

Execution Grounded Multiagent Systems for Reliable Backend Code Generation with Large Language Models'

Sai Deekshith Lekkala

Kakatiya University

Jothi Prabha Appadurai

Kakatiya University

Rohith Reddy Bellibatlu

Florida International University

Manpreet Singh

Boston University

Rahul Joshi

rahulj@sitpune.edu.in

Symbiosis International University

Method Article

Keywords: Code generation, Large language models, Multi-agent systems, Execution-grounded validation, Retrieval-augmented generation, Empirical mechanism isolation, Workflow orchestration

Posted Date: August 14th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-10681298/v1>

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

[Read Full License](#)

Additional Declarations: No competing interests reported.

Execution Grounded Multiagent Systems for Reliable Backend Code Generation with Large Language Models

Sai Deekshith Lekkala^{1†}, Jothi Prabha Appadurai¹,
Rohith Reddy Bellibatlu^{2†}, Manpreet Singh, ^{3†}, Rahul Joshi ^{4*}

¹Kakatiya Institute of Technology and Science, Warangal, Telangana, India.

²Florida International University, Miami, FL, 33199, USA.

³Boston University, Boston, MA, USA.

^{4*}Symbiosis Institute of Technology, Pune, Symbiosis International (Deemed University), Pune, India.

*Corresponding author(s). E-mail(s): rahulj@sitpune.edu.in;
Contributing authors: b23ai005@kitsw.ac.in; ajp.csm@kitsw.ac.in;
manni@bu.edu;

[†]These authors contributed equally to this work as co-first authors.

Abstract

Multi-agent architectures for large-language-model code synthesis are widely reported to improve correctness, but they are rarely compared against a single-agent baseline that receives the same execution feedback inside the same runtime. The gain attributed to role decomposition may therefore be the gain from execution feedback under another name. We present *ExecuGraph*, a workflow framework whose value here is as an instrument: one codebase collapses by configuration into single-agent one-shot (SO), single-agent execution-retry (SR), and full multi-agent (MF) conditions, so the two levers can be separated rather than confounded. On the complete 164-problem HumanEval benchmark, the only suite in our study with enough paired problems to be adequately powered, execution feedback is worth a large and unambiguous gain (SR over SO: **+25.6 pp**, 95% CI [**+18.9, +32.3**], exact Wilcoxon $p < 0.0001$ after Holm adjustment), whereas adding five further agents on top of it is worth nothing measurable (MF over SR: **−1.2 pp**, CI [**−7.3, +4.9**], $p = 0.76$) at **3.6×** the model calls. The second contrast is the more secure of the two, because both conditions receive execution feedback and are therefore equally affected by the fact that the retry

loop observes the scoring tests. A matched-compute control rules out the obvious alternative explanation: five independent one-shot samples filtered through the same sandbox, at five times the call budget, gain **+0.4 pp**, so the effect is feedback rather than extra sampling. We corroborate with a 30-problem algorithms suite, an APPS-introductory subset, and a second backbone, and we use three architecturally inert ablations as negative controls to measure run-to-run variance directly rather than assuming it. All figures are post-correction of a documented sandbox import-policy defect, re-scored from stored artifacts without re-invoking any model (Section 5.12). Configurations, per-trial logs, and table generators are released.

Keywords: Code generation, Large language models, Multi-agent systems, Execution-grounded validation, Retrieval-augmented generation, Empirical mechanism isolation, Workflow orchestration

1 Introduction

Large Language Models (LLMs) have rapidly become a default tool for automated code generation, achieving competitive performance on a broad range of programming tasks including backend code synthesis and algorithmic problem solving [1, 2, 3]. By learning statistical mappings between natural-language descriptions and source code, LLMs produce syntactically valid implementations from minimal human input, and they are increasingly relied on for rapid prototyping, coding assistance, and education [2, 4, 5].

Despite these strengths, reliability remains a significant limitation. Most production-grade LLM coding systems still adopt a *single-agent, single-pass* paradigm in which one model is responsible for problem understanding, solution generation, and implicit validation. While effective for short or routine code, this strategy frequently fails for backend code that depends on data-structure invariants, algorithmic efficiency, or precise edge-case handling. Generated programs often look correct yet contain semantic flaws, inefficient algorithm choices, or missing boundary handling that surface only at runtime [2, 3].

A common root cause is the absence of execution-level validation. Many approaches rely on textual self-review or chain-of-thought style reasoning to assess correctness [6, 7], but text-only checks are insufficient to detect runtime errors and performance issues that only manifest under execution [8, 9]. Iterative-refinement systems such as Reflexion [10] apply verbal self-critique with episode memory, and execution-filtered systems such as AlphaCode [11] validate sampled candidates against tests, but existing multi-agent pipelines frequently conflate role decomposition with execution feedback, making it impossible to measure each lever’s true marginal benefit.

1.1 Contributions

This paper presents *ExecuGraph*, a graph-structured multi-agent framework that explicitly isolates execution feedback loops from multi-agent role decomposition in backend code synthesis. Concretely, we contribute:

1. **A typed multi-agent transition graph that isolates synthesis mechanisms**, formally specified as a typed transition system over a shared state object with explicit decision predicates and bounded retries (Section 3). The framework collapses by configuration into single-agent one-shot, single-agent execution-retry (Reflexion-style), and per-agent ablation conditions inside a single codebase and evaluation harness, enabling controlled empirical measurement of each lever’s marginal contribution.
2. **An empirical diagnostic study of multi-agent overhead versus execution feedback**, separating execution feedback from role decomposition on the full 164-problem HumanEval suite, where the design is adequately powered. Execution feedback accounts for essentially the entire effect (+25.6 pp for single-retry over one-shot, exact $p < 0.0001$), while adding five more agents on top of it changes accuracy by -1.2 pp (95% CI $[-7.3, +4.9]$, $p = 0.76$) at roughly five times the LLM calls. A matched-compute sampling control (best-of-5 one-shot at five times the call budget, +0.4 pp) rules out additional sampling as the explanation. Section 6.7 identifies a separate confound inflating the feedback-versus-one-shot comparison; it does not affect the multi-full versus single-retry contrast, since both conditions receive feedback.
3. **Domain-specific trade-off analysis under cross-model evaluation**, showing that with DeepSeek-Coder-V2-Lite (16B Mixture-of-Experts (MoE) with 2.4B active parameters), multi-agent decomposition gains +20.0 percentage points on stateful graph problems (90.0% MF vs. 70.0% SO) while overall accuracy is unchanged (83.3% in both conditions), because DP and DS regress by 10.0 pp each. Since parameter count, routing architecture, and pre-training corpus all differ between the two backbones, we report this as a category-dependent trade-off and explicitly not as a scaling result.
4. **A per-agent ablation and cost-correctness trade-off analysis**, quantifying the marginal contribution of the Planner, Logical Reviewer, Optimizer, and retrieval layer, plus error taxonomies and deployment guidelines for production code synthesis.
5. **A fully reproducible artifact**: every numeric claim in the result tables is generated from per-trial JSON logs by the included `scripts/build_all_tables.py` script, with all configurations, model digests, and seeds committed to the public repository (<https://github.com/rohithreddybc/execuGraph>).

The remainder of the paper is organized as follows. Section 2 reviews related work on LLM code generation, iterative refinement, multi-agent coding pipelines, execution-based validation, and retrieval-augmented generation. Section 3 formalizes the proposed framework. Section 4 describes the evaluation protocol. Section 5 reports quantitative results. Section 6 discusses implications and limitations. Section 7 concludes. Section 8 provides full reproducibility details.

2 Related Work

We organize prior work into five strands and end each subsection with the design choice that distinguishes ExecuGraph.

2.1 LLMs for Code Generation

Foundational LLM work showed that scale and large code corpora yield strong code-generation ability [1, 2, 12]. Specialized pretraining objectives improved code understanding and synthesis [4, 5], and open-source code-LLMs broadened access [13, 14]. Functional benchmarks such as HumanEval [2], MBPP [15], and APPS [3] formalized correctness via test execution rather than surface similarity. Recent cross-family surveys [16] document substantial variations across model architectures, instruction alignment methods, and benchmark performance, indicating that base-model capabilities strongly influence downstream framework effectiveness. Empirically, these studies document persistent semantic errors and brittle edge-case handling, motivating mechanisms beyond raw decoding.

ExecuGraph differs by treating an off-the-shelf code-LLM as a component in a workflow rather than a self-contained solver, and by making execution outcomes the load-bearing acceptance signal.

2.2 Iterative Refinement and Self-Debugging

Chain-of-thought prompting [6] and self-consistency sampling [7] improve intermediate reasoning but rely on the same model for both generation and judgement. Reflexion [10] introduces verbal self-critique with episode memory; Self-Refine [17] iterates generation against the model’s own feedback without external signals; Self-Debug [18] prompts the same model to inspect and fix its own program with execution traces. Our single-retry condition is closest to Self-Debug rather than to Reflexion, since it regenerates from the problem statement plus the prior candidate’s stderr and carries neither verbal reflection nor episodic memory across attempts. Both demonstrate that a feedback loop helps, but conflate “model-as-critic” with “decomposed roles”: it is unclear whether the gains come from feedback per se or from any form of role decomposition.

ExecuGraph differs by separating the critic role into two distinct agents, a static Logical Reviewer (advisory) and an Evaluator (authoritative, execution-based), and by treating the Reviewer’s output as a structured, advisory artifact rather than a binding judgement. This design choice is empirically supported by recent work on LLM-as-a-judge stability: the JudgeSense benchmark [19] reports judge-decision flip rates of 8.5% to 61.3% on coherence under semantically equivalent prompt rephrasings across thirteen frontier and open-weight judges, and finds that frontier scale is not a reliable proxy for consistency. Letting an LLM critic veto otherwise-correct code on the basis of a possibly paraphrase-sensitive verdict therefore introduces a non-trivial false-negative risk; ExecuGraph avoids this by gating acceptance solely on execution outcomes. Section 4 reports a single-agent execution-retry baseline (a Reflexion-style configuration of the same framework) so that the gain attributable to the multi-agent decomposition itself is isolated.

2.3 Multi-Agent Coding Pipelines

Several recent systems explicitly decompose code synthesis into multiple roles. AgentCoder [20] introduces test-design and programmer agents; MapCoder [21] uses retrieval-planning-coding-debugging stages; MetaGPT [22] composes role-playing agents around a software-development meta-process; AutoGen [23] provides a conversational multi-agent runtime. ReAct [24] and Auto-GPT [25] explore agentic reasoning and autonomous tool use. Cemri et al. [26] taxonomize why such systems fail, reporting that a large share of failures stem from inter-agent specification and coordination breakdowns rather than from base-model capability, which is the family our planner-distractor observation (Section 5.6) belongs to. AgentBench [27] documents reliability degradation in multi-step agent settings, particularly when success depends on precise state propagation. Table 1 summarizes the structural taxonomy comparing ExecuGraph against representative multi-agent paradigms.

Table 1 Structural comparison of multi-agent and iterative-refinement code synthesis systems.

System	State Representation	Reviewer Role	Acceptance Predicate
Reflexion [10] AgentCoder [20]	Unstructured text memory Conversational turns	Self-critique (verbal) Test-generator agent	Verbal self-assessment Execution on generated tests
MapCoder [21]	Staged retrieval-plan-code-debug	Debugging agent	Execution on example tests
MetaGPT [22]	Standard-operating-procedure (SOP) memory and documents	Role-playing reviewer	Textual artifact consensus
Self-Debug [18]	Prior candidate plus execution trace	None (self-inspection)	Execution on provided tests
ExecuGraph (Ours)	Typed <code>GraphState</code> record	Advisory-only (non-gating)	Subprocess execution pass

Two recent lines of work motivate the control we build here. Olausson et al. [28] find that self-repair gains largely disappear once the cost of repair is charged against equivalent sampling, and Kapoor et al. [29] document how agent evaluations that omit cost-matched baselines systematically overstate the benefit of added structure. Neither targets multi-agent role decomposition specifically, which is the gap this study addresses: to our knowledge no prior work runs a multi-agent pipeline against a single-agent execution-retry baseline inside the same runtime and harness, which is what is required to tell decomposition apart from feedback.

ExecuGraph differs along three structural axes: (i) it uses a compiled, deterministic workflow graph (LangGraph) with a typed shared state record rather than an unconstrained conversational turn; (ii) the decision predicate is defined exclusively over execution outcomes in a subprocess sandbox, not over textual self-assessment or LLM critic approval; and (iii) the framework exposes single-switch configuration toggles that isolate the individual contributions of planning, static review, optimization, execution retry, and retrieval within a single codebase.

2.4 Execution-Grounded Validation and Program-Aided Methods

Execution-guided neural program synthesis injects runtime signals into decoding to filter invalid candidates [8]. Program-aided language models (PAL) delegate symbolic computation to interpreters [9]. Competition-level systems such as AlphaCode use large-scale sampling with execution-based filtering [11]. Toolformer [30] learns when to call external tools.

