keeps effectful operations behind a small, testable surface. A Rich-based command-line interface streams the plan, per-iteration progress, and the final outcome. The project is released under the MIT license and is openly available at https://github.com/harshillodhiya/SAGE.

Evaluation Methodology

Research questions

We structure the evaluation around three questions:

- **RQ1 (Does repair help?)** Does closing the loop with execution-guided repair increase the rate at which natural-language goals are turned into functionally-executing multi-file projects, relative to an otherwise identical open-loop generator?
- **RQ2 (Specify vs. enforce?)** Does a *correctness-aware oracle* — one that repairs toward planner-specified success criteria rather than accepting any program that merely exits cleanly — improve net success over a conventional exit-code oracle?
- **RQ3 (Does it depend on model capability?)** Does the benefit of repair persist for a substantially weaker model?

Benchmark suite

The primary evaluation uses *SAGE-Bench-v2*, a suite of 15 software-generation tasks that we authored specifically to exercise the failure modes that genuinely break LLM-generated projects — rather than textbook algorithmic exercises, which we found a strong model solves in a single pass and which therefore leave no room to observe repair. Every task is deliberately **self-contained and offline**: it runs with `python <entry_point>`, requires no network access and no user-supplied input files, and each prompt instructs the generator to synthesise its own sample data (matching the Planner's runnable-project contract). Because the tasks are authored for this work and not drawn from any public corpus, the suite is **free of training-data contamination** by construction — an increasingly important property as public benchmarks leak into training data (Liu et al. 2023). Table 1 groups the tasks by the failure mode they target.

SAGE-Bench-v2 tasks, grouped by the failure mode each is designed to induce.

Targeted failure mode	#	Representative tasks	What breaks in open-loop generation
API version-drift	8	seaborn correlation	models emit

Targeted failure mode	#	Representative tasks	What breaks in open-loop generation
		heatmap; pandas sales pivot; monthly time-series resample; NumPy convolution filters; scikit-learn model selection; SciPy signal spectrum; multi-sheet Excel summary; logistic regression from scratch	removed/deprecated calls: `pandas.np`, month freq `'M'→'ME'`, `matplotlib.cm.get_cmap`, `OneHotEncoder(sparse=)`, `scipy.fftpack`, `ExcelWriter.save()`
Silent incompleteness	4	multi-format dataset export; CRUD store round-trip; dataset profiling report; twelve-month KPI dashboard	program exits 0 but omits a required artifact or step (e.g. never writes the run log; skips the reload-and-verify stage)
Type/schema, multi-stage	3	ETL into a SQLite star schema; merge/reconcile on mismatched key dtypes; config-driven pipeline	foreign-key integrity, `int`-vs-`str` merge-key mismatch, multi-step orchestration gaps

Success criterion: independent acceptance

To avoid the well-documented risk of a system grading its own output (Liu et al. 2023), success for a generated project is decided by an **independent acceptance check** that is separate from the framework's internal validator. After a generation run completes, the harness re-executes the project's entry point in its virtual environment and marks the run successful only if (i) the process exits with the expected code (0); (ii) every *required* artifact is produced (the harness supports both "any-of" and, for completeness, "all-of" glob checks); and (iii) the standard output contains the required tokens (again with "any-of" and "all-of" forms). The "all-of" checks are what let the harness detect *silent incompleteness* — a program that exits cleanly while omitting a required output — independently of whatever the framework believed internally.

Metric and statistics

For each task we draw n independent generation runs and count the number c that pass acceptance. We report the unbiased Pass@k estimator of Chen et al. (Chen et al. 2021):

$$\text{pass@}k = \mathbb{E}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right], \qquad 1 \le k \le n,$$

computed with the numerically stable product form and macro-averaged across tasks. Because a single SAGE "sample" is a *full closed-loop run* (which may itself perform several internal repairs), Pass@k here measures the probability that at least one of k independent end-to-end attempts yields a functionally-executing project. Sampling is performed at temperature 0.7 to give Pass@k the sample diversity it assumes; the temperature is recorded with every run. To test whether differences between conditions are significant, we treat each task's per-condition success rate as a paired observation and apply a **paired Wilcoxon signed-rank test** at $p = 0.05$, reporting the **Vargha–Delaney $\hat{A}_{12}$ effect size** with its standard magnitude thresholds (negligible/small/medium/large). The significance tooling is released as `eval/stats.py`.

