REQWEAVER: STAGE-ALIGNED REQUIREMENT MANAGEMENT FOR GREENFIELD REPOSITORY GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Building a repository from scratch relies on a requirements document, but the requirements that need to be considered together change as development proceeds. Full-document input leaves agents to identify these combinations for each task, while reading on demand requires them to recognize missing information. We introduce ReqWeaver, which freezes exact source-to-stage bindings before coding and delivers the assigned passages at stage entry without a retrieval query. Completed-stage packets are retired and selected past outputs compacted; shared requirements can reappear in later packets. Optional contract compilation organizes dispersed requirements and records supplementary design choices. We evaluate ReqWeaver with three stochastic generation runs on 36 NL2Repo-Bench and 30 RepoGenesis tasks. With GPT-5.4-mini and OpenHands, ReqWeaver reaches 46.47% versus 37.20% NL2Repo-Bench Pass (+9.28 percentage points; 95% confidence interval (CI), +0.79 to +17.83) and 78.45% versus 55.03% RepoGenesis Pass (+23.42 percentage points; 95% CI, +2.22 to +44.18). Using the third ReqWeaver run block in the main evaluation and controls with the same frozen plans, evidence binding and Projection jointly reduce total tokens, including planning, by 10.56% relative to disabling both. Code is available at `https://anonymous.4open.science/r/ReqWeaver-72D4/`.

1 INTRODUCTION

Large language model (LLM)-based coding agents can plan, edit, test, and repair code (Wang et al., 2025). In existing repositories, code dependencies and execution feedback guide changes (Bairi et al., 2024; Wang et al., 2025). Greenfield development, or building a repository from scratch, lacks these implementation anchors. A requirements document becomes the primary guide to the system's behavior and design (ISO/IEC/IEEE, 2018; Ding et al., 2025; Peng et al., 2026). Effective use of this document is central to the task.

The role of that document changes throughout development. Planning needs system boundaries and shared constraints, module implementation needs detailed behavior and interfaces, and integration needs cross-component consistency (Washizaki, 2026). The same requirement can guide several stages, each time alongside different supporting details. Requirements presented separately in the document may therefore need to be considered together for a particular decision. The full document remains the reference, but the requirements emphasized in the active context should change with the development task.

Structured planning already organizes implementation constraints: PaperCoder derives file-level analyses, and RPG represents functionality, interfaces, and dependencies (Seo et al., 2026; Luo et al., 2025). We focus on the accompanying source evidence: which exact requirement passages should enter a stage's context, and when should they leave or reappear? Full-document access leaves this selection to the coding agent, while queries can omit requirements whose relevance it has not recognized.

Consider a scheduled-task service whose recovery requirement appears separately from its startup instructions (Figure 1). An agent connects the store and starts the Hypertext Transfer Protocol (HTTP) service but omits restoring pending-job timers. A job created before a restart therefore

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

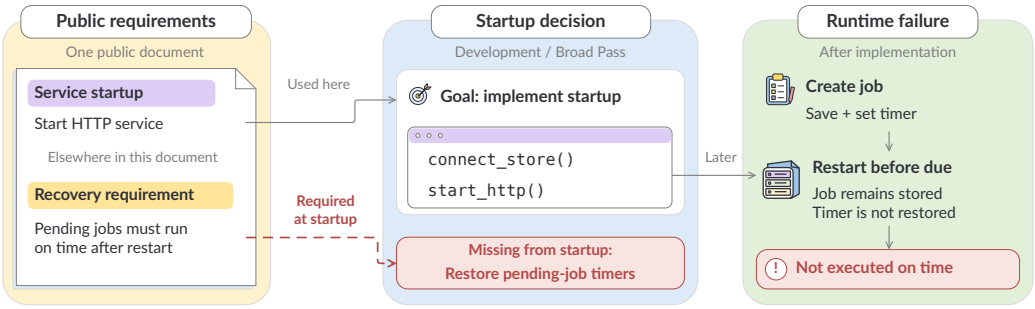

Figure 1: **Available requirements, missed prerequisites.** A recovery requirement elsewhere in the requirements document must inform startup design. Without restoring pending-job timers, a job remains stored after restart but misses its scheduled execution. The red dashed arrow links the recovery requirement to the missing startup step in the red box. Illustrative scenario; calls are pseudocode.

remains in storage yet misses its scheduled execution. A search limited to startup instructions can miss this recovery dependency. The recovery requirement should guide both timer restoration during startup design and checks of persistence, restart, and scheduled execution during validation.

We introduce **ReqWeaver**, which makes source-to-stage bindings an explicit part of the execution plan. Before coding, its Scheduler proposes stage goals and dependencies together with exact source references, then freezes the validated plan. The controller delivers assigned passages at stage entry without waiting for a retrieval query, retires completed-stage packets and selected past outputs, and delivers reused evidence in later packets. Optional contract compilation organizes dispersed requirements and records supplementary design choices. The original document remains available for further reading.

We evaluate 36 NL2Repo-Bench and 30 RepoGenesis tasks over three stochastic generation runs per task and configuration, reporting each benchmark separately. With GPT-5.4-mini in the OpenHands harness, ReqWeaver achieves a macro-averaged repository test pass rate (Pass) of 46.47% versus 37.20% for the OpenHands baseline on NL2Repo-Bench ($+9.28$ percentage points; 95% confidence interval (CI), $+0.79$ to $+17.83$). On RepoGenesis, Pass reaches 78.45% versus 55.03% for the same baseline ($+23.42$ percentage points; 95% CI, $+2.22$ to $+44.18$). To examine requirement use, we compare requirement-access strategies, upfront versus staged delivery of the same content, and scheduled versus shuffled stage assignments. The end-to-end results demonstrate ReqWeaver's effectiveness across both benchmarks, while these controls examine how requirement selection and delivery contribute to repository generation.

Our contributions are:

- ReqWeaver, an execution protocol combining frozen source-to-stage bindings, exact stage-entry delivery, and retirement and reuse of requirement evidence;
- evidence of higher mean Pass with GPT-5.4-mini/OpenHands on two benchmarks, complemented by controls that distinguish requirement access, delivery timing, and stage assignment; and
- token accounting for the third ReqWeaver run block included in the main evaluation: binding and Projection jointly reduce total tokens, including planning, by 10.56% relative to controls with the same frozen plans.

2 Related Work

Greenfield repository generation and structured planning. Coding-agent harnesses provide execution tools (Wang et al., 2025; Zhang et al., 2024), while CodeS, ZeroRepo, and PaperCoder (from Paper2Code) organize generation through multi-layer sketches, a Repository Planning Graph (RPG), and modular implementation plans, respectively (Zan et al., 2024; Luo et al., 2025; Seo et al., 2026). NL2Repo-Bench, RepoGenesis, Commit0, and RepoZero evaluate repository construction from re-

quirements or application programming interface (API) specifications (Ding et al., 2025; Peng et al., 2026; Zhao et al., 2024; Zhang et al., 2026). ReqWeaver complements these planning approaches by selecting which requirement passages reach each stage.

Repository retrieval and requirement delivery. CodePlan propagates required edits through dependencies; RepoCoder, DraCo, GraphCoder, and Repoformer retrieve context using queries, structure, or repository state (Bairi et al., 2024; Zhang et al., 2023; Cheng et al., 2024; Liu et al., 2024b; Wu et al., 2024). For requirement delivery, we consider the risk that queries omit requirements whose relevance the agent has not recognized. Repository context files also do not consistently improve task success (Gloaguen et al., 2026). ReqWeaver fixes source-to-stage assignments before coding and delivers exact passages without query-time ranking; ordinary retrieval remains available.

Closest requirement-grounded systems. SWE-RPG diagnoses implicit-requirement recovery in existing-repository issue resolution (Zhou et al., 2026). Context-Augmented Code Generation jointly changes retrieval, specification, and workflow in a small existing-repository study (Dillon & Varanasi, 2026). A closely related concurrent system, CodeSpec, compiles repository-grounded functional chains into executable architecture and behavior specifications (Wang et al., 2026). CodeSpec also evaluates greenfield generation on NL2Repo-Bench: it first creates a repository skeleton, then uses the evolving codebase to construct functional chains and compile specifications. ReqWeaver instead assigns source passages to stages before implementation and delivers them when each stage begins. Its optional compilation organizes requirements and additional design decisions into a structured specification, drawing on requirements traceability, structured requirements, Design by Contract, and architectural decision records (Gotel & Finkelstein, 1994; Mavin et al., 2009; Meyer, 1992; Tyree & Akerman, 2005).