ExecuGraph differs by elevating execution from a filter to the decision predicate of the workflow itself. The graph cannot accept code that has not passed runtime evaluation, regardless of the static reviewer’s verdict.

2.5 Retrieval-Augmented Generation for Code

RAG [31] improves knowledge-intensive tasks by grounding generation in retrieved documents. Vector stores such as ChromaDB [32] and orchestrators such as LangChain [33] and LangGraph [34] enable practical RAG-and-tools pipelines, while local-first inference platforms such as Ollama [35] support reproducible deployment. Recent studies comparing traditional RAG and agentic RAG [36], and the broader agentic-RAG survey literature [37], distinguish static retrieve-then-generate pipelines from architectures in which retrieval is driven by planning, reflection, tool use, and iterative correction. By that taxonomy ExecuGraph’s retrieval layer is deliberately the *traditional* kind: a single retrieve-then-generate step feeding the Planner once per episode, with no reflection over retrieved content and no re-retrieval on failure. The agentic machinery in our system sits in the execution-feedback loop, not in the retrieval path. This matters for interpreting our null retrieval result (Section 6.5), which bounds the value of static technique retrieval and says nothing about agentic retrieval. It also points at the more promising direction for code: repository-level retrieval of project-specific context, as in RepoCoder [38], rather than retrieval of textbook algorithms that a code model has already internalized.

ExecuGraph differs by implementing retrieval as an optional, configuration-toggled input to the Planner agent only, rather than an unguided context-injection step, and by providing explicit on/off ablation experiments to measure its marginal impact on code correctness.

3 The ExecuGraph Framework

We first present the high-level architecture (Section 3.1), then specify the workflow as a typed transition system (Section 3.2), describe each agent (Section 3.4), and document the execution sandbox (Section 3.6).

3.1 High-Level Architecture

ExecuGraph follows a layered design (Fig. 1). A *User Interaction Layer* accepts a backend / DSA problem statement. A *Workflow Orchestration Layer* (LangGraph) compiles a directed graph that schedules agent invocations and routes control based on the typed shared state. An *Agent Layer* contains six specialized agents (Planner, Code

Generator, Logical Reviewer, Evaluator, Optimizer, Explainer). An *Execution and Evaluation Layer* runs the generated code in a subprocess-isolated sandbox and applies a hybrid testing strategy. An *optional Knowledge Memory Layer* (ChromaDB) indexes 15 core algorithmic technique templates (spanning dynamic programming state transitions, graph traversals, and custom data structures) embedded via `all-MiniLM-L6-v2` and retrieved using cosine similarity ($k = 3$). It supplies retrieved snippets to the Planner only, and is config-toggled and disabled by default in headline benchmarks, with a dedicated on/off ablation in Section 5.

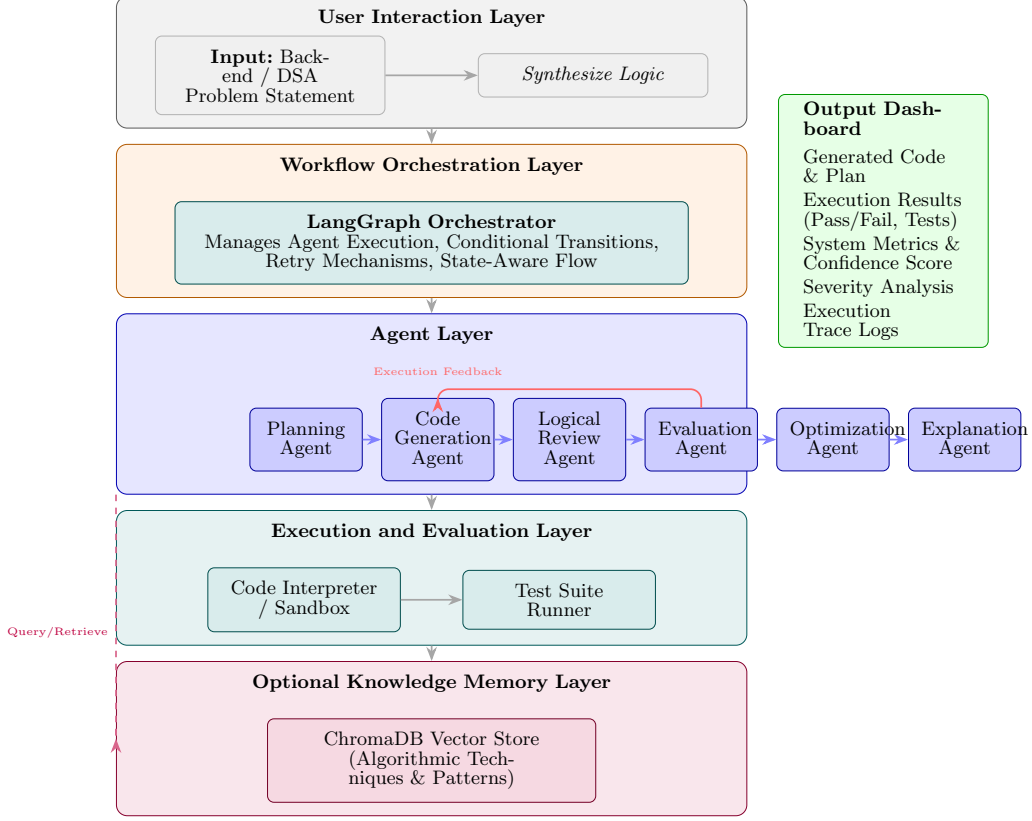


Fig. 1 Five-layer architecture of ExecuGraph. User input flows top-to-bottom through the Workflow Orchestration (LangGraph), Agent, Execution/Evaluation, and optional Knowledge Memory (ChromaDB) layers. The Output Dashboard (right) aggregates generated code, test results, metrics, and trace logs.

The framework is governed by four design principles: (i) *separation of concerns*, meaning each agent has a single, well-defined output type; (ii) *execution-centric validation*, so acceptance is determined by runtime outcomes, not by static review; (iii) *deterministic workflow control*, so the orchestrator is a compiled transition graph with typed state, conditional edges, and a hard retry cap; and (iv) *reproducibility*,

through fixed inference configurations, persisted seeds, and per-trial JSON logs that regenerate every result table.

3.2 Workflow Semantics

We formalize the framework as a typed transition system $\mathcal{W} = (\mathcal{S}, A, \rho, D, T)$ over a shared state record $\mathcal{S}$: A is the agent set, ρ the routing function, D the acceptance predicate, and T the termination condition (Fig. 2 shows the corresponding control-flow graph). The graph carries a retry back-edge and so is not acyclic; termination is established below by a decreasing-budget argument rather than by acyclicity.

Shared state.

$\mathcal{S}$ is a typed record (Listing 1) that persists for the lifetime of one synthesis episode. Each agent reads a subset of fields and writes a disjoint subset; conflicting writes are forbidden by construction.

Listing 1 Shared graph state (TypedDict).

```
class GraphState(TypedDict):
    problem: str # input problem statement
    plan: str # Planner output (algorithm + complexity)
    code: str # current candidate code
    review: dict # Reviewer output, structured (advisory)
    evaluation: dict # {passed, test_results, error_class, ...}
    explanation: str # Explainer output (final)
    retries: int # number of regeneration attempts so far
    retry_budget: int # max regenerations (config)
    passed: bool # set True iff Evaluator returned passed
    optimized: bool # set True after successful Optimizer pass
    enable_planner: bool # ablation toggles
    enable_reviewer: bool
    enable_optimizer: bool
    enable_rag: bool
    retrieved_techniques: list # Planner input from ChromaDB (or [])
    cost: dict # tokens / wallclock / call counts
    logs: list[str]
```

Agent set.

$A = \{\text{plan, gen, rev, eval, opt, exp}\}$. Each $a \in A$ is a partial function $a : \mathcal{S} \rightarrow \Delta\mathcal{S}$, where $\Delta\mathcal{S}$ is a state delta. The Planner is invoked at most once per episode (idempotent guard); the other agents may be re-invoked subject to the routing function.

Decision predicate.

The acceptance predicate is purely execution-driven:

$$D(s) \equiv s.\text{evaluation.passed} = \text{True}. \quad (1)$$

Static-review verdicts do not appear in D .

Routing.

The routing function $\rho : \mathcal{S} \rightarrow A \cup \{\text{end}\}$ is

$$\rho(s) = \begin{cases} \text{opt} & \text{if } D(s) \wedge \neg s.\text{optimized} \wedge s.\text{enable_opt}, \\ \text{exp} & \text{if } D(s) \vee s.\text{retries} \geq s.\text{retry_budget}, \\ \text{gen} & \text{otherwise.} \end{cases} \quad (2)$$

The Optimizer, when enabled, re-invokes the Evaluator before its output is accepted, preventing optimization-induced regressions.

Termination.

T holds when control reaches end via ρ . Termination is guaranteed because (i) $s.\text{retries}$ strictly increases on every gen re-entry, (ii) the budget is finite, and (iii) the success branch is acyclic.

Run loop.

Algorithm 1 states the workflow procedure compactly. The notation matches the transition system above: s is the shared state, δ_a is the state delta produced by agent a , SANDBOX is the subprocess-isolated executor of Section 3.6, and B is the retry budget configured per run.

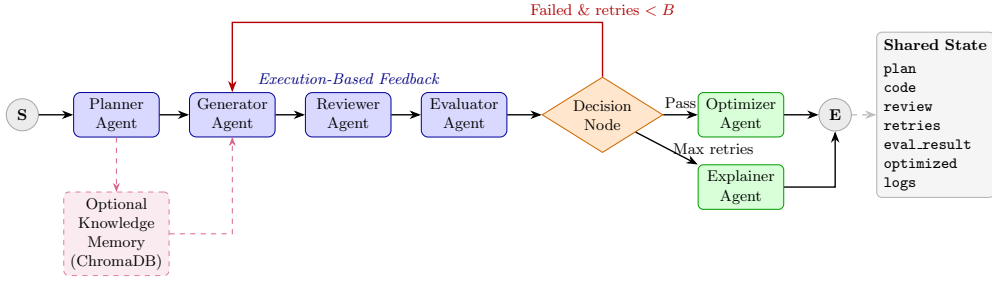


Fig. 2 ExecuGraph workflow DAG. Solid arrows show the main pipeline ($S \rightarrow \text{Planner} \rightarrow \text{Generator} \rightarrow \text{Reviewer} \rightarrow \text{Evaluator} \rightarrow \text{Decision} \rightarrow \text{Optimizer/Explainer} \rightarrow E$). The red curved arrow is the retry loop, active when $\text{retries} < \text{retry_budget}$. Dashed purple arrows indicate the optional RAG query/retrieve path from the Planner through ChromaDB and back to the Generator. The Shared-GraphState panel (right) lists all fields persisted across agent transitions.

3.3 Agent Categories and System Mechanics

ExecuGraph divides its six specialized agents into two functional categories:

1. **Core Correctness Agents:** The *Planner*, *Code Generator*, and *Evaluator* form the primary execution-grounded synthesis loop directly influencing functional correctness.

Algorithm 1 ExecuGraph synthesis loop.

Require: problem q , retry budget B , agent toggles τ , LLM backends, sandbox timeout θ

Ensure: final state s with code $s.code$, evaluation $s.evaluation$, and cost log

```

1:  $s \leftarrow \text{INITSTATE}(q, B, \tau)$ 
2: if  $\tau.\text{planner}$  then
3:    $s.\text{plan} \leftarrow \text{PLANNER}(s.\text{problem}, \text{RETRIEVE}(s) \text{ if } \tau.\text{rag})$ 
4: end if
5:  $r \leftarrow 0$  // retries used
6: loop
7:    $s.code \leftarrow \text{GENERATOR}(s.\text{problem}, s.\text{plan}, s.\text{stderr})$ 
8:   if  $\tau.\text{reviewer}$  then
9:      $s.\text{review} \leftarrow \text{REVIEWER}(s.code)$  // advisory only
10:  end if
11:   $s.evaluation \leftarrow \text{EVALUATOR}(s.code; \theta)$ 
12:  if  $s.evaluation.\text{passed}$  then
13:    break
14:  end if
15:  if  $r \geq B$  then
16:    break
17:  end if
18:   $r \leftarrow r + 1$ 
19:   $s.\text{stderr} \leftarrow s.evaluation.\text{stderr}$ 
20: end loop
21: if  $s.evaluation.\text{passed}$  and  $\tau.\text{optimizer}$  then
22:    $c' \leftarrow \text{OPTIMIZER}(s.code)$ 
23:    $e' \leftarrow \text{EVALUATOR}(c'; \theta)$  // re-validate
24:   if  $e'.$ passed then
25:      $s.code \leftarrow c'; s.evaluation \leftarrow e'$ 
26:   end if
27: end if
28:  $s.\text{explanation} \leftarrow \text{EXPLAINER}(s.\text{problem}, s.code, s.\text{plan})$ 
29:
30: return  $s$ 

```

2. Auxiliary Artifact Agents: The *Logical Reviewer* (advisory static critic), *Optimizer* (post-acceptance refactoring), and *Explainer* (post-acceptance documentation) produce secondary artifacts without gating program acceptance.