Conditions

Holding the task suite, temperature ($T = 0.7$), and sample count ($n = 5$) fixed, we compare three conditions on the primary model:

- **Baseline (open-loop, `max_iterations = 1`):** generate once, validate, stop. Isolates single-pass quality.
- **SAGE, exit-code oracle (`max_iterations = 5`, `correctness_aware = false`):** the full generate → validate → diagnose → repair loop, where success means the program runs to a zero exit code. This is the primary closed-loop condition for RQ1.
- **SAGE, correctness-aware oracle (`max_iterations = 5`, `correctness_aware = true`):** identical, but success additionally requires the plan's `success_criteria` to be met, so "runs but incomplete" becomes a repairable failure. Comparing this against the exit-code oracle answers RQ2.

Beyond Pass@k we report diagnostic measures the harness computes directly: the **solve rate** (fraction of tasks solved at least once across the n samples), the **repair yield** (successful runs that required at least one repair — successes the open-loop baseline cannot obtain), and a per-task breakdown that attributes each pass to first-shot generation or to repair.

Experimental setup and reproducibility

The released harness (`eval/harness.py`, `eval/metrics.py`, `eval/report.py`, `eval/stats.py`) adds no dependencies beyond the framework. Each run records full provenance — model id, temperature, iteration budget, samples per task, execution

timeout, oracle, git commit, and a UTC timestamp — into a `results.json`, and the reporting scripts emit paper-ready tables and the significance analysis. The primary grid is 3 conditions $\times$ 15 tasks $\times$ 5 samples $= 225$ end-to-end generations on `gpt-4.1-mini`; all runs completed without infrastructural failures.

Results

RQ1: Does execution-guided repair help?

Under the exit-code oracle, closing the loop raises Pass@1 on `gpt-4.1-mini` from **69.3%** (open-loop baseline) to **92.0%** — a **+22.7-point** improvement — and lifts the benchmark to a 100% solve rate (Table 2). The difference is statistically significant (paired Wilcoxon $p = 0.042$) with a **medium** effect size ($\hat{A}_{12} = 0.66$); 8 of the 15 tasks change (Table 3).

Table 2. *Pass@k on SAGE-Bench-v2 (`gpt-4.1-mini`, $n = 5$, $T = 0.7$). Both closed-loop conditions solve every task by pass@3.*

Pass@k (%)	Baseline (open-loop)	SAGE (exit-code)	SAGE (correctness-aware)
pass@1	69.3	**92.0**	94.7
pass@2	78.0	98.7	99.3
pass@3	81.3	100.0	100.0
pass@5	86.7	100.0	100.0
solve rate	86.7	100.0	100.0

Table 3. *Significance (paired Wilcoxon signed-rank; Vargha–Delaney $\hat{A}_{12}$). Repair vs. baseline is significant; correctness-aware vs. exit-code is not.*

Comparison	ΔPass@1	Wilcoxon p	$\hat{A}_{12}$	Effect
SAGE (exit-code) vs. baseline	+22.7	**0.042**	0.66	medium
SAGE (correctness-aware) vs. baseline	+25.3	**0.025**	0.69	medium
SAGE (correctness-aware) vs. SAGE (exit-code)	+2.7	0.500	0.54	negligible

The effect is not an artifact of easy tasks: the two hardest tasks are ones the open-loop baseline **never** solves (0/5), so any success under SAGE is unambiguously attributable to repair. Table 4 shows the per-task rescues. The flagship case is `timeseries_monthly_resample`, which open-loop generation fails 5/5 because the model emits the pandas month-frequency alias `'M'` (removed in favour of `'ME'` in a recent release); SAGE repairs this to 5/5, with all five successes reached through the repair loop. `etl_star_schema` (a `numpy.datetime64` misuse) goes 0/5 → 3/5, and `seaborn_correlation_heatmap` (the model calls `pandas.np`, removed in pandas 2.0) goes 1/5 → 5/5.