These approaches address different parts of requirement use: SWE-RPG measures requirement recovery, CodeSpec checks design–implementation consistency, and ReqWeaver delivers preassigned evidence before a stage acts. ReqWeaver guarantees delivery of the assigned evidence, not its correct use or functional correctness; this mechanism comparison does not establish empirical superiority over CodeSpec.

Long-horizon context management. Observation masking, trajectory reduction, hierarchical memory, and prompt compression manage long interaction histories (Lindenbauer et al., 2025; Xiao et al., 2026; Hu et al., 2025; Jiang et al., 2024). In question answering and key-value retrieval, models use long contexts unevenly (Liu et al., 2024a). Context Projection combines observation masking with removal of completed-stage messages: it replaces selected past tool outputs with placeholders that retain record identifiers and references, while preserving failure outputs and raw records.

3 REQWEAVER

ReqWeaver makes requirement delivery part of an execution plan. Before coding, the Scheduler defines development stages and assigns source passages to them. The controller then delivers each packet at stage entry, and Context Projection removes or compacts completed-stage context. Optional Contract Compilation makes structured requirements and supplementary design decisions available for scheduling.

3.1 PROBLEM FORMULATION

Given public inputs $D = \{d_j\}_{j=1}^{J}$, a coding agent transforms a workspace without implementation code W_0 into W_K through a dependency-ordered stage sequence $S = (s_1, \ldots, s_K)$: Plan, Broad Pass, one or more Module Deepening stages, and Integration. Broad Pass builds an initial runnable implementation, while Module Deepening refines individual modules. Each stage has a goal G_k.

The scheduling objective is to supply requirements before the decisions that depend on them. The Scheduler predicts these dependencies from stage goals; the controller verifies delivery at the assigned stage entry. Appendix C.2 formalizes this distinction.

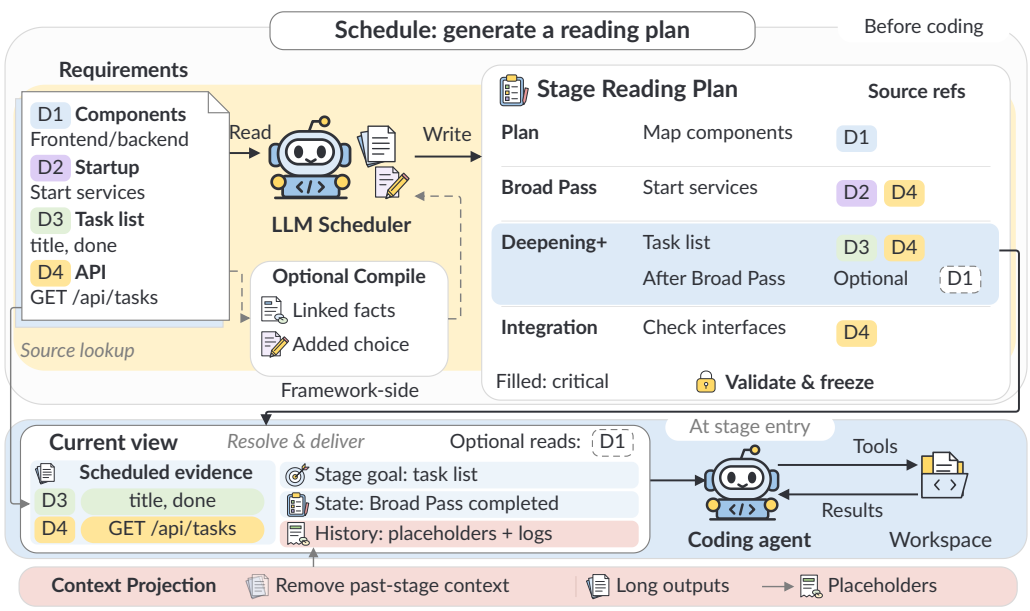

Figure 2: **ReqWeaver generates and executes stage-specific reading plans.** Required evidence joins the stage goal, state, history, and optional source references. Projection masks eligible past-stage tool outputs, preserving failures, current-stage records, raw records, and the workspace. Solid arrows show transfers or operations; dashed arrows show optional compilation. Deepening denotes Module Deepening; + indicates repetition. Assignments are illustrative.

3.2 Execution Overview

Figure 2 separates planning the work from executing it. In an isolated preprocessing session, the Scheduler inspects the public inputs and proposes stage goals, dependencies, and evidence bindings within the prescribed stage grammar. Validation freezes this plan before coding. The coding agent's subsequent Plan stage develops the implementation design within those goals; Broad Pass, Module Deepening, and Integration then implement and check it. When Compilation is enabled, it runs before scheduling.

Let $U = D \cup R(C)$ combine public inputs with the addressable rendering $R(C)$ of contract C, where $R(C) = \emptyset$ without compilation. For stored history H_k, compact execution state X_k, and scheduled evidence packet $E_k \subset U$, the model's input at stage k is

$$V_k = \Pi_k(H_k) \oplus G_k \oplus X_k \oplus E_k, \tag{1}$$

where Π_k is Context Projection and $\oplus$ denotes request assembly. The controller supplies the stage goal, compact progress state, exact required evidence, and references for optional reading. The workspace persists across stages. Appendix C.3 specifies state updates, checkpoints, and repair rules.

The example in Figure 2 shows how one source passage can support several stages, each alongside different requirements.

3.3 Evidence Scheduling

Source identifiers in $\mathcal{A}(U)$, called aliases, resolve deterministically to verbatim input passages or exact contract units. The Scheduler proposes the execution schedule

$$P = (S, M, O), \qquad M_k, O_k \subseteq \mathcal{A}(U), \qquad E_k = \mathrm{Resolve}_U(M_k), \tag{2}$$

where M_k references required evidence (marked critical in Figure 2) and O_k lists optional source references. Selection follows stage goals and dependencies, not document order; evidence may recur across stages. Validation checks the bindings and rejects full-source delivery at every stage or a complete source-order partition (Appendix C.2).

The Scheduler selects source references rather than writing evidence summaries. Its prompt asks it to identify module responsibilities and stage deliverables, then assign the source clauses needed before each behavior is first implemented. API, format, validation, and state-transition constraints belong in M_k for the responsible stage; supporting details remain in O_k unless they constrain that stage's work. For Broad Pass, the Scheduler selects prerequisites for an initial runnable implementation rather than every later detail. Integration reuses evidence when it supports a distinct cross-module check. These LLM instructions are followed by deterministic checks of dependency order, alias resolution, and source integrity (Appendix C).

For example, the saved `xlrd` plan groups workbook-opening and worksheet-lookup requirements for Broad Pass, reserves detailed workbook and sheet behavior for Module Deepening, and reuses import requirements during Integration. This illustrates how bindings follow implementation responsibilities and allow later reuse. Appendix C.2.1 reconstructs the exact packets and source references.

At stage entry, required evidence is inserted verbatim into the request, while optional source references remain in the reading card. Delivery precedes execution of the assigned stage; coverage of the decisions that actually occur depends on the accuracy of the bindings. The scheduling comparison jointly covers source inspection, selection, assignment, and delivery (Appendix C.2).

3.4 Context Projection

After Broad Pass, Context Projection removes completed-stage messages and masks selected past tool outputs. Let $A_k \subset H_k$ contain eligible outputs from successful tool calls, B_k contain past stage-entry messages, and $\rho(a)$ denote placeholders retaining original record identifiers and recovery references. The projected history is

$$\Pi_k(H_k) = \mathrm{Replace}(H_k, A_k, \rho) \setminus B_k. \tag{3}$$

Removing B_k retires the completed stage's request, including its evidence and goal. Eligible successful tool outputs become placeholders, while failures and current-stage records remain expanded. Appendix C.4 gives the selection rules and preservation exceptions.

Projection changes the model's input without LLM summarization or a repository restart. The agent can recover information through permitted source reads or later scheduled delivery. Placeholder identifiers support audit tracing rather than direct retrieval; raw events and checkpoints remain available to the framework.

3.5 Optional Contract Compilation

When enabled, an isolated compiler produces a structured specification called a contract, $C = (F, \Delta)$. Here F consolidates requirements with exact source references, while Δ records design decisions left unspecified by the inputs, such as response conventions or storage formats shared across stages. These decisions are labeled as additions, not recovered stakeholder intent. A deterministic rendering $R(C)$ makes each unit independently addressable without changing its content.

Public inputs remain authoritative: $D \succ C \succ P$, where P is the execution schedule. Compilation cannot write code; units are validated and frozen before scheduling. Validation checks source references and structure; it does not establish semantic correctness (Appendix C.1).