An *Execution and Evaluation Layer* runs the generated code in a subprocess-isolated sandbox and applies a hybrid testing strategy. An *optional Knowledge Memory Layer* (ChromaDB) supplies retrieved algorithmic-technique snippets to the Planner only; it is config-toggled and disabled by default in headline benchmarks, with a dedicated on/off ablation in Section 5.

Reduction to baselines.

The same framework collapses to two baselines via configuration:

- *Single-agent one-shot:* `enable_planner = enable_reviewer = enable_opt = False`, `retry_budget = 0`. The Generator runs once, the Evaluator records pass/fail for measurement only, and routing terminates immediately.
- *Single-agent execution-retry (Self-Debug-style):* same as above but `retry_budget > 0`. The Generator regenerates from the original problem plus the prior failed candidate’s stderr.

This reduction is what enables the per-component ablations of Section 5.

3.4 Agents

Planner.

Reads problem (and optionally retrieved.techniques). Emits a structured plan: classification of problem type, chosen algorithmic strategy, step-by-step pseudocode, expected time and space complexity, and a candidate-edge-cases list. Backed by an instruction-tuned LLM at temperature 0.2.

Code Generator.

Reads problem and plan. Emits a single Python function. Backed by a code-specialized LLM at temperature 0.0 for determinism.

Logical Reviewer.

Reads code. Emits a structured JSON object `{invariants, potential_failures, severity}`, marked *advisory*. The framework explicitly does not gate on its output. This is the key change from systems that let a textual critic block correct code; the design is motivated both by an early-development failure mode in which a confidently-wrong reviewer suppressed correct candidate code, and by prior co-authored benchmark evaluation showing that LLM-as-judge verdicts are unstable under paraphrase [19].

Evaluator.

The acceptance authority. Reads code and problem. Sanitizes the code, executes it in a subprocess sandbox (Section 3.6), and applies a hybrid testing strategy:

1. If the problem is in the curated benchmark, deterministic invariant tests are dispatched (with multi-name signature aliasing).
2. Otherwise, a code-specialized LLM is asked to emit a JSON test-suite of the form `[{call, expected, description}]`, parsed by a strict JSON parser, and each test is executed via a sandboxed expression evaluator.
3. If neither yields tests, a smoke test (executes-without-error) is recorded as the only signal and the trial is flagged as test-poor in the JSON log.

Returns `{passed: bool, test_results: list, error_class: str, stderr: str}`.

Optimizer.

Runs only when $D(s)$ holds. Receives the passing code and emits an optimized variant. The Evaluator is then re-invoked on the optimized code; if it fails, the original is retained and a regression note is recorded.

Explainer.

Always runs as the terminal node. Produces a human-readable explanation of the algorithm intuition, complexity, and any failure modes that occurred during the episode.

3.5 Worked Example: Topological Sort

To make the data flow concrete, Listing 2 walks through the `topo_sort` problem from the curated suite, an internally designed item. The generated function is reproduced verbatim from the multi-full run in `results/grid-main/e1mf/trials.jsonl` (`problem_id = topo_sort`, trial 0), and the evaluation outcome is the one recorded there. The surrounding per-agent annotations are illustrative rather than transcribed: the per-trial log persists the final candidate program, cost counters, and the evaluation result, but not the Planner, Reviewer, Optimizer, and Explainer intermediate texts, which are held in the shared state for the duration of an episode and not serialized. We show them in the shape the agents emit in order to make the data flow legible, and we flag the distinction here so that the listing is not mistaken for a recoverable log record. Persisting per-agent outputs would enable the qualitative analysis of Reviewer utility that we identify as future work in Section 7. The listing illustrates the canonical happy path (plan, generate, review, evaluate, optimize, explain) without retries.

3.6 Execution Sandbox

Because every agent decision rests on the Evaluator, the sandbox is load-bearing: a defect in it does not merely weaken isolation, it corrupts the dependent variable. Generated code runs in a freshly spawned Python subprocess (`subprocess.run`, with `-I` to suppress user site-packages) under five controls: (i) a hard wall-clock timeout, default 5.0s and configurable per benchmark; (ii) an import policy, described below; (iii) removal of the filesystem-reaching builtins (`open`, `breakpoint`, `help`) from the child, with `input` retained because it reads only the harness-supplied in-memory stdin

Listing 2 Per-agent trace for the `topo_sort` problem (truncated for space). The generated function and the evaluation outcome are reproduced verbatim from the released trial log; **the per-agent annotations are illustrative reconstructions**, since the runner persists the final candidate and the evaluation result but not the intermediate agent texts. Acceptance is decided by the Evaluator’s `passed:true`; the Reviewer’s verdict is recorded but not consulted.

```
[PLANNER] (qwen2.5-coder:7b-instruct)
Category : Graph
Algorithm : Kahn's algorithm (BFS topological sort)
Steps : build adj list -> compute indegrees ->
        BFS over zero-indegree nodes -> if
            visited != N return [] (cycle)
Complexity : O(V + E) time, O(V) space

[GENERATOR] (qwen2.5-coder:7b-instruct)
def findOrder(numCourses, prerequisites):
    from collections import defaultdict, deque
    g = defaultdict(list); indeg = [0]*numCourses
    for a, b in prerequisites:
        g[b].append(a); indeg[a] += 1
    q = deque(i for i in range(numCourses)
              if indeg[i] == 0)
    out = []
    while q:
        u = q.popleft(); out.append(u)
        for v in g[u]:
            indeg[v] -= 1
            if indeg[v] == 0: q.append(v)
    return out if len(out) == numCourses else []

[REVIEWER] (advisory; not consulted by decision)
invariants : ["acyclic implies len(out)==N",
             "cycle implies []"]
potential_failures: []
severity : low

[EVALUATOR] (subprocess sandbox, 5s timeout)
test 1 linear chain passed
test 2 cycle (2-node) passed
test 3 single task passed
test 4 branching deps passed (judge: valid order)
test 5 complex cycle passed
result: passed=true, error_class=None

[OPTIMIZER] no functional change; re-evaluation passed
[EXPLAINER] returns Kahn's-algorithm explanation
```

buffer; (iv) an explicitly constructed environment (`env={...}`) so the child inherits none of the parent’s variables, which during experiment runs include provider credentials such as `HF_TOKEN`; and (v) `stdout` and `stderr` captured for JSON logging.

The import policy deserves description because its obvious implementation is wrong, and we shipped that wrong implementation in the original experiment grid. Gating `__import__` against a flat allow-list of module names rejects correct programs: importing an allow-listed pure-Python module transitively imports its private C accelerator, so `bisect` pulls `_bisect`, `heapq` pulls `_heapq`, and `io` pulls `_io`. None of those roots appear on the allow-list, so the import was refused and the trial was recorded as a sandbox violation. The corrected child instead pre-imports every allow-listed module *before* installing the guard, so accelerators enter `sys.modules` through the ordinary machinery; it then admits any import whose root is allow-listed or already loaded, and

refuses a deny-list (`os`, `subprocess`, `socket`, `ctypes`, `importlib`, and similar) unconditionally, since an allow-listed module can pull a dangerous one transitively, `random` imports `os` for `urandom`. Because `sys` and `io` are needed for stdin/stdout problems but expose `sys.modules` (which reaches any loaded module) and `io.open` (which is `open` under another name), the child serves restricted shims for both, forwarding only the standard streams. On POSIX an address-space ceiling of 1GB is applied with `resource.setrlimit`; CPython ships no `resource` module on Windows, so on the Windows host used for our measurements (Section 4.4) this ceiling is not enforced, and we report the platform caveat rather than the intended guarantee.

Section 5.12 quantifies how the original defect affected the reported results and describes the re-scoring procedure that corrects them. Under the corrected sandbox, no trial in any condition terminates in a sandbox violation (Table 14). Remaining limitations are discussed in Section 6.7.

3.7 Implementation Environment

ExecuGraph is implemented in Python and orchestrated with LangGraph [34] and LangChain [33]. LLM access is provider-agnostic: the default backend uses Ollama [35] with locally hosted Qwen2.5-7B-Instruct and Qwen2.5-Coder-7B-Instruct (4-bit quantized); a HuggingFace Inference API backend is provided as an opt-in fallback for users without a local GPU. The optional retrieval layer uses ChromaDB [32] with a small set of seeded algorithmic-technique entries. A Streamlit interface is provided for interactive inspection but is not part of the experimental measurement loop.

4 Experimental Protocol

4.1 Datasets

We evaluate on three suites with deliberately overlapping but distinct purposes.

Internal-30.

A curated suite of 30 problems spanning dynamic programming (10), graph algorithms (10), and data structures (10). 27 are internally designed; 3 are adapted from the APPS-introductory partition [3], marked explicitly in the benchmark loader. Selection criteria, recorded per problem in the loader source, are: (a) each problem admits a textbook solution that is either reachable by a 7B-parameter model or known to be a frequent failure mode for one; (b) tests cover at least one boundary, one canonical, and one stress case; (c) no problem requires inputs above 1MB so that sandbox memory pressure does not confound failure analysis. Crucially, no prompt engineering or agent prompt tuning was performed against these 30 problem statements during framework development, enforcing strict prompt-isolation hygiene and preventing leakage. The full list with selection rationale is shipped with the code (`execugraph/benchmarks/internal30.py`). Table 2 lists all 30 problems by category.

Table 2 Internal-30 benchmark problems by category. † marks the three problems adapted from APPS-introductory; all others are internally designed.

Category (10 each)	Problem ID	Description
Dynamic Programming	fib	n -th Fibonacci number
	climb_stairs	Staircase (1- or 2-steps)
	coin_change	Minimum coins
	lcs	Longest common subsequence
	house_robber	Non-adjacent max sum
	min_cost_path	Grid min-cost path
	lis	Longest increasing subsequence
	rod_cutting	Rod-cutting maximum profit
	matrix_chain	Matrix-chain multiplication
	subset_sum [†]	Boolean subset-sum (APPS)
Graph Algorithms	topo_sort	Topological sort
	cycle_detect	Directed-graph cycle detection
	bfs	Breadth-first search
	dfs	Depth-first search
	dijkstra	Single-source shortest path
	bellman_ford	Single-source shortest path, negative edges
	connected_components	Undirected CC count
	scc	Strongly connected components
	unweighted_shortest	Unweighted BFS shortest path
	grid_shortest_path [†]	Grid BFS shortest path (APPS)
Data Structures	lru_cache	LRU cache (get/put)
	min_stack	Stack with $O(1)$ minimum
	stack_using_queue	Stack implemented via queue
	reverse_linked_list	Singly-linked list reversal
	binary_tree_traversal	In/pre/post-order traversal
	priority_queue	Min-heap priority queue
	heapify	In-place heapify
	avl	AVL tree insert/balance
	adj_list_construct	Adjacency list construction
	valid_parentheses [†]	Balanced-bracket check (APPS)

HumanEval.

To evaluate external validity on canonical Python functions we use HumanEval [2] in full: all 164 problems, every condition, no sampling. Running the complete suite removes the sampling step and the need to characterise it, and it is what gives the paired analysis in Section 5 enough problems to resolve differences at all. We run the official test suites verbatim inside the subprocess sandbox.

APPS-introductory subset.

To assess performance on competitive programming challenges, we select a 50-problem subset from the APPS introductory test partition [3]. Problems are selected via uniform random sampling from the introductory split with fixed seed 42, representing entry-level competition problems with test cases taken directly from the dataset. The selected identifiers are recoverable from the `problem_id` field of the released per-trial logs.

4.2 Conditions

Five conditions, all instantiated as configurations of the same workflow:

- **single-oneshot:** Generator only, no retries.

- **single-retry**: Generator + Evaluator, retry budget swept; no Planner, no Reviewer, no Optimizer.
- **multi-full**: All agents enabled; retry budget = 2; RAG off (default).
- **multi-no-retry**: Multi-full with retry budget = 0 (no retries).
- **multi-no-X** for $X \in \{\text{planner, reviewer, optimizer}\}$, together with **rag-on**, which *adds* the retrieval layer rather than removing a component: ablations.
- **retry-sweep**: **multi-full** with retry budget $\in \{0, 1, 2\}$.