Table 4. *Representative per-task rescues under the exit-code oracle. The two 0/5 tasks are solved only by repair.*

Task	Open-loop failure	Baseline	SAGE	of which via repair
timeseries_monthly_resample	pandas freq `'M'`→`'ME'`	0/5	5/5	5
etl_star_schema	`numpy.datetime64` misuse	0/5	3/5	2
seaborn_correlation_heatmap	`pandas.np` removed	1/5	5/5	4
pandas_sales_pivot	dtype/schema error	1/5	4/5	1
excel_multisheet_summary	`ExcelWriter.save()` removed	3/5	5/5	1
multiformat_dataset_export	`int64` not JSON-serialisable	4/5	5/5	1

We observed one regression: `crud_store_roundtrip` fell from 5/5 (baseline) to 3/5 under SAGE, but with **zero** repairs recorded on the failing samples — i.e. the two failures are first-shot generations that happened to fail, sampling variance rather than harm from the loop. **Answering RQ1:** execution-guided repair significantly and substantially improves project-level Pass@1, and does so by fixing exactly the class of real runtime and dependency failures that open-loop generation cannot observe.

RQ2: Does specifying success beat enforcing it?

The correctness-aware oracle reaches Pass@1 of **94.7%**, a further +2.7 points over the exit-code oracle — but this difference is **not** statistically significant (Wilcoxon $p = 0.50$) and the effect size is **negligible** ($\hat{A}_{12} = 0.54$), even though *SAGE-Bench-v2* was deliberately built to include silent-incompleteness tasks that give the oracle its best chance. Table 5 shows why: the correctness-aware oracle rescues exactly the incompleteness cases it targets — `config_driven_pipeline` (which exits 0 but omits its run log) improves 4/5 → 5/5, `etl_star_schema` 3/5 → 5/5, `crud_store_roundtrip` 3/5 → 4/5 — but the additional repair

churn it triggers *regresses* other tasks — `multiformat_dataset_export` 5/5 → 4/5 and `pandas_sales_pivot` 4/5 → 3/5 — so the net change is a wash.

Table 5. *Where the correctness-aware oracle differs from the exit-code oracle. It fixes the incompleteness it can see but its extra repairs break other tasks, netting no significant gain.*

Task	SAGE (exit-code)	SAGE (correctness-aware)	Direction
config_driven_pipeline	4/5	5/5	rescued (missing-log incompleteness)
etl_star_schema	3/5	5/5	rescued
crud_store_roundtrip	3/5	4/5	rescued
multiformat_dataset_export	5/5	4/5	regressed (extra repair churn)
pandas_sales_pivot	4/5	3/5	regressed

This distinguishes two things that are easy to conflate. *Specifying* success — having the Planner articulate `success_criteria` that the Coding Agent then targets — is cheap and is already part of both closed-loop conditions. *Enforcing* success — having the Validator fail-and-repair whenever those criteria are unmet — adds repair actions, and on a strong model each extra repair carries a non-trivial risk of introducing a new defect. **Answering RQ2:** enforcing correctness criteria through additional repair does not significantly improve over an exit-code oracle; the value lies in *specifying* success, not in aggressively *enforcing* it.

RQ3: Does the benefit persist for a weaker model?

To probe capability-dependence without a full multi-tier re-run, we draw on measurements on the earlier *SAGE-Bench-v1* suite (12 tasks, same harness and protocol) with a substantially weaker model, `gpt-3.5-turbo`. There, execution-guided repair under the exit-code oracle raises Pass@1 from 60.0% to 75.0% — a **+15.0-point** lift that is significant (Wilcoxon $p = 0.042$, $\hat{A}_{12} = 0.59$, small effect). **Answering RQ3:** the repair benefit is not confined to a single model; it persists, and is if anything proportionally larger, on a weaker one. We caveat that this corroboration is on the v1 suite rather than v2 (the two suites are not directly comparable), and a full model × suite grid remains future work (Section 8).