Configurations. NL2Repo-Bench uses Scheduling and Projection; RepoGenesis additionally enables Compilation and schedules source passages and contract units together. Configurations are fixed by benchmark.

This split accommodates both detailed and underspecified inputs. Compilation supplies structured requirements and design decisions where enabled, while Scheduling determines when they are delivered.

The controller advances when a coding turn completes normally; stage goals are not independently verified. Other statuses stop progression. Repair uses public checks only, with the benchmark-specific rules in Appendix C.3; final-evaluator feedback never triggers repair.

4 EXPERIMENTS

4.1 EXPERIMENTAL SETUP

Research questions. We ask whether ReqWeaver improves end-to-end functional performance (**RQ1**), how requirement access, delivery timing, and stage alignment contribute within the staged protocol (**RQ2**), and how evidence binding and Context Projection jointly change token consumption under a shared plan (**RQ3**).

Benchmarks and metrics. *Pass* macro-averages repository test-pass fractions computed using each benchmark's official evaluator settings (Ding et al., 2025; Peng et al., 2026). Within each benchmark, the same official test-collection and scoring rules apply to all compared methods, providing a common basis for score comparison. *Resolved* is the percentage of repositories passing 100% of the tests under these scoring rules: the passed-test count must equal the positive test denominator. Failures and zero-denominator evaluations are not resolved. Benchmarks use the fixed configurations in Section 3 and are reported separately.

Generation receives `start.md` on NL2RepoBench and the public README with dependency metadata on RepoGenesis. Contracts and schedules derive from these inputs, not evaluator feedback. RepoGenesis tests are publicly released, so the protocol restricts generation-time access rather than claiming test secrecy.

Task selection. NL2RepoBench includes 36 randomly selected tasks: 12 per difficulty, with four per README-length stratum within each difficulty. Tasks with fewer than 10 evaluator tests or prior use in method development or protocol preparation were excluded; selection was frozen before the main campaign (Appendix B.1). RepoGenesis includes all 30 tasks; its difficulty display uses a frozen public-test-load proxy, not a sampling stratum.

Models, harnesses, and baselines. We evaluate GPT-5.4-mini and DeepSeek V4 Pro. In Table 1, ReqWeaver runs on OpenHands, with unmodified OpenHands as the within-harness baseline and unmodified Codex as an additional agent baseline. MetaGPT v0.8.2 (Hong et al., 2024), ZeroRepo (Luo et al., 2025), and Zero2Paper (Paper2Code-adapted; Seo et al., 2026) provide external-system comparisons, not exact original-paper reproductions or matched mechanism controls. Input mappings appear in Appendix B.3 and generation settings in Appendix B.2; frozen final repositories are scored locally under Appendix B.4.

Controlled comparisons. Requirement-access comparisons retain Stage Trajectory. Timing holds packets fixed; Aligned–Shuffled varies their assignment across stages. Projection pairs share a Broad Pass checkpoint, and the token comparison reuses the third ReqWeaver run block from the main evaluation with a control that holds its complete stage plan fixed. Appendix B.3 specifies the interventions and shared conditions. Final evaluation scores frozen workspaces without returning feedback to generation. All selected tasks enter aggregation under the scoring policy in Appendix B.4. Generation has no aggregate budget cap; these are functional comparisons rather than matched-compute comparisons.

Statistics and reproducibility. We report means and sample SDs across three stochastic generation runs with fixed model versions and sampling settings, equally weighting repositories within each benchmark. Eligible paired contrasts use crossed repository/run-block bootstrap 95% CIs. Token accounting covers the third ReqWeaver run block (456), rather than a three-run resource average. Its scope and the retained-run diagnostics are summarized in Appendix B.5 with the pairing and resampling procedures.

4.2 END-TO-END PERFORMANCE (RQ1)

ReqWeaver has the highest mean Pass in all four model–benchmark settings in the three-run summary (Table 1). Run-to-run variability remains substantial for some low-scoring baselines. Against OpenHands, Pass intervals exclude zero with GPT-5.4-mini but not DeepSeek V4 Pro.

Table 1: End-to-end functional results under this benchmark adaptation and generation protocol (%; higher is better). Cells report three-run means with sample SDs as subscripts. Zero2Paper is Paper2Code-adapted. Bold marks the best mean per model, benchmark, and metric. Reporting conventions follow Section 4.1.

Method	Model	NL2RepoBench		RepoGenesis	
		Pass	Resolved	Pass	Resolved
MetaGPT	GPT-5.4-mini	$6.77_{\pm 1.26}$	$0.00_{\pm 0.00}$	$10.56_{\pm 1.68}$	$8.89_{\pm 1.92}$
	DeepSeek V4 Pro	$2.46_{\pm 0.77}$	$0.00_{\pm 0.00}$	$3.33_{\pm 2.52}$	$1.11_{\pm 1.92}$
ZeroRepo	GPT-5.4-mini	$0.06_{\pm 0.10}$	$0.00_{\pm 0.00}$	$27.59_{\pm 2.45}$	$0.00_{\pm 0.00}$
	DeepSeek V4 Pro	$2.80_{\pm 2.07}$	$0.93_{\pm 1.60}$	$28.94_{\pm 2.48}$	$0.00_{\pm 0.00}$
Zero2Paper	GPT-5.4-mini	$25.46_{\pm 2.18}$	$2.78_{\pm 2.78}$	$7.87_{\pm 1.47}$	$5.56_{\pm 1.92}$
	DeepSeek V4 Pro	$12.15_{\pm 1.63}$	$0.00_{\pm 0.00}$	$0.00_{\pm 0.00}$	$0.00_{\pm 0.00}$
Codex	GPT-5.4-mini	$36.69_{\pm 2.41}$	$\mathbf{5.56}_{\pm 2.78}$	$69.05_{\pm 2.53}$	$36.67_{\pm 3.33}$
	DeepSeek V4 Pro	$34.31_{\pm 2.37}$	$\mathbf{8.33}_{\pm 2.78}$	$44.29_{\pm 2.72}$	$23.33_{\pm 3.33}$
OpenHands	GPT-5.4-mini	$37.20_{\pm 2.42}$	$2.78_{\pm 2.78}$	$55.03_{\pm 2.72}$	$33.33_{\pm 3.33}$
	DeepSeek V4 Pro	$32.79_{\pm 2.35}$	$2.78_{\pm 2.78}$	$69.69_{\pm 2.52}$	$\mathbf{33.33}_{\pm 3.33}$
ReqWeaver (Ours)	GPT-5.4-mini	$\mathbf{46.47}_{\pm 2.49}$	$\mathbf{5.56}_{\pm 2.78}$	$\mathbf{78.45}_{\pm 2.25}$	$\mathbf{40.00}_{\pm 3.33}$
	DeepSeek V4 Pro	$\mathbf{35.83}_{\pm 2.40}$	$5.56_{\pm 2.78}$	$\mathbf{78.61}_{\pm 2.25}$	$\mathbf{33.33}_{\pm 3.33}$

Table 2: Pass by difficulty with GPT-5.4-mini/OpenHands (%). NL2RepoBench uses benchmark labels; RepoGenesis uses the frozen public-test-load proxy.

Benchmark	Method	Easy	Medium	Hard
NL2RepoBench	OpenHands	$62.53_{\pm 4.19}$	$32.98_{\pm 4.07}$	$16.08_{\pm 3.17}$
	ReqWeaver	$\mathbf{70.29}_{\pm 3.96}$	$\mathbf{40.87}_{\pm 4.26}$	$\mathbf{28.26}_{\pm 3.90}$
RepoGenesis	OpenHands	$62.88_{\pm 4.37}$	$47.57_{\pm 4.52}$	$54.51_{\pm 5.28}$
	ReqWeaver	$\mathbf{80.52}_{\pm 3.58}$	$\mathbf{69.02}_{\pm 4.18}$	$\mathbf{88.57}_{\pm 3.37}$

Comparison with OpenHands. With GPT-5.4-mini, the NL2RepoBench Pass difference is $+9.28$ points (95% CI, $[+0.79, +17.83]$). On RepoGenesis, the Pass difference is $+23.42$ points ($[+2.22, +44.18]$). Both intervals exclude zero, supporting higher average test pass rates across the two benchmarks.

With DeepSeek V4 Pro, Pass differences are $+3.04$ points on NL2RepoBench (95% CI, $[-3.72, +9.83]$) and $+8.92$ points on RepoGenesis ($[-7.76, +25.55]$). Both intervals include zero.

Functional coverage and complete resolution. ReqWeaver's lead in mean Pass is more consistent than its lead in Resolved. Higher average test-pass fractions do not uniformly yield more fully resolved repositories. External-system scores also reflect code incompatibility, startup/collection blockers, and workflow failures under the adapted protocol (Appendix D.7.2); the within-OpenHands comparisons provide the primary system contrast.