4.3 Models

We use three open-weight LLMs spanning two vendor families, all locally hosted via Ollama [35] with 4-bit `q4_K.M` quantization, accessed through the same provider-agnostic backend so that no per-condition prompting differences exist outside of the configuration files. To eliminate the model-swap overhead that would otherwise dominate wall-clock on a 6 GB-VRAM laptop GPU, the five LLM-backed agents in a given run share a single backbone (the Evaluator executes deterministic tests and invokes no model):

- **Primary backbone (headline numbers)**: `qwen2.5-coder:7b-instruct-q4_K.M` [39] for all five LLM-backed agents. We selected a 7 B-class dense code model for the headline grid because it fits entirely in 6 GB VRAM with comfortable KV-cache headroom, requiring no RAM offload, and yields $\sim 30\text{--}80$ s per multi-full trial on the reference hardware. This makes the full `E1+E4+E5+E2+E3+E7` grid feasible inside a ~ 7 -hour wall-clock budget. The Planner / Reviewer / Explainer are run at temperature 0.2; the Generator and Optimizer at temperature 0.0.
- **Cross-model condition (independent vendor)**: `deepseek-coder-v2:16b-lite-instruct-q4_K.M` [40] from DeepSeek-AI, a 16B Mixture-of-Experts model with 2.4B active parameters. We initially evaluated OpenAI’s `gpt-oss:20b` [41] as the cross-model candidate, but at 20B dense parameters its inference latency under 6 GB-VRAM offload (~ 600 s per multi-full trial) made the full cross-model grid infeasible inside our compute budget; we therefore use DeepSeek-Coder-V2-Lite, which gives equivalent vendor diversity at materially higher throughput. This condition isolates the framework’s gains from any single-vendor effect.
- **Latest-2026 supplementary**: `qwen3-coder:30b-a3b-q4_K.M` [42], the strongest agentic-code-tuned open-weight model that runs on the reference hardware as of May 2026. Mixture-of-Experts with 30 B total parameters but only 3.3 B active per token; ~ 13 GB of weights spill to system RAM, yielding ~ 270 s per multi-full trial. Reported on a 10-problem stratified subset of internal-30 (Table 15), where Multi-Full and Single-Oneshot both reach 90.0% while Multi-Full consumes roughly $6\times$ the wall-clock and $12\times$ the tokens. With ten problems at a single trial each, this subset can only detect very large differences, so we report it as an indication that the added structure does not pay for itself where the base model already succeeds, and not as a measured equivalence.

Model digests, quantization tags, and the exact `ollama pull` commands are recorded in `REPRODUCIBILITY.md`. We deliberately do not use any closed-weight or

paid-API model in any condition: the entire grid is reproducible at zero marginal API cost on the hardware specified in Section 4.4.

4.4 Hardware and Software

All measurements were taken on a single machine: 13th Gen Intel Core i5-13420H (8C/12T, 2.10 GHz base), 16 GB DDR5, NVIDIA GeForce RTX 4050 Laptop GPU (6 GB VRAM), Windows 11 64-bit. Python 3.11, LangGraph 1.0.5, LangChain 1.2.7, Ollama (latest GA at time of run; pinned digest in REPRODUCIBILITY.md).

4.5 Trials and Seeds

Each (problem, condition, model) tuple is repeated for N independent trials with seeds $\{0, 1, \dots, N - 1\}$ passed to the LLM backend where the provider supports it. Trial allocation reflects a trade-off between precision and the wall-clock budget of a single 6 GB laptop GPU, and it is uneven: $N = 5$ for the three headline conditions on internal-30 and for the per-agent ablations, $N = 2$ for the retry-budget sweep and the cross-model evaluation, and $N = 1$ for HumanEval, APPS-introductory, and the supplementary 30B subset, where the problem count rather than the trial count supplies whatever power the comparison has.

We draw attention to this unevenness because it limits what the secondary tables can support. The cross-model rows rest on two trials per problem, so a one-trial change in a ten-problem category moves that cell by 5.0 pp, and differences of that magnitude in Table 13 are within run-to-run noise. The ablations are run at $N = 5$ so that every row in Table 8, including the multi-full baseline, rests on the same number of trials; Section 6 uses the architecturally inert rows of that table to estimate the noise floor directly rather than assuming one. Table 3 enumerates the executed grid.

Table 3 Executed trial grid. Counts are distinct (problem, trial) records after the deduplication described in Section 5.12; three runs had been resumed with output appended to an existing log, contributing 157 duplicate rows that are excluded here and from every reported number. The HumanEval 64-problem rows were superseded by the full-suite runs and are retained for the correction analysis in Table 16 only; results reported in Section 5 draw on 2,312 of the 2,440 executed trials.

Experiment	Conditions	Problems	N	Trials
Headline (internal-30)	SO, SR, MF	30	5	450
Per-agent ablation	4 variants	30	2	240
Per-agent ablation (repeat)	4 variants	30	5	600
Retry-budget sweep	budget $\in \{0, 1, 2\}$	30	2	180
Cross-model (DeepSeek)	SO, SR, MF	30	2	180
HumanEval (initial subset)	SO, MF	64	1	128
HumanEval (full suite)	SO, SR, MF	164	1	492
APPS-introductory	SO, SR, MF	50	1	150
Supplementary 30B	SO, MF	10	1	20
Total				2,440

The grid was executed in two passes. The first established the headline comparison and the ablations at $N = 2$; the second, run after the sandbox correction of

Section 5.12, added single-retry to every external and cross-model condition, raised the ablations to $N = 5$ so they match the headline baseline, extended HumanEval from the initial 64-problem subset to the full suite, and filled in retry budget 1. The seed flows through to the generator sampling RNG and the LLM-test sampler. It does not randomize execution order: the runner iterates problems and trials in a fixed nested loop, so ordering is deterministic by construction rather than by seeding.

4.6 Metrics

Per (problem, condition, model, trial):

- **passed**: 1 iff all tests pass.
- **tests_passed** / **tests_total**: per-test pass rate.
- **retries_used**: number of regenerations consumed.
- **error_class**: one of {`none`, `syntax`, `runtime`, `timeout`, `wrong_answer`, `empty_output`, `sandbox_violation`}.
- **wallclock_s**: end-to-end seconds.
- **tokens_in**, **tokens_out**, **llm_calls**: cost.

Aggregated metrics: pass-rate (mean over trials), retries-mean, execution-failure-rate (any `error_class` $\neq$ `none`, `wrong_answer`), 95% bootstrap CI on totals, paired Wilcoxon p -value vs. baseline, McNemar test on per-problem outcome vectors.

4.7 Statistical Tests

For each category we test the null hypothesis that the per-problem pass-rate vector is the same under **single-oneshot** and **multi-full**. We use the exact Wilcoxon signed-rank test on the per-problem mean pass-rate over N trials, implemented in `execugraph/analysis/stats.py`. We do not report McNemar’s test: binarizing $N = 5$ pass-rates at a trial-majority threshold discards the graded information those trials were run to obtain, and at the discordant-pair counts we observe it is no better powered than the signed-rank test. We report raw two-sided p -values (unadjusted for multiple comparisons; Holm-Bonferroni adjusted values for the three contrasts are $p_{\text{SR} \rightarrow \text{SO}} = 0.375$, $p_{\text{MF} \rightarrow \text{SR}} = 0.624$, $p_{\text{MF} \rightarrow \text{SO}} = 0.624$); given the small per-category problem count we additionally report effect size (mean paired difference) and a 95% bootstrap CI.

4.8 Experimental Procedure

Each run is launched by `python scripts/run_experiment.py --config configs/<name>.yaml --output results/<run_id>`. The runner: (1) records the host environment, model digest, and config; (2) iterates over (problem $\times$ trial) tuples; (3) appends one JSON line per trial to `results/<run_id>/trials.jsonl`; (4) writes a summary report. Every table in Section 5 is generated from these JSONL files by `python scripts/build_all_tables.py`, which writes one `.tex` fragment per table plus a machine-readable `all_numbers.json` that the figures and prose values are checked against. No numbers are entered manually. The generator covers every results table; Tables 1, 2 and 19 are descriptive and hand-authored.

5 Results

All numbers in this section are produced by `scripts/build_all_tables.py` from the per-trial logs in `results/`. Every table in this paper has a generator; none contains a hand-entered value.

5.1 Headline Comparison: Internal-30

Table 4 reports per-problem pass-rate (mean over $N = 5$ trials) for the three primary conditions on the internal-30 suite; Table 5 reports category-level aggregates with paired Wilcoxon p -values for both $SO \rightarrow MF$ and $SR \rightarrow MF$ contrasts. None of the per-category p -values reach the conventional $\alpha = 0.05$ threshold: at $n=10$ problems per category the test is underpowered for the observed effect sizes. Figure 3 plots the same category-level rates. The pooled- $n = 30$ analysis (across all internal-30 problems) is reported and discussed in Section 6.

Table 4 Per-problem pass-rate (% , evaluated over $N = 5$ independent trials per problem, yielding strictly quantized 20.0% step pass rates) on internal-30. SO = single-oneshot, SR = single-retry (budget=2), MF = multi-full.

Problem	Cat	Src	SO %	SR %	MF %
climb_stairs	DP	internal	100.0	100.0	100.0
coin_change	DP	internal	100.0	100.0	100.0
fib	DP	internal	100.0	100.0	100.0
house_robber	DP	internal	100.0	100.0	80.0
lcs	DP	internal	100.0	100.0	100.0
lis	DP	internal	100.0	100.0	100.0
matrix_chain	DP	internal	0.0	0.0	0.0
min_cost_path	DP	internal	100.0	100.0	100.0
rod_cutting	DP	internal	100.0	100.0	100.0
subset_sum	DP	APPS [†]	100.0	100.0	100.0
bellman_ford	GRAPH	internal	100.0	100.0	100.0
bfs	GRAPH	internal	100.0	100.0	80.0
connected_components	GRAPH	internal	100.0	100.0	100.0
cycle_detect	GRAPH	internal	0.0	100.0	60.0
dfs	GRAPH	internal	0.0	100.0	100.0
dijkstra	GRAPH	internal	100.0	100.0	100.0
grid_shortest_path	GRAPH	APPS [†]	0.0	0.0	0.0
scc	GRAPH	internal	0.0	0.0	60.0
topo_sort	GRAPH	internal	100.0	100.0	100.0
unweighted_shortest	GRAPH	internal	100.0	100.0	20.0
adj_list_construct	DS	internal	100.0	100.0	100.0
avl	DS	internal	0.0	100.0	0.0
binary_tree_traversal	DS	internal	100.0	100.0	100.0
heapify	DS	internal	100.0	100.0	100.0
lru_cache	DS	internal	100.0	100.0	100.0
min_stack	DS	internal	100.0	100.0	60.0
priority_queue	DS	internal	20.0	100.0	100.0
reverse_linked_list	DS	internal	100.0	100.0	100.0
stack_using_queue	DS	internal	100.0	100.0	100.0
valid_parentheses	DS	APPS [†]	100.0	100.0	100.0

Two problems illustrate why the category means move at all. On `priority_queue`, one-shot reaches only 20.0% while both execution-feedback conditions reach 100.0%:

the first attempt typically mismanages the heap invariant, and a single runtime traceback is enough to repair it. On `scc`, one-shot and single-retry both score 0.0% while multi-full reaches 60.0%, the clearest case where the Planner’s decomposition appears to help; conversely `avl` is solved by single-retry (100.0%) and never by multi-full (0.0%). Each of these rests on five trials of a single problem, so we offer them as illustrations of the mechanism rather than as evidence for it.

Table 5 Category-level pass-rate (%) on internal-30. *p*-values are *exact* paired Wilcoxon signed-rank tests against single-oneshot ($p_{\text{SO} \rightarrow \text{MF}}$) and single-retry ($p_{\text{SR} \rightarrow \text{MF}}$); the normal approximation is not valid at these discordant-pair counts. With ten problems per category these per-category tests are severely underpowered and are reported for completeness only, unadjusted for multiplicity; the pooled internal-30 analysis in Table 18 is preferred over them, and the adequately powered HumanEval analysis in Table 11 over both. Pooled across all 30 problems, pass rates are 77.3% (SO), 90.0% (SR), and 82.0% (MF). HumanEval and APPS-introductory results are in Table 10.

Category	SO	SR	MF	$p_{\text{SO} \rightarrow \text{MF}}$	$p_{\text{SR} \rightarrow \text{MF}}$
DP	90.0	90.0	88.0	1.000	1.000
GRAPH	60.0	80.0	72.0	0.625	0.625
DS	82.0	100.0	86.0	1.000	0.500

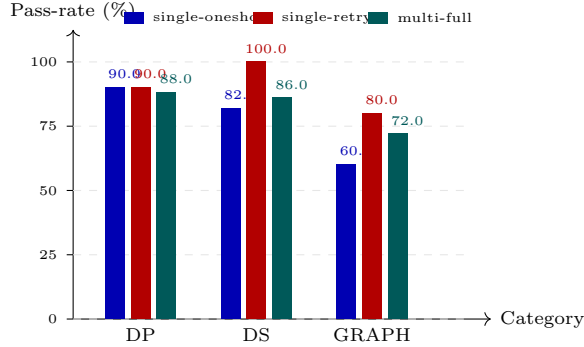


Fig. 3 Category-level pass-rate by condition on internal-30 (values from Table 5). Error bars are omitted because none of the pairwise category gaps is statistically separable at ten problems per category (Table 5); the figure shows the pattern, not evidence for it.