Summary

Execution-guided repair delivers a significant, medium-effect improvement in project-level Pass@1 on a benchmark built to be hard for the right reasons, rescuing real dependency- and version-drift failures that open-loop generation cannot see. Making the loop's oracle correctness-aware fixes the silent-incompleteness cases it is designed for but does not

significantly raise net success, because extra repair on an already-strong model regresses about as much as it rescues.

Discussion and Lessons Learnt

Our design isolates the effect of each factor. Grounding success in an *independent re-execution* rather than the framework's own validator guards against the optimistic self-assessment that inflates results when a generator scores itself (Liu et al. 2023), and the *typed* feedback contract narrows the repair action space to per-file operations and explicit dependency deltas that the orchestrator applies deterministically. Against this backdrop we draw three lessons.

Execution feedback earns its keep on the failures that dominate real code — version and dependency drift. The rescues in Table 4 are not exotic bugs; they are the everyday consequence of a model trained on a moving ecosystem. `matplotlib.cm.get_cmap`, `pandas.DataFrame.applymap`, `pandas.np`, and the `'M'` frequency alias were all valid recently and are all now errors. An open-loop generator has no way to know the ecosystem moved beneath it; a single execution does. This suggests that the practical value of closed-loop synthesis will *grow* over time, as the gap between training-data APIs and deployed APIs widens.

A feedback controller can only correct what its sensor can measure — but measuring more is not free. The MAPE-K framing predicts that silent, exit-0 failures are invisible to an exit-code monitor, and the correctness-aware oracle is precisely the richer sensor that makes them visible; Table 5 confirms it rescues exactly those cases. Yet the same table delivers the more surprising lesson: on a strong model, acting on the richer signal triggers repairs whose collateral damage cancels the benefit. Self-repair being "not a silver bullet" (Olausson et al. 2024) is, here, quantitative and mechanistic: each repair is a stochastic edit that can regress a working file, so *more enforcement is not uniformly better*.

Specify, then repair conservatively. The combination that works is to have the planner state what success looks like (cheap, and it improves first-shot generation) while keeping the repair trigger conservative. This directly motivates **regression-guarded repair** — accepting a repaired state only when it does not worsen the best result observed so far — as the natural way to let a correctness-aware oracle realise its upside without its downside. We leave this to future work but note our data predicts it would convert the correctness-aware oracle's several rescues into a net gain by suppressing its regressions.

Limitations and Threats to Validity

Construct validity. Our acceptance check certifies *functional execution* (exit code, required artifacts, required stdout tokens), not full functional *correctness*. A project could pass acceptance while being subtly wrong. The "all-of" completeness checks tighten this beyond mere execution, but richer per-task oracles (property tests, reference outputs) are

future work; this is the same ceiling that motivated EvalPlus (Liu et al. 2023) at the function level.

Internal validity. Pass@k depends on sampling temperature and on n; low temperature reduces diversity and can bias Pass@k for $k > 1$, so we sample at $T = 0.7$ and report it. LLM APIs are non-deterministic and evolve, so we record model id, timestamp, and git commit for every run, but exact reproduction may require pinning a model snapshot. The single observed regression (`crud_store_roundtrip`) is attributable to sampling variance (zero repairs on the failing samples), which we report rather than suppress; at $n = 5$, per-task rates are necessarily coarse.

Conclusion validity. We test each comparison with a paired Wilcoxon signed-rank test and report Vargha–Delaney effect sizes rather than relying on point differences. With 15 tasks the tests have limited power, and both closed-loop conditions saturate the benchmark by pass@3; the significant RQ1 result and the null RQ2 result should therefore be read as holding *on this suite at this scale*. A larger suite would tighten the confidence intervals.