Difficulty breakdown. ReqWeaver improves mean Pass in every difficulty stratum (Table 2). The largest descriptive gaps occur on Hard tasks: $+12.18$ points on NL2RepoBench and $+34.06$ under RepoGenesis's proxy, so gains are not confined to easier repositories.

4.3 Mechanism Analysis (RQ2)

Evidence Scheduling controls. The stage-assignment contrast favors Aligned over Shuffled ($+2.74$ points; 95% CI, $[+0.23, +5.24]$), while access-strategy and matched-content timing intervals include zero (Table 3). These comparisons distinguish evidence selection, delivery time, and stage assignment.

Alternative strategies. Full-source upfront and Stage-wise RAG have similar mean Pass, both above agent-directed reading but below Scheduling. Scheduling leads these two alternatives by 6.74 and 6.30 points, respectively, suggesting that complete initial exposure or stage-conditioned retrieval

Table 3: Evidence Scheduling controls on NL2RepoBench (GPT-5.4-mini/OpenHands; Pass, %; three-run mean ± SD). Agent-directed reading retains Stage Trajectory. Timing fixes trajectory, packets, and budget; stage assignment permutes the same packets. Aligned − Shuffled: $+2.74$ pp (95% CI, $[+0.23, +5.24]$).

(a) Access strategy	Pass
Agent-directed reading	$32.95_{\pm 2.35}$
Full-source upfront	$37.46_{\pm 2.42}$
Stage-wise RAG	$37.91_{\pm 2.43}$
Evidence Scheduling	$\mathbf{44.21}_{\pm 2.48}$

(b) Matched-content timing	Pass
Upfront delivery	$41.19_{\pm 2.46}$
Staged delivery	$\mathbf{44.48}_{\pm 2.48}$

(c) Stage assignment	Pass
Shuffled	$40.95_{\pm 2.46}$
Aligned	$\mathbf{43.69}_{\pm 2.48}$

Table 4: Cumulative component ablation with GPT-5.4-mini/OpenHands (%). Each path retains previously enabled components, including Stage Trajectory in all subsequent rows. NL2RepoBench omits Compilation; RepoGenesis first reports Trajectory together with Compilation. Dashes mark the omitted Compilation step on NL2RepoBench and the unreported trajectory-only result on RepoGenesis.

Configuration	NL2RepoBench		RepoGenesis	
	Pass	Resolved	Pass	Resolved
OpenHands	$37.20_{\pm 2.42}$	$2.78_{\pm 2.78}$	$55.03_{\pm 2.72}$	$33.33_{\pm 3.33}$
+ Stage Trajectory	$32.95_{\pm 2.35}$	$5.56_{\pm 2.78}$	—	—
+ Contract Compilation	—	—	$60.26_{\pm 2.68}$	$20.00_{\pm 3.33}$
+ Evidence Scheduling	$44.21_{\pm 2.48}$	$5.56_{\pm 2.78}$	$67.60_{\pm 2.56}$	$23.33_{\pm 3.33}$
+ Context Projection (**ReqWeaver**)	$\mathbf{46.47}_{\pm 2.49}$	$\mathbf{5.56}_{\pm 2.78}$	$\mathbf{78.45}_{\pm 2.25}$	$\mathbf{40.00}_{\pm 3.33}$

alone does not reproduce its mean performance. The access-strategy contrasts remain uncertain: all three 95% CIs include zero (Appendix D.1).

Matched-content timing. With packet content held fixed, staged delivery has a $+3.29$-point mean advantage over upfront delivery (95% CI, $[-8.45, +14.92]$). The direction favors staged access to the same evidence, although the interval leaves its benefit uncertain.

Stage alignment. The Aligned–Shuffled comparison preserves packet content while changing its stage assignment. Its positive interval supports the intended assignment relative to this fixed shuffled control, although it does not separate semantic relevance from differences in per-stage packet length.

Ordered protocol ablation. The cumulative paths in Table 4 place these controls in the full system. NL2RepoBench first adds Stage Trajectory; RepoGenesis first adds Stage Trajectory together with Contract Compilation. Projection follows Scheduling in both paths.

On NL2RepoBench, the Stage Trajectory configuration has lower mean Pass than OpenHands (32.95% versus 37.20%), while adding Scheduling raises it to 44.21%, the largest mean increment on this path ($+11.25$ points). This cumulative comparison does not identify coordination overhead as the cause of the initial decrease.

On RepoGenesis, mean Pass increases along the cumulative path, with the largest increment following Projection ($+10.86$ points). Resolved does not rise at every step, distinguishing broader functional coverage from complete repository success. The different paths suggest that component contributions depend on the task and configuration rather than following a universal ranking.

With Stage Trajectory and Evidence Scheduling fixed on NL2RepoBench, adding Projection raises three-run mean Pass from 44.21% to 46.47% ($+2.27$ points; 95% CI, $[-4.02, +8.44]$), leaving the functional benefit uncertain. All ordered-component Pass and Resolved intervals include zero; Appendix D.1 reports the contrasts. These cumulative comparisons suggest coordination hypotheses but do not isolate component interactions or the standalone benefit of optional Contract Compilation. Retained-run task diagnostics appear separately in Appendix D.6.

Implementation evidence. The retained `xlrd` Scheduling–Stage Trajectory pair illustrates a concrete implementation difference: the Scheduling arm parses worksheets where its control leaves an empty list. Appendix D.7 traces the source evidence, code, and test outcomes. A separate complete-system comparison with raw OpenHands examines 36 retained trajectory pairs (Appendix D.5); its descriptive patterns do not identify individual component effects.

4.4 Token Consumption under a Shared Plan (RQ3)

Under the same frozen plan, ReqWeaver uses fewer total tokens than the control without evidence binding or Projection (Figure 3). The ReqWeaver trajectories are from the third run block (456) included in Table 1. Across these 36 repositories and their same-plan controls, consumption decreases from 130.08M to 116.34M (-10.56%), including identical planning allocations. The control retains full history and autonomous access to all requirements.

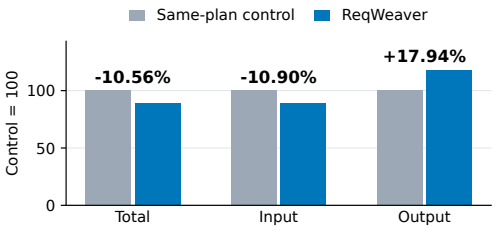

Figure 3: Same-plan tokens for the third ReqWeaver run block (456) from Table 1, across 36 NL2RepoBench repositories (GPT-5.4-mini/OpenHands), control $= 100$. Total includes planning; Input and Output cover coding only. Output includes reasoning.

Lower cumulative input accounts for the reduction: coding input decreases by 10.90%, while output including reasoning increases by 17.94%. The absolute input savings outweigh the additional output tokens.

Coding tokens decrease in 23 repositories and increase in 13. Excluding the pair whose `boltons` control reaches the context limit, the 35 generation-complete pairs still use 10.17% fewer coding tokens. The accounting includes failed records and equal planning allocations; Appendix D.2 reports the complete token breakdown and scope. The token totals describe this single-run comparison; Table 1 reports functional performance averaged over three runs.

The shared plan fixes what work is requested, while evidence binding and Projection change the context available during that work. The control can still read the full requirements, and both arms retain their workspaces across stages. Their actual tool actions and call counts can therefore diverge. The measured reduction concerns cumulative consumption along these resulting trajectories, including repeated input across calls. It does not imply that every request is shorter or every repository consumes fewer tokens. The per-repository breakdown in Appendix D.2 makes this variation explicit.

5 Conclusion

ReqWeaver extends execution planning with frozen source-to-stage bindings, exact stage-entry delivery, and retirement and reuse of requirement evidence. Three-run evaluation shows higher mean Pass than OpenHands with GPT-5.4-mini on both benchmarks, with both confidence intervals excluding zero. The Aligned–Shuffled comparison supports intended stage assignment, while the benefit of matched-content delivery timing remains uncertain. Using the third ReqWeaver run block from the main evaluation and same-plan controls, evidence binding and Projection jointly reduce total tokens, including planning, by 10.56% across 36 NL2RepoBench repositories. Together, these comparisons characterize a protocol that specifies both the development work and the source evidence accompanying it.

The protocol also exposes a concrete boundary for further improvement: the Scheduler predicts which evidence a stage will need, but the coding agent may act beyond that stage's anticipated scope. Keeping public sources readable allows corrective access when this happens. Future work can examine when these deviations justify revising a frozen plan and how revised bindings should be evaluated. Such extensions would need to preserve source traceability while distinguishing better requirement selection from additional coding effort.