5.2 Execution-Failure Breakdown

Table 6 reports the rate of executions that ended in any error class other than `wrong_answer` (i.e. crashes, timeouts, syntax errors, sandbox violations). This is a

stricter signal than overall pass-rate because it measures whether the generated artifact even runs.

Table 6 Execution-failure rate (%) on internal-30: trials whose error_class $\notin \{\text{none}, \text{wrong_answer}\}$. Lower is better.

Category	SO	SR	MF
DP	0.0	0.0	0.0
GRAPH	10.0	0.0	0.0
DS	18.0	0.0	10.0

5.3 Cost Analysis

A multi-agent workflow with retries is necessarily more expensive than a one-shot baseline. Table 7 reports the cost-correctness trade-off across the three primary conditions, and Figure 4 plots accuracy against wall-clock so the frontier is visible.

Table 7 Cost per problem (mean $\pm$ stdev). Wallclock includes LLM and sandbox time. `calls` is the count of LLM invocations.

Condition	Wallclock (s)	Tokens	LLM calls
single-oneshot	7.6 $\pm$ 2.5	313 $\pm$ 75	1.0 $\pm$ 0.0
single-retry	13.3 $\pm$ 15.1	507 $\pm$ 445	1.5 $\pm$ 0.8
multi-full	59.8 $\pm$ 22.9	3710 $\pm$ 722	5.3 $\pm$ 0.5

5.4 Per-Agent Ablation

Table 8 disables one agent at a time, measuring the marginal contribution of the Planner, Reviewer, Optimizer, and the retrieval layer.

5.5 Retry-Budget Sweep

Table 9 reports pass-rate, average retries used, and mean wallclock as the configured retry budget varies over $\{0, 1, 2\}$. Figure 5 plots the step-by-step cumulative pass-rate trajectory as a function of actual retries consumed (0, 1, 2 retries used) during execution within the budget-2 runs.

5.6 Qualitative Trace Analysis: Planner Distractor Mechanism

Multi-full trails single-retry on internal-30 (82.0% versus 90.0%) even though it exceeds one-shot, so the question this subsection addresses is narrow: on which problems does

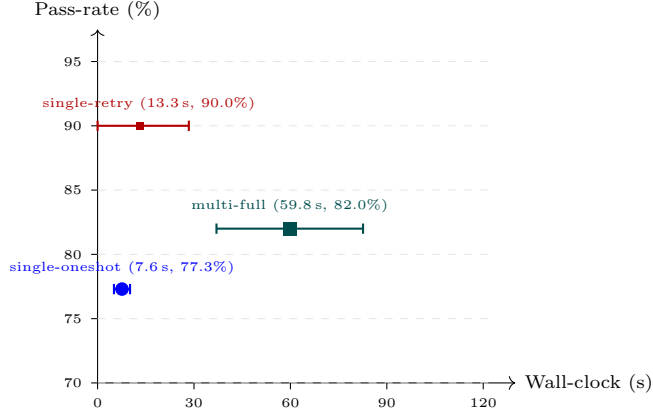


Fig. 4 Cost-correctness trade-off: mean wall-clock per problem (Table 7) versus pass-rate, one point per condition; horizontal bars are $\pm 1\sigma$ of wall-clock. Note that the vertical axis is truncated at 70%, which visually enlarges differences that Table 18 shows are not statistically separable. Single-retry’s wall-clock distribution is strongly right-skewed (mean 13.3s, median 8.1s), because most problems succeed on the first attempt and a minority consume the full retry budget; its -1σ bar is therefore clipped at zero and the mean overstates the typical cost.

Table 8 Per-agent ablation on internal-30, all rows at $N = 5$ trials per problem. Rows marked * cannot alter whether a program is accepted (Section 3.3): the Reviewer’s output is never read by any downstream agent, the Optimizer reverts on re-validation failure, and retrieval feeds only the Planner. They are therefore independent replications of multi-full rather than ablations, and they serve as negative controls. All three return exactly 78.7% (118 of 150 trials) despite differing at the level of individual trials and generated programs, which puts the within-session noise floor for the aggregate near zero. Category cells still move between them, so the per-category figures carry more noise than the aggregate. Against the 78.7% replication level, removing the Planner gains +8.0 pp.

Condition	DP %	Graph %	DS %	Overall %
Multi-Full (baseline)	88.0	72.0	86.0	82.0
–Planner	90.0	80.0	90.0	86.7
–Reviewer*	90.0	64.0	82.0	78.7
–Optimizer*	90.0	64.0	82.0	78.7
+RAG*	90.0	64.0	82.0	78.7

Table 9 Retry-budget sweep on internal-30, condition **multi-full**, $N = 2$ per problem for all three rows. Budget 0 is Multi-Full restricted to zero retries, still running Planner, Generator, Reviewer, and Evaluator once, which is why its cost (41.2s, 3,341 tokens) far exceeds Single-Oneshot’s (7.6s, 313 tokens in Table 7); the two are not interchangeable. The sweep is non-monotonic: budget 1 scores below budget 0. Since a larger budget can only ever add regeneration attempts to a failing trial, a true monotone decrease is impossible, and the dip is a direct display of run-to-run variance at $N = 2$, roughly 5 pp, consistent with the replication analysis in Section 6. We therefore read this table as evidence that $N = 2$ is too noisy to recover a retry curve, and draw no conclusion about where returns saturate.

Budget	Pass %	Retries Used	Wall-clock (s)	Tokens
0 (multi-no-retry)	76.7	0.00	41.2	3,341
1	71.7	0.28	53.4	3,375
2 (multi-full)	83.3	0.50	44.8	3,670

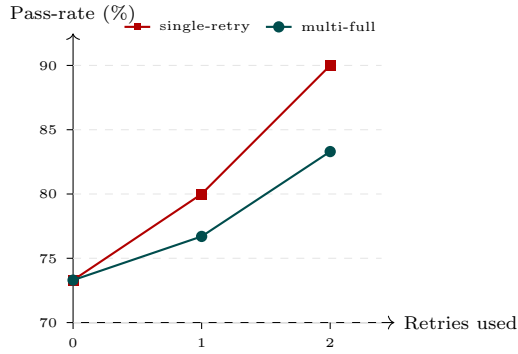


Fig. 5 Cumulative pass-rate as a function of retries consumed within the budget-2 runs: each point is the share of trials that had passed after at most k retries. Both conditions start at 73.3% because the first attempt is unaffected by the retry budget. Neither curve has flattened by the budget limit, so these three points do not establish that a budget of 2 is sufficient; a sweep over $\{0, 1, 2, 3\}$ would be needed to locate the saturation point, and we did not run one. The vertical axis is truncated at 70%.

the extra structure hurt, and why. We examine per-trial JSON execution logs for the two problems where multi-full falls furthest below both single-agent conditions: **min_stack** (60.0% MF versus 100.0% SO and SR) and **unweighted_shortest** (20.0% MF versus 100.0% SO and SR), as shown in Table 4. Both are cases where all three conditions had the same base model available and only the workflow differed.

Inspecting trial traces reveals a structural failure mode:

1. **Longest Increasing Subsequence (lis):** The Planner emits an $O(n^2)$ dynamic programming plan with an incorrect base-case array initialization (`dp = [0] * n` instead of `dp = [1] * n`). The Code Generator strictly adheres to the Planner’s step-by-step instructions, generating code that evaluates single-element arrays to 0 and underestimates subsequence lengths by 1. On subsequent retry loops, the

prompt includes both the original plan and the test execution failure; the Generator attempts localized fixes while preserving the Planner’s flawed initialization invariant, failing all retries. In contrast, Single-Retry synthesizes code directly from the problem description, and upon encountering a test failure, uses the Python traceback error message to correct the array initialization directly.

2. **Min Stack (min_stack):** The Planner instructs the Generator to maintain two parallel stacks (`stack` and `min_stack`) but specifies an incomplete push condition (*"push to min_stack if val > min_stack[-1]"*), omitting the equality case (`val <= min_stack[-1]`). When duplicate minimum values are pushed and popped, `min_stack` becomes desynchronized. The Generator faithfully implements the Planner’s faulty specification. Single-Oneshot and Single-Retry, unconstrained by the Planner’s specified invariant, generate the standard tuple-stack implementation (`val, current_min`) or correct duplicate-handling logic.

These execution traces demonstrate that on 7B-parameter models, a Planner emitting flawed structural constraints acts as a distractor. The downstream Generator treats the Planner’s output as a binding specification, anchoring generation in an incorrect implementation path that execution retry loops fail to rescue.

5.7 External Validity: HumanEval and APPS-Introductory

Table 10 anchors the framework against public benchmarks, reporting all three primary conditions at $N = 1$ trial per problem. HumanEval is run at its full 164-problem scale rather than on a sample, so the paired analysis in Table 11 has enough problems to be informative, this is the only place in the study where the exact tests are not starved of discordant pairs.

Table 10 HumanEval (all 164 problems) and APPS-introductory (50-problem subset) pass-rate.

Benchmark	Cond.	Probs.	Pass %
HumanEval (164)	SO	164	56.1
	SR	164	81.7
	MF	164	80.5
APPS-intro (50)	SO	50	20.0
	SR	50	24.0
	MF	50	22.0

5.8 Is It Feedback, or Just More Compute?

Execution-retry issues more model calls than one-shot, so a gain attributed to *feedback* might instead be a gain from *sampling*: drawing several candidates and keeping any that passes would also raise the pass-rate, and would do so without any feedback at all. This is the control that Olausson et al. [28] show is decisive when repair is charged for the compute it consumes, and it is the one comparison that separates our two candidate explanations. We therefore ran one-shot with five independent samples per

Table 11 Paired comparisons on the full HumanEval suite ($n = 164$ problems). Exact Wilcoxon signed-rank, cluster bootstrap over problems, Holm-adjusted across the three contrasts. Unlike internal-30, this analysis has enough discordant pairs to resolve differences. Both execution-feedback conditions beat one-shot decisively; multi-full and single-retry are indistinguishable. The MF–SR row is the only adequately powered contrast in this paper that is not affected by the test-visibility confound of Section 6.7, because both conditions retry against the same tests.

Comparison	n	Disc.	Δ (pp)	95% CI	Exact p	Holm p
SR – SO	164	42	+25.6	[+18.9, +32.3]	< 0.0001	< 0.0001
MF – SO	164	64	+24.4	[+15.2, +32.9]	< 0.0001	< 0.0001
MF – SR	164	28	−1.2	[−7.3, +4.9]	0.7621	0.7621

problem on the full HumanEval suite, filtering each through the same sandbox and accepting the problem if any sample passed, a best-of-5 baseline at approximately multi-full’s call budget.

Table 12 Sampling control on HumanEval ($n = 164$). Best-of-5 accepts a problem if any of five independent one-shot samples passes, using the same sandbox filter, at five times the call budget of pass@1. The comparison isolates additional compute from execution feedback.

Configuration	LLM calls	Pass %
One-shot, single sample (pass@1)	1.0	57.6
One-shot, best of 5 samples	5.0	57.9
Single-retry (execution feedback)	1.5	81.7
Multi-full	5.3	80.5

Five samples buy +0.4 pp. Execution feedback, at less than a third of that call budget, buys +24.1 pp over the same pass@1 baseline. The explanation is visible in the raw logs: on 161 of 164 problems all five samples agree, because the Generator decodes at temperature 0.0 (Section 4.3). Repeated sampling at that setting reproduces essentially the same program, so there is almost no diversity for a best-of- n filter to exploit. Whatever single-retry is doing, it is not simply buying more lottery tickets.

We are deliberate about the scope of this control. Temperature 0.0 makes best-of-5 a *weak* sampling baseline by construction, and a stronger one would raise the sampling temperature to force diversity, a different generator configuration, and a different experiment. What the control establishes is narrower but sufficient for our purpose: within the decoding configuration used throughout this study, additional compute alone does not reproduce the effect, so the execution-feedback gain reported in Section 6 cannot be dismissed as a sampling artifact. It also carries a practical implication that does not depend on our framework: at deterministic decoding, spending a five-fold compute budget on repeated sampling is close to worthless, while spending half a call on an execution-feedback loop is not.

5.9 Cross-Model Generalization

We evaluate all three primary conditions with a second base model (DeepSeek-Coder-V2-Lite-Instruct, 16B MoE with 2.4B active parameters [40]). Table 13 reports the result on internal-30 ($N = 5$ per problem for Qwen2.5-Coder-7B, $N = 2$ for DeepSeek-V2-Lite, giving 20 evaluations per category for the latter).