External validity. The primary grid uses a single strong model (`gpt-4.1-mini`); RQ3 corroborates on `gpt-3.5-turbo` but on the earlier v1 suite, so the two are not directly comparable. A full model × suite grid — additional tiers (e.g. a nano-class model and a non-OpenAI model such as DeepSeek-V3, which the provider-agnostic implementation already supports) crossed with v2 — is the most important extension and would substantiate the capability-dependence claim rigorously. Separately, *SAGE-Bench-v2* targets small, self-contained, offline Python projects; results may not transfer to large multi-module systems, network- or credential-dependent applications, or other languages, and repository-scale editing is a different problem better represented by SWE-bench-style settings (Jimenez et al. 2024; Yang et al. 2024).

Contamination. A recurring threat for LLM-code evaluation is that benchmark solutions are present in training data (Liu et al. 2023). Because *SAGE-Bench-v2* was authored for this work and is not drawn from any public corpus, the primary evaluation is contamination-free by construction — a deliberate advantage over reusing public suites.

Cost and safety. Each closed-loop run performs real dependency installation and code execution as subprocesses. A virtual environment is *not* a security sandbox; executing model-generated code and installing arbitrary packages should be done in a disposable, network-controlled container for untrusted goals. Timeouts bound runaway executions but not resource exhaustion during installation; Section 9 lists containerised execution as the top hardening priority.

Bias in task design. Because the benchmark author also designed the acceptance signals, there is a risk of tasks that implicitly favour the system under test. The tasks mirror the same runnable-project contract given to any generator, and the identical independent acceptance check is applied to every condition, but an externally-authored task suite would further strengthen the claim.

Conclusion and Future Work

We presented SAGE, an open-source, multi-agent framework that treats natural-language-to-project synthesis as a closed-loop, self-adaptive control problem. By executing generated multi-file projects inside isolated virtual environments and converting the resulting runtime observations into typed, machine-actionable repairs, SAGE drives projects toward functional execution without human intervention. On a benchmark built to exercise the failure modes that actually break LLM-generated code, execution-guided repair improves Pass@1 from 69.3% to 92.0% (+22.7 points; $p = 0.042$, medium effect), rescuing real dependency- and version-drift failures that a single pass cannot observe. Introducing a *correctness-aware oracle* that repairs toward specified success criteria fixes the silent-incompleteness cases it targets but does not significantly beat an exit-code oracle, because the extra repair churn regresses as much as it rescues — evidence that *specifying* success matters more than aggressively *enforcing* it.

Future work proceeds along lines directly motivated by these findings. **(1) Regression-guarded repair:** accept a repaired state only when it does not worsen the best result observed so far, which our data predicts would let the correctness-aware oracle realise its upside without its regressions. **(2) A full model × suite grid:** cross additional model tiers (including a non-OpenAI model, which the implementation already supports) with *SAGE-Bench-v2* to rigorously establish capability-dependence and tighten confidence intervals. **(3) Richer oracles:** augment success criteria with property-based and reference-output tests. **(4) Hardened execution:** replace virtual-environment isolation with containerised, network-controlled sandboxes so untrusted goals can be handled safely. **(5) Learning from the loop:** closed-loop traces (goal, plan, error, repair, outcome) are natural training data; fine-tuning or reinforcement learning on successful repair trajectories could extend SAGE's self-adaptation from within-run repair to across-run learning. We release the framework, *SAGE-Bench-v2*, and the evaluation harness to lower the barrier to rigorous, execution-grounded evaluation of self-healing software-engineering pipelines.

Statements and Declarations

Funding. The author received no specific grant for this research from any funding agency in the public, commercial, or not-for-profit sectors.

Competing interests. The author declares that they have no competing interests.

Ethics approval. Not applicable (this work involves no human participants or animal subjects).

Consent to participate / for publication. Not applicable.

Use of generative AI. During the preparation of this work the author used a large-language-model-based coding assistant to help implement the evaluation harness. The author reviewed, edited, and verified all content and all reported results, and takes full

responsibility for the content of the publication. No LLM is listed as an author, consistent with the publisher's authorship policy.

Data availability. The benchmark suite (*SAGE-Bench-v2*), all generated projects, and all result files are available in the project's open-source repository https://github.com/harshillodhiya/SAGE.

Code availability. SAGE and its evaluation harness are open-source under the MIT license, available in the project repository https://github.com/harshillodhiya/SAGE.