AI USE STATEMENT

We used generative AI tools to suggest manuscript structure, draft, revise, and proofread prose, discover and organize candidate literature, and prepare the LaTeX manuscript scaffold. Apart from the LLM-based systems studied in this work, we did not use generative AI assistance for research methodology, experiment design, method implementation, data processing, result interpretation, or translation. The authors reviewed the AI-assisted material and take responsibility for the final content of this work.

REPRODUCIBILITY STATEMENT

Experimental inputs, task selection, and evaluation settings are described in Section 4.1 and Appendix B. Appendix C presents protocol details alongside prompt excerpts and structured examples. Appendix B.5 describes three-run aggregation and uncertainty estimation, and distinguishes them from retained-run diagnostics and resource measurements. The main-method implementation, benchmark input and evaluation adapters, and environment setup instructions are available in the anonymous repository: `https://anonymous.4open.science/r/ReqWeaver-72D4/`.

REFERENCES

Ramakrishna Bairi, Atharv Sonwane, Aditya Kanade, Vageesh D. C., Arun Iyer, Suresh Parthasarathy, Sriram Rajamani, B. Ashok, and Shashank Shet. CodePlan: Repository-level coding using LLMs and planning. *Proceedings of the ACM on Software Engineering*, 2024. doi: 10.1145/3643757. URL https://doi.org/10.1145/3643757.

Wei Cheng, Yuhan Wu, and Wei Hu. Dataflow-guided retrieval augmentation for repository-level code completion. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024. doi: 10.18653/v1/2024.acl-long.431. URL https://doi.org/10.18653/v1/2024.acl-long.431.

Drew Dillon and Kasyap Varanasi. Context-augmented code generation: How product context improves AI coding agent decision compliance by 49%. *arXiv preprint arXiv:2605.08112*, 2026. URL https://arxiv.org/abs/2605.08112.

Jingzhe Ding, Shengda Long, Changxin Pu, Huan Zhou, Hongwan Gao, Xiang Gao, Chao He, et al. NL2Repo-Bench: Towards long-horizon repository generation evaluation of coding agents. *arXiv preprint arXiv:2512.12730*, 2025. URL https://arxiv.org/abs/2512.12730.

Thibaud Gloaguen, Niels Mündler, Mark Müller, Veselin Raychev, and Martin Vechev. Evaluating AGENTS.md: Are repository-level context files helpful for coding agents? *arXiv preprint arXiv:2602.11988*, 2026. URL https://arxiv.org/abs/2602.11988.

Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First IEEE International Conference on Requirements Engineering*, pp. 94–101, 1994. doi: 10.1109/ICRE.1994.292398. URL https://doi.org/10.1109/ICRE.1994.292398.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, 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. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Mengkang Hu, Tianxing Chen, Qiguang Chen, Yao Mu, Wenqi Shao, and Ping Luo. HiAgent: Hierarchical working memory management for solving long-horizon agent tasks with large language model. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics*, 2025. doi: 10.18653/v1/2025.acl-long.1575. URL https://doi.org/10.18653/v1/2025.acl-long.1575.

ISO/IEC/IEEE. ISO/IEC/IEEE International Standard—Systems and Software Engineering—Life Cycle Processes—Requirements Engineering. International Standard 29148:2018, IEEE, 2018. URL https://doi.org/10.1109/IEEESTD.2018.8559686.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LongLLMLingua: Accelerating and enhancing LLMs in long context scenarios via prompt compression. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024. doi: 10.18653/v1/2024.acl-long.91. URL https://doi.org/10.18653/v1/2024.acl-long.91.

Tobias Lindenbauer, Igor Slinko, Ludwig Felder, Egor Bogomolov, and Yaroslav Zharov. The complexity trap: Simple observation masking is as efficient as LLM summarization for agent context management. *arXiv preprint arXiv:2508.21433*, 2025. URL https://arxiv.org/abs/2508.21433. DL4C at NeurIPS 2025 camera-ready version.

Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, 2024a. doi: 10.1162/tacl_a_00638. URL https://doi.org/10.1162/tacl_a_00638.

Wei Liu, Ailun Yu, Daoguang Zan, Bo Shen, Wei Zhang, Haiyan Zhao, Zhi Jin, and Qianxiang Wang. GraphCoder: Enhancing repository-level code completion via coarse-to-fine retrieval based on code context graph. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024b. doi: 10.1145/3691620.3695054. URL https://doi.org/10.1145/3691620.3695054.

Jane Luo, Xin Zhang, Steven Liu, Jie Wu, Jianfeng Liu, Yiming Huang, Yangyu Huang, Chengyu Yin, Ying Xin, Yuefeng Zhan, Hao Sun, Qi Chen, Scarlett Li, and Mao Yang. RPG: A repository planning graph for unified and scalable codebase generation. *arXiv preprint arXiv:2509.16198*, 2025. URL https://arxiv.org/abs/2509.16198.

Alistair Mavin, Philip Wilkinson, Adrian Harwood, and Mark Novak. Easy approach to requirements syntax (EARS). In *2009 17th IEEE International Requirements Engineering Conference*, pp. 317–322, 2009. doi: 10.1109/RE.2009.9. URL https://doi.org/10.1109/RE.2009.9.

Bertrand Meyer. Applying 'design by contract'. *Computer*, 25(10):40–51, 1992. doi: 10.1109/2.161279. URL https://doi.org/10.1109/2.161279.

Zhiyuan Peng, Xin Yin, Pu Zhao, Fangkai Yang, Lu Wang, Ran Jia, Xu Chen, Qingwei Lin, Saravan Rajmohan, and Dongmei Zhang. RepoGenesis: Benchmarking end-to-end microservice generation from readme to repository. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, 2026. doi: 10.18653/v1/2026.acl-long.415. URL https://doi.org/10.18653/v1/2026.acl-long.415.

Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. Paper2Code: Automating code generation from scientific papers in machine learning. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=3DcaUTjdKc.

Jeff Tyree and Art Akerman. Architecture decisions: Demystifying architecture. *IEEE Software*, 22(2):19–27, 2005. doi: 10.1109/MS.2005.27. URL https://doi.org/10.1109/MS.2005.27.

Peiding Wang, Li Zhang, Fang Liu, Taichuan Li, and Yinghao Zhu. CodeSpec: Dual executable specifications for agentic long-horizon feature development. *arXiv preprint arXiv:2607.26777*, 2026. URL https://arxiv.org/abs/2607.26777.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, et al. OpenHands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Hironori Washizaki (ed.). *Guide to the Software Engineering Body of Knowledge (SWEBOK Guide), Version 4.0a*. IEEE Computer Society, 2026. URL https://www.computer.org/education/bodies-of-knowledge/software-engineering.

Di Wu, Wasi Uddin Ahmad, Dejiao Zhang, Murali Krishna Ramanathan, and Xiaofei Ma. Repoformer: Selective retrieval for repository-level code completion. In *Proceedings of the 41st International Conference on Machine Learning*, 2024. URL https://arxiv.org/abs/2403.10059.

Yuan-An Xiao, Pengfei Gao, Chao Peng, and Yingfei Xiong. Reducing cost of LLM agents with trajectory reduction. *Proceedings of the ACM on Software Engineering*, 2026. doi: 10.1145/3797084. URL https://doi.org/10.1145/3797084. FSE 2026; arXiv:2509.23586.

Daoguang Zan, Ailun Yu, Wei Liu, Dong Chen, Bo Shen, Wei Li, Yafen Yao, Yongshun Gong, Xiaolin Chen, Bei Guan, Zhiguang Yang, Yongji Wang, Qianxiang Wang, and Lizhen Cui. CodeS: Natural language to code repository via multi-layer sketch. *arXiv preprint arXiv:2403.16443*, 2024. URL https://arxiv.org/abs/2403.16443.

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 *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. doi: 10.18653/v1/2023.emnlp-main.151. URL https://doi.org/10.18653/v1/2023.emnlp-main.151.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*, 2024. doi: 10.18653/v1/2024.acl-long.737. URL https://doi.org/10.18653/v1/2024.acl-long.737.

Zhaoxi Zhang, Yiming Xu, Jiahui Liang, Weikang Li, Xiaoshuai Chen, Liwei Qian, Xin Pei, Jizhou Huang, Run Sun, and Yunfang Wu. RepoZero: Can LLMs generate a code repository from scratch? *arXiv preprint arXiv:2605.07122*, 2026. URL https://arxiv.org/abs/2605.07122.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T. Chiu, Claire Cardie, Matthias Gallé, and Alexander M. Rush. Commit0: Library generation from scratch. *arXiv preprint arXiv:2412.01769*, 2024. URL https://arxiv.org/abs/2412.01769.