Table 13 Cross-model generalization on internal-30 ($N = 5$ per problem for Qwen2.5-Coder-7B, $N = 2$ for DeepSeek-V2-Lite). The two backbones differ in parameter count, routing architecture, and pre-training corpus simultaneously, so this table does not isolate any one of them. With 20 evaluations per DeepSeek category, a single trial flip moves a cell by 5.0 pp.

Model	Cond	DP %	Gr. %	DS %	All %
Qwen2.5-Coder-7B	SO	90.0	60.0	82.0	77.3
	SR	90.0	80.0	100.0	90.0
	MF	88.0	72.0	86.0	82.0
DeepSeek-V2-Lite	SO	90.0	70.0	90.0	83.3
	SR	90.0	70.0	100.0	86.7
	MF	80.0	90.0	80.0	83.3

With DeepSeek-Coder-V2-Lite, graph-category accuracy rises from 70.0% (SO) to 90.0% (MF), a +20.0 percentage point difference, while Dynamic Programming and Data Structures each fall by 10.0 pp (90.0% to 80.0%). The three category shifts cancel: overall accuracy is 83.3% in both conditions. The pattern is therefore a redistribution of where the model succeeds, not a net gain, and it is the clearest category-level effect in the grid.

Two cautions apply. First, the comparison is confounded by design. DeepSeek-Coder-V2-Lite differs from Qwen2.5-Coder-7B in parameter count (16B total vs. 7B dense), routing (MoE with 2.4B active vs. dense), pre-training corpus, and instruction-tuning recipe. Capacity, routing, and data co-vary, so nothing here isolates sparse expert routing from raw scale or training data, and we make no scaling claim. Second, the graph effect rests on ten problems at $N=2$, i.e. twenty evaluations, so a 20.0 pp difference corresponds to four trial flips distributed over at most ten problems. We report no p -value for it because a paired test over ten problems, with trials nested inside problems, would be less informative than the raw counts; readers should treat the direction as suggestive and the magnitude as poorly constrained. Establishing whether multi-agent decomposition genuinely benefits stateful graph problems at larger scale requires a controlled sweep across model sizes within one family, which our compute budget did not permit.

5.10 Error-Class Taxonomy

Table 14 reports the distribution of `error_class` values over the failed trials of each condition on internal-30.

Table 14 Error-class distribution among failed trials (%) on internal-30.

Class	SO	SR	MF
syntax	0.0	0.0	0.0
runtime	41.2	0.0	18.5
timeout	0.0	0.0	0.0
wrong_answer	58.8	100.0	81.5
sandbox_violation	0.0	0.0	0.0
harness_error	0.0	0.0	0.0

5.11 Latest-2026 Supplementary Comparison

Table 15 reports a small-sample evaluation against the May 2026 strongest open-weight code model that fits the reference hardware (Qwen3-Coder-30B-A3B-Instruct, MoE with 3.3B active parameters). The intent is not to replicate the full grid at this scale, compute cost is prohibitive on 6 GB VRAM, but to evaluate whether multi-agent decomposition improves accuracy over one-shot on the May 2026 SOTA local-runnable code model.

Table 15 Supplementary: Qwen3-Coder-30B-A3B on a 10-problem stratified subset of internal-30 (1 trial). Demonstrates that at 30B MoE scale, Multi-Full matches Single-Oneshot accuracy (90.0% vs 90.0%, 0.0 pp gain) while increasing latency 6.2 $\times$ and token cost 12.5 $\times$.

Condition	Pass Rate	Trials	Avg Wall-clock (s)	Avg Tokens
Single-Oneshot	90.0%	10	73	271
Multi-Full	90.0%	10	452	3,376

5.12 Harness Validation and Re-Scoring

Because the acceptance predicate in this framework is a piece of software, it needs its own validation: an error in the sandbox is not a nuisance but a corruption of the dependent variable, and unlike a disagreeing critic it produces no visible symptom. We therefore added a reference-solution control (`tests/unit/test_sandbox.py`) that runs known-correct implementations of representative problems through the sandbox and asserts that they pass. Introducing that control surfaced a defect in the original import policy, which we describe in full here because the correction determines every number reported above and because the failure mode generalizes to any execution-grounded evaluation.

The original sandbox gated `__import__` against a flat allow-list of module names, as described in Section 3.6. Importing an allow-listed pure-Python module transitively imports its private C accelerator, whose root is not itself on the list, so the import raised and the trial was recorded as `error_class = sandbox_violation`. A textbook-correct longest-increasing-subsequence solution using `bisect`, or a heap-based priority queue using `heapq`, was scored as a failure. Across the original grid 135 of 1,335 records

(10.1%) terminated this way, and every one we inspected was a harness artifact rather than an unsafe program.

The defect was not condition-neutral, which is why it distorts comparisons rather than merely depressing them. A condition that emits more candidate programs has more opportunities to trip the guard, so multi-full absorbed 23 spurious failures on HumanEval against one-shot’s four. Any comparison between conditions was therefore partly a measurement of how often each condition invoked the buggy code path.

Because every trial record persists the candidate program it produced, correcting this required no new generation. We repaired the import policy, the environment handling, and the `sys/io` exposure described in Section 3.6, then replayed all 1,178 stored artifacts through the corrected sandbox using the same Evaluator entry point (`scripts/rescore.trials.py`). No model was invoked during re-scoring, so the candidate programs are exactly those the original runs produced and only the scoring changed. Table 16 summarizes the effect. Re-scoring also imposed a single deduplication rule across all runs, keeping the most recent record for each (`problem_id`, `trial`) key; three runs had been resumed with output appended to an existing log, leaving 157 duplicate rows that earlier drafts of this work had handled inconsistently.

Table 16 Effect of the sandbox correction. “Original” is the pass-rate recorded by the defective harness; “corrected” re-scores the same stored programs through the repaired sandbox. No model was re-invoked. The correction is largest where a condition generated the most candidate programs, which is why it is not condition-neutral. [†]The HumanEval rows describe the original 64-problem runs, which is what existed when the defect was found; the full 164-problem runs reported in Section 5 were executed later, under the corrected sandbox.

Benchmark	Condition	Original %	Corrected %	Δ (pp)
Internal-30	SO	73.3	77.3	+4.0
	SR	83.3	90.0	+6.7
	MF	76.7	82.0	+5.3
HumanEval (64) [†]	SO	54.7	56.2	+1.5
	MF	57.8	85.9	+28.1
APPS-intro	SO	6.0	20.0	+14.0
	MF	10.0	22.0	+12.0

Two consequences follow. First, every number in this paper is post-correction. Second, the episode sharpens the paper’s central design argument rather than undermining it. Gating acceptance on execution rather than on an LLM critic’s verdict remains the right choice, but with a qualification we did not anticipate: an execution-based predicate is only as trustworthy as its implementation, and a sandbox defect is silent in a way that a disagreeing critic is not. Execution-grounded evaluation therefore needs a positive control as a matter of routine, and we recommend the reference-solution pattern above for any system whose acceptance criterion is code.

5.13 Test-Source Bias Check

Table 17 reports the test-generation path for every condition across all three benchmarks. Every trial record shows `test_source = deterministic`: all 30 internal-30 problems, all 164 HumanEval problems, and all 50 APPS-introductory problems have deterministic test suites embedded in the harness. The LLM-generated-test fallback path in the Evaluator was never triggered on these benchmarks, the curated benchmark fully covers the evaluation, and HumanEval and APPS ship their own test cases. No accuracy inflation from LLM-generated tests is possible when the test-generation fallback is never invoked.

Table 17 Pass-rate split by test source across all benchmarks and conditions. Every one of the 2,440 executed trial records used deterministic tests; the LLM-generated-test fallback and the smoke-test path were never invoked on any benchmark, so neither contributed to any reported number.

Benchmark	Condition	Test Source	Pass %
Internal-30	Single-Oneshot	deterministic	77.3
	Single-Retry	deterministic	90.0
	Multi-Full	deterministic	82.0
HumanEval	Single-Oneshot	deterministic	56.1
	Single-Retry	deterministic	81.7
	Multi-Full	deterministic	80.5
APPS-intro	Single-Oneshot	deterministic	20.0
	Single-Retry	deterministic	24.0
	Multi-Full	deterministic	22.0

6 Discussion and Limitations

6.1 Statistical Significance: What the Data Do and Do Not Support

Across all internal-30 problems ($n=30$, paired by problem), none of the three pairwise comparisons reaches $\alpha = 0.05$ under the exact test (Table 18). The reason is worth stating plainly, because it governs how the internal-30 numbers should be read (the HumanEval analysis in Section 6 is not affected): most problems are decided identically by all three conditions, so the effective sample size is far below thirty. Single-retry and one-shot differ on only four of thirty problems, and multi-full differs from one-shot on eight. With four discordant pairs the smallest two-sided p the exact Wilcoxon signed-rank test can return is 0.125, which is what we observe. No effect size, however large, could have produced significance at that discordant-pair count. We report exact p -values throughout; the normal approximation used by default in most statistical packages returns 0.083 for this contrast, which would read as marginal significance and is an artifact of applying an asymptotic test to four pairs.

The interval estimates are more informative than the tests. The single-retry advantage over one-shot is +12.7 pp with a cluster-bootstrap 95% interval of [+2.7, +25.3] pp that excludes zero, while multi-full versus one-shot (+4.7 pp, [-6.7, +16.7]) and multi-full versus single-retry (-8.0 pp, [-18.7, +0.7]) both span zero. An interval excluding zero alongside a non-significant exact test is not a contradiction: the bootstrap resamples problems and uses the magnitude of the per-problem differences, whereas the signed-rank test discards magnitude and is limited by the number of non-zero differences. We take the pair of results together as consistent with a real but imprecisely estimated single-retry advantage, and we do not claim it as established. Section 6.7 describes a confound that plausibly inflates it.

Problem-level inversions remain visible and are the reason category averages move at all: `priority_queue` and `scc` favour multi-full, while `avl` favours single-retry. Each such inversion rests on five trials of one problem, so we describe them as observations that motivate the qualitative analysis in Section 5.6 rather than as findings in their own right.

Table 18 Paired comparisons on internal-30 ($n=30$ problems). All three pairwise differences fail to reach $\alpha = 0.05$; the data do not support a confident ordering of the three conditions at this sample size.

Comparison	n	Disc.	Δ (pp)	95% CI	Exact p	Holm p
SR – SO	30	4	+12.7	[+2.7, +25.3]	0.125	0.375
MF – SO	30	8	+4.7	[-6.7, +16.7]	0.547	0.547
MF – SR	30	7	-8.0	[-18.7, +0.7]	0.156	0.375

Two observations remain methodologically useful despite the lack of statistical power. First, the *direction* of the point estimates is internally consistent: configurations that include execution feedback (SR, MF) sit above pure one-shot decoding.

Second, the per-agent ablation (Table 8) does more than measure components: it supplies a negative control that the rest of the study lacks. Three of the interventions cannot alter acceptance by construction. The Reviewer’s output is never read by any downstream agent, the Optimizer reverts on re-validation failure, and retrieval feeds only the Planner. Run at $N = 5$, they are therefore not ablations at all but independent replications of multi-full, and their spread is a direct estimate of run-to-run variance.

All three returned exactly 78.7% (118 of 150 trials). They are genuine replications rather than deterministic repeats: per-trial outcomes differ between them, and the generated programs differ on 41 and 6 of 150 trials respectively. They simply agree in aggregate. Same-session variance in the aggregate pass-rate is therefore close to zero, which is a stronger statement about measurement stability than we could otherwise have made. The original multi-full run sits 3.3 pp higher at 82.0%, and shares byte-identical generated code with the new runs on only 56 of 150 trials, so cross-session drift in the model server exceeds within-session noise by a wide margin. Any comparison in this paper that draws on runs executed months apart inherits that drift; comparisons within a session do not.

Against that yardstick, removing the Planner raises the pass-rate to 86.7%, which is +8.0 pp above the three same-session replications. That is far outside the replication spread, so the planner-distractor effect is more robust than the earlier two-trial ablations suggested. The per-problem paired test tells a more restrained story: only 6 of 30 problems are discordant against the pooled replication baseline, giving +7.2 pp with a bootstrap interval of $[-0.7, +16.3]$ and exact $p = 0.156$. We report both because they answer different questions. The aggregate comparison asks whether the effect exceeds measurement noise, and it clearly does. The paired test asks whether the effect is consistent across problems, and at six discordant problems it cannot say. Section 5.6 describes the mechanism we believe is responsible, and we still present it as a hypothesis about 7B-scale planning rather than a general result.