Author contributions. Harshil Lodhiya: conceptualization, methodology, software, investigation, formal analysis, and writing (original draft and editing).

References

Austin, Jacob, Augustus Odena, Maxwell Nye, et al. 2021. "Program Synthesis with Large Language Models." *arXiv Preprint arXiv:2108.07732*.

Chen, Mark, Jerry Tworek, Heewoo Jun, Qiming Yuan, et al. 2021. "Evaluating Large Language Models Trained on Code." *arXiv Preprint arXiv:2107.03374*.

Chen, Xinyun, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. "Teaching Large Language Models to Self-Debug." *arXiv Preprint arXiv:2304.05128*.

Hong, Sirui, Mingchen Zhuge, Jonathan Chen, et al. 2024. "MetaGPT: Meta Programming for a Multi-Agent Collaborative Framework." *International Conference on Learning Representations (ICLR)*.

Jimenez, Carlos E., John Yang, Alexander Wettig, et al. 2024. "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" *International Conference on Learning Representations (ICLR)*.

Kephart, Jeffrey O., and David M. Chess. 2003. "The Vision of Autonomic Computing." *Computer* 36 (1): 41–50.

Le, Hung, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven C. H. Hoi. 2022. "CodeRL: Mastering Code Generation Through Pretrained Models and Deep Reinforcement Learning." *Advances in Neural Information Processing Systems (NeurIPS)*.

Liu, Jiawei, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. "Is Your Code Generated by ChatGPT Really Correct? Rigorous Evaluation of Large Language Models for Code Generation." *Advances in Neural Information Processing Systems (NeurIPS)*.

Madaan, Aman, Niket Tandon, Prakhar Gupta, et al. 2023. "Self-Refine: Iterative Refinement with Self-Feedback." *Advances in Neural Information Processing Systems (NeurIPS)*.

Nijkamp, Erik, Bo Pang, Hiroaki Hayashi, et al. 2023. “CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis.” *International Conference on Learning Representations (ICLR)*.

Olausson, Theo X., Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2024. “Is Self-Repair a Silver Bullet for Code Generation?” *International Conference on Learning Representations (ICLR)*.

Qian, Chen, Wei Liu, Hongzhang Liu, et al. 2024. “ChatDev: Communicative Agents for Software Development.” *Proceedings of the Association for Computational Linguistics (ACL)*.

Ridnik, Tal, Dedy Kredo, and Itamar Friedman. 2024. “Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering.” *arXiv Preprint arXiv:2401.08500*.

Roucher, Aymeric, Albert Villanova del Moral, Thomas Wolf, et al. 2024. *smolagents: A Smol Library to Build Great Agents*. Https://github.com/huggingface/smolagents.

Salehie, Mazeiar, and Ladan Tahvildari. 2009. “Self-Adaptive Software: Landscape and Research Challenges.” *ACM Transactions on Autonomous and Adaptive Systems* 4 (2): 1–42.

Shinn, Noah, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. “Reflexion: Language Agents with Verbal Reinforcement Learning.” *Advances in Neural Information Processing Systems (NeurIPS)*.

Wang, Xingyao, Yangyi Chen, Lifan Yuan, et al. 2024. “Executable Code Actions Elicit Better LLM Agents.” *International Conference on Machine Learning (ICML)*.

Wei, Jason, Xuezhi Wang, Dale Schuurmans, et al. 2022. “Chain-of-Thought Prompting Elicits Reasoning in Large Language Models.” *Advances in Neural Information Processing Systems (NeurIPS)*.

Wu, Qingyun, Gagan Bansal, Jieyu Zhang, et al. 2023. “AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation.” *arXiv Preprint arXiv:2308.08155*.

Yang, John, Carlos E. Jimenez, Alexander Wettig, et al. 2024. “SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering.” *Advances in Neural Information Processing Systems (NeurIPS)*.

Yao, Shunyu, Jeffrey Zhao, Dian Yu, et al. 2023. “ReAct: Synergizing Reasoning and Acting in Language Models.” *International Conference on Learning Representations (ICLR)*.