Xin Zhou, Chun Yong Chong, Kisub Kim, Yun Peng, Rui Shu, Zihan Wu, Xu Han, Guowen Yuan, Zeyang Zhuang, Jounghoon Kim, Jeongjin Ju, Seongmin Ju, Taein Yoon, and David Lo. A unified issue resolution benchmark for requirement clarification, planning, and code generation for coding agents. *arXiv preprint arXiv:2608.09072*, 2026. URL https://arxiv.org/abs/2608.09072.

A Limitations

Evidence Scheduling jointly changes source selection, stage assignment, packet size, delivery time, and persistence. Full-source-upfront and Stage-wise RAG controls narrow the explanation but do not isolate timing. The matched-content timing interval includes zero, whereas the paired Aligned–Shuffled interval excludes zero. The stage-assignment result is established in one model–benchmark–harness setting; its generality remains to be tested. Delivery telemetry verifies exposure before stage execution, not semantic use. The cumulative ablation does not establish the necessity or standalone benefit of optional Contract Compilation.

Evidence for individual components and cross-model transfer remains limited by the small number of repeated runs. Higher average Pass also does not consistently translate into more fully resolved repositories. Resource, trajectory, and length diagnostics describe retained runs rather than independent replications.

B Experimental Details

B.1 Benchmarks, Task Selection, and Input Access

We use the fixed 36-task NL2RepoBench selection and all 30 RepoGenesis tasks described in Section 4.1 (Ding et al., 2025; Peng et al., 2026). The NL2RepoBench exclusions include tasks previ-

ously used as canaries. README lengths use `o200k_base`. RepoGenesis difficulty is a descriptive public-test-load proxy: at most 12, 13–33, and more than 33 declared tests define Easy, Medium, and Hard, with 11/11/8 tasks. Membership is fixed independently of generated repositories' test-collection outcomes.

Generation inputs are `start.md` for NL2RepoBench and the public README with dependency metadata for RepoGenesis. Treatment preprocessing constructs contracts and schedules from these inputs. Benchmark tests and final-evaluator feedback are excluded from the intended generation input; agent-written public checks remain available. RepoGenesis tests are publicly released, so this restriction concerns generation-time access rather than secrecy.

B.2 Models, Runtimes, and Generation Budgets

The deployments are `gpt-5.4-mini-20260317` and `DeepSeek-V4-Pro`. OpenHands uses SDK 1.33.0 with Python 3.12; Codex uses CLI 0.145.0. All models use high reasoning effort, API-default temperature and top-p, and a maximum output of 65,535 tokens per request. No aggregate generation budget is imposed on tokens, calls, iterations, or wall time. The per-request output limit is distinct from a total generation budget. Table 5 summarizes these settings.

Table 5: Generation settings shared across the evaluated models. API defaults are used without explicit temperature or top-p overrides.

Setting	Value
Reasoning effort	High
Temperature / top-p	API defaults
Maximum output per request	65,535 tokens
Aggregate generation budget	No token, call, iteration, or wall-time cap

Method-specific validation, repair, and projection rules are described with their prompt excerpts in Appendix C.

B.3 Baseline Adaptations and Controlled Configurations

MetaGPT receives requirements as its project idea; Zero2Paper maps them into Paper2Code's paper-JSON interface; ZeroRepo uses simple feature selection (Hong et al., 2024; Seo et al., 2026; Luo et al., 2025). These are system comparisons, not exact original-paper reproductions. Table 6 defines the controlled conditions. Public sources remain readable throughout; scheduled delivery does not prohibit self-directed retrieval.

Table 6: Requirement-access conditions: NL = NL2RepoBench; RG = RepoGenesis. Non-raw rows use a Stage Trajectory; RG treatments also compile a contract. Full denotes no ReqWeaver Projection.

Benchmark	Configuration (ID)	Framework-side evidence delivery	History
NL	Raw (E0)	Agent-initiated reads	Full
NL	Stage Trajectory (E1)	No stage source packets	Full
NL	Full Source Upfront (E2)	Complete source in opening Plan turn	Full
NL	Stage-wise RAG (E3)	Retrieved sections at stage entry	Full
NL	Evidence Scheduling (E4)	Frozen exact packets at stage entry	Full
NL	Aligned control	Frozen control packets at intended stages	Full
NL	Shuffled control	Same control packets reassigned across stages	Full
NL	Context Projection (E5)	Same scheduled delivery as E4	Projected
NL	Matched timing (T0/TU)	Same packets at stage entry / opening	Full
RG	Raw (R0)	Agent-initiated reads	Full
RG	Contract + Trajectory (R2)	Compiled contract; no stage packets	Full
RG	+ Scheduling (R3)	Stage-addressed evidence	Full
RG	+ Projection (R4)	Stage-addressed evidence	Projected

Full Source Upfront (E2) supplies the complete source document in the opening Plan turn. E3 equally weights min–max-normalized BM25 ($k_1 = 1.5$, $b = 0.75$) and `BAAI/bge-small-en-v1.5` similarity, using stage goals, dependencies, and workspace state as the query. Whole sections fit the corresponding E4 character budget, with a logged smallest-section fallback if none fits.

Primary OpenHands E4/E5 suffixes share the Broad Pass workspace, schedule, and event prefix. T0/TU instead holds the trajectory and exact packets fixed while varying delivery time. RepoGenesis R2 jointly adds Contract Compilation and Stage Trajectory; it is not a trajectory-only control.

Aligned–Shuffled stage-assignment comparison. The Aligned–Shuffled experiment uses separately generated plans and a shared neutral prompt wrapper, rather than reusing the E4 trajectories in the access-strategy comparison. Aligned retains the intended source-to-stage assignment, while Shuffled applies a fixed Sattolo permutation (shuffle seed: 20260913) to the same packet multiset. The paired arms share the task input, initial workspace, Stage Trajectory, prompt wrapper, history policy, generation budget, evaluator, repository, and preregistered run block; only source-to-stage assignment changes.

B.4 Final Evaluation and Failure Handling

Frozen workspaces are scored with each benchmark's official evaluator settings, without feedback to generation. The same collection and scoring rules apply to every method; retained-run timeout exceptions are specified below. Collection failures can nevertheless produce different evaluated test sets across generated repositories. Pass is therefore interpreted under the benchmark's scoring protocol; identical scoring rules do not guarantee identical test coverage across implementations. As a sensitivity analysis motivated by the evaluated-test-set mismatch in `deepdiff`, we exclude this repository from both arms in all three runs of each NL2RepoBench comparison in Table 7, retaining the same 35 of 36 repositories. The intervals follow Appendix B.5. The GPT-5.4-mini complete-system and Aligned–Shuffled intervals remain above zero; the other four intervals continue to include zero. These results are conditional on the retained subset and do not establish identical test coverage across implementations.

Table 7: Sensitivity to excluding `deepdiff` from all three runs on NL2RepoBench (35 repositories; Pass differences and 95% CIs in percentage points). All comparisons use OpenHands; mechanism comparisons use GPT-5.4-mini. The complete-system comparisons identify the model explicitly.

Comparison	Mean difference	95% CI
ReqWeaver − OpenHands (GPT-5.4-mini)	$+9.31$	$[+0.66, +18.09]$
ReqWeaver − OpenHands (DeepSeek V4 Pro)	$+3.04$	$[-4.16, +10.26]$
Scheduling − Stage-wise RAG	$+6.45$	$[-3.84, +16.75]$
Projection − Full History	$+2.25$	$[-4.03, +8.65]$
Staged − Upfront	$+3.30$	$[-9.14, +15.74]$
Aligned − Shuffled	$+2.91$	$[+0.40, +5.40]$

Following the official protocol, NL2RepoBench continues collected tests despite collection errors, normally with a 30 s per-test limit; both retained `deepdiff` timing arms use a 5 s process limit. The official scoring rule uses collected counts consistent with reported outcomes and otherwise the adapter-reported total; skips earn no credit and zero denominators yield zero. RepoGenesis uses versioned Python/Java build-and-test adapters; recognized failures without test outcomes receive zero credit and a recorded failure phase.

Repository-level aggregation includes all 36 NL2RepoBench or all 30 RepoGenesis tasks; these task counts are distinct from the within-repository test denominators determined by the official evaluators. Missing generations or evaluator timeouts receive assigned zeros only under the corresponding result's scoring policy, distinct from observed test failures. The retained E3 campaign includes 33 evaluated tasks and three generation-failure zeros. Infrastructure recovery and evaluator reruns are not independent generation repetitions.