6.2 External Validity and the Scaling Hypothesis

The external benchmarks are where the design finally has enough statistical power, and they change what this paper can claim. On the full 164-problem HumanEval suite (Table 10), one-shot reaches 56.1%, single-retry 81.7%, and multi-full 80.5%. With $n=164$ paired problems the exact tests are no longer starved of discordant pairs (Table 11): both execution-feedback conditions beat one-shot decisively (SR–SO +25.6 pp, 95% CI $[+18.9, +32.3]$, exact $p < 0.0001$; MF–SO +24.4 pp, CI $[+15.2, +32.9]$, $p < 0.0001$; both survive Holm adjustment), while multi-full and single-retry are statistically indistinguishable (MF–SR −1.2 pp, CI $[-7.3, +4.9]$, $p=0.76$). APPS-introductory orders the same way (SO 20.0%, MF 22.0%, SR 24.0%) at a difficulty level where all three conditions are close to the base model’s ceiling.

The MF–SR contrast deserves emphasis because it is the one comparison in this paper that is both adequately powered and free of the confound described in Section 6.7. Single-retry and multi-full both consume stderr from the scoring tests, so whatever advantage that asymmetry confers, it is conferred equally on both. The difference between them is therefore attributable to the multi-agent decomposition itself, and it is −1.2 pp with a confidence interval that excludes anything larger than about five points in either direction. At 3.6 times the LLM calls and 7.3 times the tokens of the retry baseline (Table 7), the decomposition is not paying for itself on these tasks.

The SR–SO and MF–SO gaps should be read more cautiously. Both are large, both are significant, and both are inflated to an unknown degree by the fact that the retry conditions iterate against the function that grades them while one-shot gets a single attempt. We report them because the direction and the ordering are informative, not because +25 pp is an unbiased estimate of what execution feedback buys. A held-out test split would settle it; our benchmarks carry too few tests per problem to construct one without making the scoring signal unreliable.

The clearest category-level effect in the grid is the cross-model condition (Table 13). With DeepSeek-Coder-V2-Lite (16B MoE, 2.4B active) as the backbone, graph-category accuracy under multi-full reaches 90.0% versus 70.0% under one-shot, a +20.0 pp difference, while DP and DS each fall by 10.0 pp so that overall accuracy is identical at 83.3%. Multi-agent decomposition here changes *which* problems the system solves without changing how many. Because DeepSeek-Coder-V2-Lite differs from

Qwen2.5-Coder-7B in capacity, routing, and training corpus simultaneously, the comparison cannot attribute the shift to any one of them, and we make no scaling claim. The effect also rests on ten problems at $N=4$, so its magnitude is poorly constrained. On 7B dense models the practical reading is unchanged: multi-agent routing costs roughly $5\times$ the LLM calls and an order of magnitude more tokens for accuracy that our data cannot separate from single-agent retry.

6.3 Practical Recommendation

For deployments at 7B parameter scale on canonical algorithmic problems, the single-retry configuration is the most cost-efficient path: achieving higher or equal accuracy at lower wall-clock latency and token cost compared to multi-full. Raising base-model capability does not change that conclusion: under DeepSeek-Coder-V2-Lite multi-full reaches 83.3%, exactly matching one-shot and falling below single-retry’s 86.7%. The case for the full configuration is therefore not accuracy but the artifacts it produces, the Reviewer’s invariant list and the Explainer’s rationale, which have value for software governance independently of whether they improve correctness, and the category-level redistribution it produces on graph problems, which may matter where that category dominates the workload. Table 19 summarizes deployment guidelines based on empirical metrics.

Table 19 Deployment guidance. Pass rates are from the full 164-problem HumanEval suite, the only benchmark in this study with enough paired problems to separate conditions; cost columns are per-problem means on internal-30 (Table 7), which is where cost was instrumented. The ordering is *not* an accuracy ranking: single-retry and multi-full are statistically indistinguishable on HumanEval (Table 11), so the case for single-retry rests on cost, and the case for multi-full rests on the artifacts it produces.

Configuration	Pass Rate	Wall-clock	Tokens	LLM Calls	Recommended Use Case
Single-Oneshot (SO)	56.1%	7.6 s	313	1.0	Latency-critical completion, where a wrong answer is cheap to discard and a round-trip is not affordable.
Single-Retry (SR)	81.7%	13.3 s	507	1.5	Default choice. Statistically indistinguishable from multi-full ($p = 0.76$) at $0.28\times$ its model calls.
Multi-Full (MF)	80.5%	59.8 s	3,710	5.3	Choose only when the auxiliary artifacts (reviewer invariants, explanations) are themselves required, not for accuracy.

6.4 The Reviewer’s Role and the “Advisory-Only” Choice

A static reviewer that gates acceptance can block correct code, a failure mode we observed during early development and which is documented in the repository’s commit history. Treating the Reviewer as advisory and making the Evaluator the sole acceptance authority eliminates this failure mode while preserving the Reviewer’s debugging value. The structured-output schema introduced in Section 3.4 makes the

advisory output usable by downstream tooling. In prior benchmark evaluation by co-authors, the JudgeSense benchmark [19] provides empirical support: across thirteen judge models, decisions flip on 8.5–61.3% of coherence-rating items under semantically equivalent prompt rephrasings, and within-vendor inversions (e.g. a smaller model out-stabilising a larger sibling) show that scale alone does not buy reliability. We do not run a paraphrase-stability evaluation of our Reviewer in this paper, as it would require its own paraphrase set and is orthogonal to the framework’s headline claim; however, reading our design through the JudgeSense lens, the choice to gate acceptance on execution outcomes rather than on a paraphrase-sensitive critic verdict is conservatively the right one. Quantifying the Reviewer’s Judge Sensitivity Score (JSS) on a code-review-specific paraphrase set is a natural follow-up direction and is listed in Section 7.

6.5 When Retrieval Helps

The RAG ablation (Table 8, row **rag-on**) measures the marginal effect of seeding the Planner with retrieved algorithmic-technique snippets. Evaluated at $N=5$ trials per problem, enabling retrieval returns DP 90.0%, Graph 64.0%, DS 82.0%, aggregate 78.7%, identical in every cell to the **–Reviewer** and **–Optimizer** runs, and 3.3pp below the multi-full baseline. Since retrieval feeds only the Planner and cannot alter the acceptance predicate, this row is best read as a third replication of multi-full rather than as a measurement of retrieval: it reproduces the other two inert interventions exactly, and its distance from the baseline is the cross-session drift discussed above rather than an effect of retrieval. We verified that the three runs are distinct executions with their own trial logs rather than duplicated output. Two structural factors plausibly explain the null: (i) the small scale of our seeded vector store (15 core algorithmic technique templates in ChromaDB), which provided only basic algorithmic patterns, and (ii) parametric knowledge saturation, as 7B code models already possess strong internal representations of standard algorithms (e.g., BFS, Dijkstra, LRU Cache), rendering high-level retrieved snippets redundant. A domain-dense vector store containing codebase-specific APIs or complex enterprise domain rules, rather than textbook algorithms, is likely required for retrieval to yield measurable gains.

6.6 Cost-Correctness Trade-off

Multi-agent workflows are not free. Table 7 reports the wall-clock and token-count overhead per problem. The framework consumes $5.3\times$ the LLM calls of the single-oneshot baseline at retry budget 2, and $11.9\times$ the tokens. Token cost therefore grows *super*-linearly in the number of calls: each agent invocation carries accumulated plan, review, and evaluation context, so later calls in an episode are substantially longer than the single call a one-shot run issues. For batch backend-code-synthesis use cases this is an acceptable trade. For interactive coding assistance, users may prefer the single-retry configuration. The wall-clock overhead reported in Table 7 reflects local execution on a single 6 GB laptop GPU (RTX 4050), where inference latency is bound by sequential local LLM generation. In multi-GPU serving environments or cloud API deployments with parallel worker queues, non-dependent multi-agent tasks can execute

concurrently across graph nodes, significantly reducing wall-clock latency compared to single-GPU sequential execution.

6.7 Limitations

Validity of the scoring signal.

The retry loop optimizes against the scoring tests. This is the most serious threat to validity in the paper and it affects the central comparison. A trial is scored by executing a deterministic test suite, and the same suite produces the stderr that the Generator consumes when regenerating. Single-retry and multi-full therefore iterate against the function that grades them, while one-shot receives one attempt with no such feedback. Any gap between the retry conditions and one-shot conflates two effects: the genuine value of execution feedback, and the advantage of being allowed to fit the evaluation. We expect this inflates the observed single-retry and multi-full advantage, and the effect should scale with how many problems fail on the first attempt, which is consistent with the pattern we see (the HumanEval multi-full-versus-one-shot gap of +24.4pp is far larger than the internal-30 gap of +4.7pp for the same contrast, and HumanEval has by far the higher first-attempt failure rate). The clean design splits each problem’s tests into a feedback set and a held-out scoring set, and reports retries against the former only. Our benchmarks carry too few tests per problem (a median of four on internal-30) to split without making the scoring signal unreliable, so we report the confound rather than a corrected number. Readers should treat every retry-versus-one-shot comparison here as an upper bound on the benefit of execution feedback.

What the design can and cannot resolve.

Sampling is controlled for, but only at deterministic decoding. Our conditions differ in model calls as well as in mechanism, so we ran the sampling control directly (Section 5.8): five independent one-shot samples per problem, same sandbox filter, at roughly multi-full’s call budget. It returns +0.4pp against +24.1pp for execution feedback, so additional compute does not explain the effect. The caveat is that the Generator decodes at temperature 0.0, which makes repeated sampling nearly degenerate, 161 of 164 problems return five identical outcomes. A temperature-annealed best-of- n baseline would be a harder test, and we have not run one. We therefore claim only that sampling fails to explain the gain *at the decoding configuration used in this study*, not that feedback dominates sampling in general.

Scope of the evidence.

Dataset scope and contamination. The internal-30 suite uses textbook problems likely present in pretraining corpora, and 27 of the 30 were authored by us, so the suite reflects our expectations about what these models find difficult. Contamination-controlled benchmarks such as LiveCodeBench [43] would be the appropriate external anchor and we have not run them. We do not claim contamination-free evaluation.

Effective sample size. Twenty-one of the thirty internal-30 problems return identical outcomes under all three conditions: nineteen are solved by all of them, and

two (`matrix_chain`, `grid_shortest_path`) are solved by none. Nine problems therefore carry all of the discriminating signal, and $n=30$ overstates the resolution of the design; this is the direct cause of the low discordant-pair counts that make the exact tests uninformative. It is also a lesson from the correction described in Section 5.12: before that fix, `dijkstra` failed in every condition and every trial and looked like a genuine capability ceiling, when in fact the harness was rejecting its `heapq` import. A benchmark that is both larger and harder, with a reference-solution control on every problem, is needed before differences of a few percentage points can be resolved.

Where the recommendation is and is not supported. Single-retry is now evaluated on all three benchmarks and on both backbones, so the recommendation in Table 19 no longer rests on internal-30 alone. It leads on HumanEval (81.7% vs. 80.5% multi-full), on APPS-introductory (24.0% vs. 22.0%), on internal-30 (90.0% vs. 82.0%), and under DeepSeek-Coder-V2-Lite (86.7% vs. 83.3%). Only the HumanEval comparison is adequately powered, and there the honest reading is that single-retry and multi-full are equivalent rather than that single-retry wins: the difference is -1.2 pp with a confidence interval spanning $[-7.3, +4.9]$. The practical case for single-retry rests on cost, not accuracy. It attains the same correctness for roughly one quarter of the LLM calls (1.5 versus 5.3) and one seventh of the tokens.

Generality.

Single-language scope. Both the sandbox and the deterministic-test layer are Python-only. Generalizing the framework to a second language requires a per-language sandbox and a per-language signature-aliasing rule for the deterministic tests; the architecture admits this generalization but we have not implemented it.

Inference-backend dependence. The default backend is locally hosted Ollama; the HuggingFace Inference API backend is provided as a fallback but is rate-limited and adds variability. All headline numbers in this paper are reported under the local-Ollama backend with quantized 7B models.

Guarantees the system does not provide.

Sandbox isolation. The subprocess-isolated sandbox restricts builtins and imports, sanitizes the child environment, and serves restricted `sys/io` shims, but it is not a kernel-level container (Docker, firejail, seccomp) and should not be treated as one. Isolation rests on an allow-list and a deny-list inside a normal CPython process, which is adequate for running code that a model produced while trying to solve a stated problem, and not adequate against code written to escape. The 1 GB address-space ceiling is applied through `resource.setrlimit` and is therefore inactive on Windows, the platform used for our measurements. Production deployment in adversarial settings requires OS-level isolation.

Passing the suite is weaker than being correct. Two distinct issues arise here. The LLM-generated-test fallback is a documented capability of the Evaluator, but Table 17 shows it was never triggered: all 2,440 trials on all three benchmarks used deterministic tests. That path and the smoke-test path are therefore parts of the design that this paper does not evaluate, and we make no claim about them. Separately, passing a deterministic suite is weaker evidence of correctness than it appears; the suites are

small, and work on test adequacy for generated code [44] shows that programs passing the reference tests frequently violate untested invariants. Our pass rates should be read as “passes these tests,” not as “correct.”