B.5 AGGREGATION AND UNCERTAINTY

Generation repetitions. Tables 1–4 and Figure 8 summarize three stochastic generation runs labeled 42, 123, and 456, not API sampling seeds. Model versions and sampling settings remain fixed, and all runs are included. Contracts and schedules are regenerated between runs; within-run matched-content and shared-prefix comparisons retain the common artifacts specified in Appendix B.3.

Table 8: Studies and reporting scopes. Resource and trajectory studies are not additional three-run functional replications.

Study	Runs and controls	Interpretation
Functional comparisons	Three stochastic runs; controls in Appendix B.3	Repository-macro means, run SDs, and paired crossed-bootstrap CIs
Same-plan tokens	The third ReqWeaver run block (456) from the main evaluation, paired with a control using identical frozen goals, order, and dependencies	Joint binding-plus-Projection totals; failed records included and planning allocated equally
Retained diagnostics	Task pairs for access, implementation, Projection resources, and length	Conditional on the retained runs; not decompositions of three-run means
Contract binding	Separate 30-task development comparison	Exploratory delivery of compiled evidence with the compiler retained

Aggregation. Let $p_{m,r,s} \in [0, 100]$ be the unrounded Pass percentage for configuration m, repository r, and run block $s \in \{42, 123, 456\}$. Resolved is $z_{m,r,s} = 100\mathbf{1}[p_{m,r,s} = 100]$, implemented by comparing integer passed-test counts with positive denominators. For either outcome, each run equally weights repositories within its benchmark. Means and sample SDs summarize the three run aggregates, with SD denominator $3 - 1$; calculations use unrounded scores.

Paired crossed-bootstrap intervals. For configurations A and B, each replicate independently draws N repository indices $(r_1, \ldots, r_N)$ with replacement and three run-block indices (s_1, s_2, s_3) with replacement. The block indices are drawn once per replicate and shared across all sampled repositories, not redrawn within each repository. Their Cartesian product defines $3N$ cells, using identical repository and block indices for both configurations. The replicate mean difference is

$$\widehat{\Delta}^{*(b)} = \frac{1}{3N} \sum_{i=1}^{N} \sum_{j=1}^{3} \left(y_{A,r_i,s_j} - y_{B,r_i,s_j} \right). \tag{4}$$

We use 10,000 replicates (bootstrap seed: 20260908) and the 2.5th and 97.5th percentiles for the 95% CI. Overall intervals resample all 36 NL2RepoBench or 30 RepoGenesis repositories without stratification. Difficulty-specific intervals use the same crossed procedure within each group: 12 tasks per NL2RepoBench group and 11/11/8 for RepoGenesis.

Pairing eligibility and reporting scopes. Pairs require the same benchmark, repository, preregistered run block, input, initial environment, budget, and evaluator; shared-prefix comparisons additionally require the same checkpoint. Numeric run labels alone do not establish pairing. Retained-run diagnostics instead use paired repository bootstrap intervals, conditional on those runs, and are not decompositions of the three-run means.

Captions identify mixed-scope displays: three-run Pass summaries and retained-run resources. The contract-binding study (Appendix D.8) is a separate exploratory development comparison.

Resource accounting includes recorded input/output tokens, calls, and generation time. OpenHands costs include preflight and coding, counting the shared prefix once per E4/E5 arm. Codex call proxies are not pooled with OpenHands invocation counts. Input/call is total input divided by calls; cumulative input and monetary cost are distinct quantities.

C METHOD DETAILS AND PROMPTS

Each module below pairs its runtime rules with the relevant prompt or example. Prompt boxes reproduce task-message excerpts: braces denote dynamic substitutions and `[...]` denotes omissions. They retain source terminology such as TRP and do not include every system/tool message or adapter branch.

C.1 CONTRACT COMPILATION AND SOURCE GROUNDING

Compilation and validation. When enabled, a separate compiler receives public requirements and dependency metadata and produces a Repository Contract, not application code. The contract covers runtime requirements, interfaces, validation, shared state, and transitions. Source-backed facts cite exact public-text anchors; supplementary decisions record their choice, rationale, scope, and confidence. Public requirements take precedence over conflicting decisions.

Validation checks types, unique identifiers, resolvable references, verbatim anchors, compactness, and adapter-specific consistency. Invalid candidates receive bounded repair before handoff. These checks establish structural consistency rather than semantic entailment.

Schedulable units. The validated `repo_spec.json` is rendered as `repo_spec.schedule.md`, whose runtime, interface, behavior, state, decision, and source-reference units join the scheduling universe. References are validated against the complete contract, but packets include only explicitly selected units: reference targets are not expanded automatically. Complete contract artifacts remain framework-side; the coding agent receives selected units rather than the full contract. Without Compilation, scheduling uses public sources directly.

Compiler prompt. The full template supplies metadata, the schema, validation constraints, and exact public inputs. Its core grounding instructions are:

Prompt 1: Contract compilation

```
Create one compact implementation contract JSON at
.forgezero/repo_spec.json.  [...]  Do not merely transcribe the
README. First extract explicit facts, then identify missing
externally observable details needed for a coherent runnable
implementation, resolve them conservatively, and review each choice
for unjustified rejection or hardening.  Explicit facts use source
references.  Every inferred observable choice uses a decision
reference.  [...]  Never present an inferred choice as an explicit
README fact.  [...]  Public inputs remain authoritative over
decisions.  Never invent hidden-test-specific behavior.

Exact public inputs:
===== {document} =====
{exact_public_input}
```

C.2 SOURCE ADDRESSING AND EVIDENCE SCHEDULING

Delivery objective. An evidence item e is a source passage or compiled specification unit. Let $\mathcal{Q}_k(e)$ contain the decisions in stage s_k whose correctness depends on e. ReqWeaver seeks to deliver e to a stage predicted to need it before that stage acts:

$$\tau_{\text{deliver}}(e, k) < \min_{q \in \mathcal{Q}_k(e)} \tau(q). \tag{5}$$

For nonempty $\mathcal{Q}_k(e)$, this is a design objective: the true decision dependencies are not observed by the Scheduler. It predicts bindings from stage goals; the controller can verify delivery before the assigned stage starts, not that every relevant decision received its prerequisites.

Addressing and assignment. SourceMap indexes document names, headings, line intervals, and content fingerprints. Deterministic aliases such as `S001` identify units sorted by document and

source position. In an isolated preflight request, the Scheduler receives the complete input universe and alias catalog and selects references rather than paraphrasing source text.

Each stage records its kind, goal, reading intent, dependencies, required aliases M_k, and optional aliases O_k. Required evidence corresponds to the schema field `must_read` and the figure label *critical*; optional references use `optional_read`. Assignment follows anticipated stage decisions rather than document order. Units may recur across stages; critical and optional sets within a stage are disjoint.

Validation and freezing. The validator checks stage grammar, dependency order, alias resolution, nonempty critical bindings, compactness, and source integrity. For multiple stages and multiple source units, it rejects plans in which every stage requires the entire source universe. The source-order check rejects a nonempty universe partitioned completely into consecutive required-alias groups in execution order, with no optional, reference-only, or unresolved aliases. Each group must be nonempty and contiguous, and their concatenation must cover the source ordering exactly once. This structural check detects that specific full-sweep pattern; it does not validate semantic relevance. Invalid plans receive at most two repair attempts; valid plans are frozen with input, alias-table, and schedule fingerprints before coding.

Scheduler prompt. The full template also specifies stage grammar and validation rules. Validation errors are returned in the same preflight conversation.