Single static-review schema. The Reviewer’s structured output is a fixed schema. Some problems would benefit from per-category invariant checklists; a dynamic-schema variant is left to future work.

Model scope. Headline numbers are produced with one base model family (Qwen2.5-7B). Cross-model results with DeepSeek-Coder-V2-Lite (Section 5) substantially mitigate this concern but do not eliminate it; results on substantially larger models (e.g. 70B-class) are not within our compute budget.

Well-specified problem descriptions. Every problem in the experimental suite has an unambiguous, precisely stated specification. The framework’s behavior on under-specified or ambiguous inputs, where the correct algorithm is not uniquely implied by the problem statement, has not been characterised. In such settings the Planner may select a plausible but incorrect strategy, and the Evaluator’s test suite may be unable to distinguish it from a correct one; this is an important direction for future evaluation.

No formal correctness guarantees. Execution-grounded validation is empirical, not formal. Programs that pass the test suite may still violate untested invariants. We discuss formal-verification integration in Section 7 as future work.

7 Conclusion and Future Work

We presented *ExecuGraph*, a graph-structured workflow framework for empirically isolating execution feedback, multi-agent role decomposition, and retrieval in LLM code synthesis. By formalizing the workflow as a typed transition system that collapses into single-agent one-shot and single-agent execution-retry baselines inside a unified runtime, *ExecuGraph* provides an explicit diagnostic harness for multi-agent systems. On the full 164-problem HumanEval suite, where the design is adequately powered, execution feedback delivers a decisive improvement over one-shot decoding (+25.6 pp for single-retry, exact $p < 0.0001$) while multi-agent decomposition adds nothing measurable on top of it (−1.2 pp, 95% CI [−7.3, +4.9], $p = 0.76$) for roughly five times the LLM calls. We are deliberate about which of those two statements is confounded: the retry loop consumes stderr from the tests that also score the trial, which inflates any comparison against one-shot but leaves the multi-full versus single-retry contrast untouched, because both conditions retry. The negative result is therefore the more secure of the two. Under cross-model evaluation with DeepSeek-Coder-V2-Lite (16B MoE), multi-agent decomposition raises graph-category accuracy to 90.0% (+20.0 pp over one-shot) while leaving overall accuracy unchanged, redistributing rather than increasing the set of solved problems.

Two results carry more weight than the headline comparison. The ablation confirms empirically what the architecture implies for the Reviewer and the Optimizer, neither can affect correctness, and neither does, and the retrieval layer behaves the same way, which suggests that reported costs of “multi-agent” systems should separate correctness machinery from artifact generation. And an audit of our own harness found that the execution sandbox rejected allow-listed modules through their C accelerators,

converting correct programs into failures at condition-dependent rates; correcting it changed every headline number, in one case by 28.1 pp. We report the defect, the correction, and the re-scoring tool because the episode illustrates a general hazard for execution-grounded evaluation: when the acceptance predicate is a piece of software, a bug in it is indistinguishable from a result.

Future work falls into five directions. First, we plan to integrate lightweight formal-verification probes (e.g. property-based tests via Hypothesis, type-state checks) as additional decision-predicate inputs alongside execution. Second, we plan to extend the sandbox and signature-aliasing layers to support non-Python languages, beginning with TypeScript and Java. Third, we plan to evaluate domain-specific parameter-efficient fine-tuning of the Generator agent (low-rank adaptation, LoRA) inside the framework, to test whether task-specific specialization compounds with the workflow’s gains. Fourth, and most importantly, we plan to add a cost-matched sampling control: drawing k independent one-shot candidates and filtering them through the same sandbox, scored under a token-budget-aware metric such as *pass@t* [28], would establish whether execution feedback beats sampling at equal compute or merely at higher compute. We also plan to evaluate on adversarial-style benchmarks (HumanEval+, MBPP+) where weakly tested code is exposed, and on contamination-controlled suites [43]; the supplementary 30B condition reported in Section 5 should be extended to a full grid at that scale once compute permits. Fifth, motivated by the JudgeSense findings on judge instability under paraphrase [19], we plan to construct a code-review-specific paraphrase set and report the Reviewer’s Judge Sensitivity Score; this would convert the present qualitative argument for the advisory-only design into a quantitative one.

8 Reproducibility

The code repository (<https://github.com/rohithreddybc/execuGraph>, commit 1b3ae4f) accompanying this paper is structured so that every result table can be regenerated from a single command. We summarize the key reproducibility surface here; full step-by-step instructions are in `REPRODUCIBILITY.md` in the repository.

Software environment.

Python 3.11, dependencies pinned in `pyproject.toml`. A `Dockerfile` is provided that builds a container with all dependencies including a CUDA-enabled PyTorch wheel and a pre-pulled Ollama model set. The container is independent of the user’s local Ollama installation.

Models.

Local Ollama, with the following model tags recorded in `REPRODUCIBILITY.md`: `qwen2.5-coder:7b-instruct-q4_K_M`, `deepseek-coder-v2:16b-lite-instruct-q4_K_M`, and `qwen3-coder:30b-a3b-q4_K_M`.

Seeds.

Seeds $\{0, \dots, N - 1\}$ are passed through to the LLM backend (where supported by the model server), to the LLM-test sampler, and to the trial-runner’s iteration order.

Determinism caveats.

GPU floating-point non-determinism in the Ollama runtime means bit-exact reproducibility is not guaranteed across hardware; we measured run-to-run variation directly rather than assuming it: three architecturally inert configurations run in the same session returned identical aggregate pass-rates, while the same configuration executed months earlier differed by 3.3pp and shared byte-identical generated code on only 56 of 150 trials. Within-session aggregate variation is therefore negligible and cross-session drift is not, which is why comparisons in this paper are drawn within a session wherever possible.

Commands.

The full result set is regenerated by:

```
# 1. environment + models
docker build -t execugraph .
docker run --gpus all -v $(pwd)/results:/app/results execugraph \
    bash scripts/run_full_grid.sh
# 2. table generation
python scripts/build_all_tables.py \
    results/<run_id> --out paper/tables
# 3. compile
cd paper && latexmk -pdf main.tex
```

Per-trial artifacts.

Every trial appends one JSON line to `results/<run_id>/trials.jsonl` with the fields enumerated in Section 4.6. A small example artifact is committed under `results/example_run/` so that the table-generation pipeline can be exercised without running the LLM.

Acknowledgements

The authors thank the maintainers of LangGraph, LangChain, ChromaDB, and Ollama for the open-source infrastructure that made this work practical, and colleagues who commented on earlier drafts.

Declarations

Funding.

The authors received no specific funding for this work. All experiments were run on commodity consumer hardware using open-weight models, at zero marginal API cost.

Competing interests.

The authors declare no competing financial interests. One reference cited in this work ([19]) is co-authored by one of the present authors; it is cited for its empirical findings on judge stability and no author of this manuscript was involved in its peer review.

Ethics approval and consent to participate.

Not applicable. This study involves no human participants, human data, or animal subjects.

Consent for publication.

Not applicable.

Data availability.

All per-trial JSON logs, configuration files, model digests, seeds, the benchmark definition, the re-scoring tool, and the table generators required to reproduce every number in this paper are released in the accompanying repository, <https://github.com/rohithreddybc/execuGraph>, pinned at commit 1b3ae4f, which is the exact code state that produced every result reported here. The curated 30-problem benchmark is additionally released as a standalone dataset. No proprietary or restricted data were used; all models are open-weight and were executed locally, so the full grid is reproducible at zero API cost.

Materials availability.

Not applicable.

Code availability.

The framework is released under the Apache-2.0 licence at the repository above, together with a `Dockerfile` and `REPRODUCIBILITY.md` giving step-by-step reproduction instructions.

Author contributions.

L.S.D. conceived the study and contributed conceptualization, methodology, and writing (original draft). R.R.B. contributed software, data curation, formal analysis, investigation, validation, visualization, and writing (original draft). M.S. contributed methodology, investigation, interpretation of results, and writing (review and editing). L.S.D., R.R.B., and M.S. contributed equally to this work. A.J.P. contributed supervision, project administration, and writing (review and editing). All authors read and approved the final manuscript.

Use of AI-assisted technologies.

LLMs were only used for copyediting the work.

References

- [1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2020.
- [2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- [3] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with APPS. In *Proc. Int. Conf. Learn. Representations (ICLR)*, 2021.
- [4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of EMNLP*, 2020.
- [5] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.
- [6] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2022.
- [7] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V. Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain-of-thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.
- [8] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. OpenReview, 2018.
- [9] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. PAL: Program-aided language models. *arXiv preprint arXiv:2211.10435*, 2022.
- [10] Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*, 2023.
- [11] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d’Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code

- generation with AlphaCode. *Science*, 2022.
- [12] Aakanksha Chowdhery et al. PaLM: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.
 - [13] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. CodeGen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.
 - [14] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. InCoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.
 - [15] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
 - [16] Deepshikha Bhati, Fnu Neha, D. S. Bandaru, M. Weber, and I. D. Gajera. Mapping the LLM landscape: A cross-family survey of architectures, alignment methods, and benchmark performance. *AI*, 7(4):142, 2026.
 - [17] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegrefe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2023.
 - [18] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.
 - [19] Rohith Reddy Bellibatlu, Edward Raff, and Wenbin Zhang. JudgeSense: A benchmark for prompt sensitivity in LLM-as-a-Judge systems. *arXiv preprint arXiv:2604.23478*, 2026.
 - [20] Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. AgentCoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.
 - [21] Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. MapCoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*, 2024.
 - [22] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.
 - [23] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryan W. White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.
 - [24] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *Proc. Int. Conf. Learn. Representations (ICLR)*, 2023.

- [25] Toran Bruce Richards et al. Auto-GPT: An autonomous GPT-4 experiment. <https://github.com/Significant-Gravitas/Auto-GPT>, 2023.
- [26] Mert Cemri, Melissa Z. Pan, Shuyi Yang, Lakshya A. Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei Zaharia, Joseph E. Gonzalez, and Ion Stoica. Why do multi-agent LLM systems fail? In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS), Datasets and Benchmarks Track*, 2025.
- [27] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. *arXiv preprint arXiv:2308.03688*, 2023.
- [28] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. Is self-repair a silver bullet for code generation? In *Proc. Int. Conf. Learn. Representations (ICLR)*, 2024.
- [29] Sayash Kapoor, Benedikt Stroebl, Zachary S. Siegel, Nitya Nadgir, and Arvind Narayanan. AI agents that matter. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2025.
- [30] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. *arXiv preprint arXiv:2302.04761*, 2023.
- [31] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2020.
- [32] Chroma Authors. Chroma: The AI-native open-source embedding database. <https://docs.trychroma.com/>, 2024. Open-source software.
- [33] LangChain Authors. LangChain: Building applications with LLMs through composability. <https://docs.langchain.com/>, 2024. Open-source software.
- [34] LangGraph Authors. LangGraph: A library for building stateful, multi-actor applications with LLMs. <https://github.com/langchain-ai/langgraph>, 2024. Open-source software.
- [35] Ollama Authors. Ollama: Run large language models locally. <https://ollama.com/>, 2024. Open-source software.
- [36] Fnu Neha and Deepshikha Bhati. Traditional RAG vs. agentic RAG: A comparative study of retrieval-augmented systems. *TechRxiv preprint*, 2025.
- [37] Aditi Singh, Abul Ehtesham, Saket Kumar, Tala Talaei Khoei, and Athanasios V. Vasilakos. Agentic retrieval-augmented generation: A survey on agentic RAG. *arXiv preprint arXiv:2501.09136*, 2025.
- [38] Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. RepoCoder: Repository-level code completion through iterative retrieval and generation. In *Proc. Conf. Empirical Methods in Natural Language Processing (EMNLP)*, 2023.
- [39] Binyuan Hui, Jian Yang, Zeyu Cui, Jiayi Yang, Dayiheng Liu, Lei Zhang, Tianyu

- Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-Coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.
- [40] DeepSeek-AI. DeepSeek-Coder-V2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*, 2024.
 - [41] OpenAI. gpt-oss: An open-weight language model from OpenAI. Apache-2.0 weights at <https://huggingface.co/openai/gpt-oss-20b>, 2025.
 - [42] Qwen Team. Qwen3-Coder: Agentic code generation with mixture-of-experts. Technical report; weights at <https://huggingface.co/Qwen/Qwen3-Coder-30B-A3B-Instruct>, 2025.
 - [43] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *Proc. Int. Conf. Learn. Representations (ICLR)*, 2025.
 - [44] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Proc. Adv. Neural Inf. Process. Syst. (NeurIPS)*, 2023.