Prompt 2: Schedule authoring

```
Create one compact Semantic Reading Plan at
.forgezero/trp_semantic_plan.json.  [...]  Goals are work
deliverables, not reading activities.  must_read is the smallest
semantic prerequisite before the related write; optional_read is
useful fallback.  [...]  Assign aliases dependency-first, not by
README order.  Do not mechanically partition S001 through the final
alias into consecutive per-stage must_read chunks.  [...]  Do not
assign the full source as must_read to every stage.  [...]

SOURCE-ORDER ALIAS TABLE:
{source_order_alias_table}
===== BEGIN EXACT SOURCE {document}:1-{line_count} =====
{exact_public_input}
===== END EXACT SOURCE {document} =====
```

C.2.1 SOURCE-TO-PACKET EXAMPLE

The saved `xlrd` plan uses schema `forgezero_trp_semantic_plan_v2`. The stage object below illustrates its Broad Pass assignment; execution follows list order, with dependencies defaulting to the preceding stage when omitted. Packet text is reconstructed from the saved plan and source, not a captured historical request. The implementation outcome is reported separately in Appendix D.7.

```
{
  "id": "broad_first_pass",
  "kind": "broad_first_pass",
  "goal":
    "Build the installable package shell and core workbook open path",
  "read_intent":
    "Resolve top-level API and workbook bootstrap before the first write",
  "must_read": ["S010:S013"],
  "optional_read": []
}
```

Recompiling the saved public input reproduces the historical SourceMap identifier. The alias range resolves to four `start.md` units: S010, module imports (lines 96–108); S011, `open_workbook` (109–142); S012, `sheet_by_index` (143–156); and S013, `sheet_by_name` (157–170). Thus the packet contains the corresponding source bytes, not merely an instruction to find them. For example, S011 begins:

```
#### 2.  open_workbook() Function - Open an Excel File
```

```
**Function**:  Open a spreadsheet file for data extraction,
supporting `.xls` format files with comprehensive parameter
control.
```

The four adjacent aliases resolve to one exact span covering lines 96–170, packaged with its document, range, and content hash.

The plan reserves `Book` collection behavior (S031–S035) and `Sheet` data access (S014–S021) for later deepening stages. Integration reuses S010 alongside S041–S049 and S051–S057 for imports, workbook usage, errors, and cross-module checks. The same requirement thus supports different stage goals.

C.3 Stage Execution and Public-Check Repair

Stage requests and progression. At stage entry, the controller resolves M_k to exact source spans and includes O_k as reading-card addresses. The goal, instructions, compact execution state X_k, reading card, and packet form one user message; the opening message also supplies the trajectory. Public-source reads remain available.

Only completed turns advance the controller. The compact state X_k records completed-stage identifiers, changed files and workspace inventories, the preceding status, and observed public-check failures. The controller updates this state and saves a workspace checkpoint before advancing. Failed or paused turns stop without automatic retry; resumption is caller-controlled. Turn completion does not certify that every stage goal has been achieved.

Public-check repair. NL2RepoBench permits one additional repair turn after the static trajectory if recognized public-check failures remain. Later success clears a failure in the same command family; probe-scaffold and network-install failures do not trigger repair. The repair turn receives failures and execution state, but no new scheduled packet. Public reads and enabled Projection remain available; remaining failures produce a failed outcome.

RepoGenesis repairs within Module Deepening and Integration rather than appending a repair turn. Final gates check stage completion, condition-specific contract visibility, exact delivery, startup artifacts, declared entrypoints when enabled, and public-source preservation. Neither controller uses final-evaluator feedback for repair.

Coding-stage prompt. This excerpt is the NL2RepoBench E4/E5 Module Deepening branch. Its full message adds workspace restrictions, execution state, and the reading card; RepoGenesis also includes service-startup and public-input preservation instructions.

Prompt 3: Stage execution (Module Deepening)

```
TRP coding stage {stage_position}/{stage_count}:  {stage_id}
(module_deepening).  Goal:  {stage_goal}.  [...]  Deepen the
module/workstream named by this stage.  Implement its exact behavior,
errors, and edge cases without gratuitously rewriting unrelated
modules.  Run focused public-source-derived checks when useful.
[...]  The exact must_read bytes are Framework-delivered in this
message; do not reread the same ranges.  Optional and corrective
source reads remain self-directed.

{exact_current_stage_packet}
Finish this stage naturally and stop; the Framework preserves the
workspace.
```

C.4 Context Projection and Runtime Adapters

Selection and preservation. Projection begins after Broad Pass. At each transition it expires prior source-bearing stage messages and masks eligible completed-stage successful outputs before the next goal and packet arrive. OpenHands selects observation bodies longer than 128 characters using

action and tool-call associations; Codex uses a completed-turn selector. Both preserve failures, timeouts, unknown outcomes, short outputs, unsupported records, and current-stage records.

Expiration replaces the entire prior stage message, including its goal, state, reading card, instructions, and any co-located trajectory. Masking uses identifier-bearing placeholders rather than LLM summaries. Raw histories and workspaces remain unchanged. During coding, recovery relies on permitted reads, checks, or later scheduled redelivery; placeholders are not an agent-callable retrieval interface, and full contracts remain framework-side. Workspace checkpoint restoration is a separate operation, not automatic restoration of retired observations.

Projection example. The following illustrative records use the OpenHands adapter's exact placeholder templates. The `receipt=` field identifies the original record; event identifiers and input bodies are illustrative.

Context projection: before and after (illustrative)

```
1. Eligible completed-stage tool observation (obs-17)
Before: [long successful tool output]
After: [completed-stage successful observation masked;
receipt=obs-17]

2. Previous source-bearing stage message (msg-12)
Before: [goal + state + reading card + exact evidence packet]
After: [previous stage exact-source projection expired;
receipt=msg-12]

3. Failed tool observation (obs-18)
Before and after: ImportError: cannot import name 'BaseAdapter'

4. Current-stage message (msg-19)
Before and after: [current goal + state + reading card + packet]
```

D Additional Results and Cases

The additional results first present functional contrasts, then token accounting and Projection resources, followed by requirement access, implementation trajectories, and task-level effects. Success and failure cases connect requirements to code and test outcomes. Exploratory contract-binding and requirement-length analyses close the appendix. Table 8 distinguishes the reporting scopes; diagnostics and cases use retained runs unless their captions state otherwise.

D.1 Complete-System and Mechanism Contrasts

The Aligned–Shuffled contrast supports intended stage assignment, whereas the access-strategy, timing, and cumulative-component Pass intervals include zero (Table 9). Ordered-component Resolved intervals also include zero. On RepoGenesis, the first cumulative step jointly adds Contract Compilation and Stage Trajectory; Compilation remains enabled thereafter.

Resolved changes across the three cumulative steps are $+2.78$, 0.00, and 0.00 points on NL2RepoBench, with 95% CIs $[0.00, +8.33]$, $[0.00, 0.00]$, and $[0.00, 0.00]$. On RepoGenesis, the corresponding changes are -13.33, $+3.33$, and $+16.67$ points, with 95% CIs $[-28.89, 0.00]$, $[-10.00, +17.78]$, and $[0.00, +33.33]$.

For the auxiliary Resolved outcome, ReqWeaver–OpenHands differences under GPT-5.4-mini are $+2.78$ points on NL2RepoBench (95% CI, $[0.00, +8.33]$) and $+6.67$ points on RepoGenesis ($[-10.00, +23.33]$). Under DeepSeek V4 Pro, they are $+2.78$ points on NL2RepoBench ($[0.00, +8.33]$) and 0.00 points on RepoGenesis ($[-14.44, +14.44]$).

Table 9: Three-run Pass contrasts with GPT-5.4-mini/OpenHands (percentage points). Intervals follow Appendix B.5.

Comparison	Mean difference	95% CI
NL2RepoBench: access, timing, and alignment		
Scheduling − agent-directed reading	+11.25	[−0.56, +23.06]
Scheduling − full-source upfront	+6.74	[−4.54, +18.05]
Scheduling − Stage-wise RAG	+6.30	[−3.38, +15.98]
Staged − upfront (matched content)	+3.29	[−8.45, +14.92]
Aligned − Shuffled	+2.74	[+0.23, +5.24]
NL2RepoBench: cumulative steps		
+ Stage Trajectory	−4.25	[−11.93, +3.43]
+ Evidence Scheduling	+11.25	[−0.56, +23.06]
+ Context Projection	+2.27	[−4.02, +8.44]
RepoGenesis: cumulative steps		
+ Stage Trajectory and Contract Compilation	+5.23	[−4.10, +14.62]
+ Evidence Scheduling	+7.34	[−3.17, +17.87]
+ Context Projection	+10.86	[−0.58, +22.27]

D.2 Token Accounting under a Shared Plan

Token accounting uses the third ReqWeaver run block (456) included in Table 1, covering all 36 NL2RepoBench repositories with GPT-5.4-mini/OpenHands. Each is compared with a control using the same frozen stage plan. The control has autonomous access to the complete requirements, no stage evidence binding, and full history without Projection. ReqWeaver enables evidence binding and Projection. This joint comparison is conditional on the plan. Stage goals, order, and dependencies are fixed within each pair; actual tool actions and call counts may differ. Selection is not restricted to successful generation or parser-accepted evaluation scores.

Coding tokens equal input plus visible output plus reasoning, with cached input included in input and reasoning counted once. The including-planning totals allocate the same successful planning cost of 1,425,309 tokens to each arm. Totals describe these runs and their same-plan controls, including failed records, and exclude cumulative spending on historical experiments and discarded retries. No three-run resource SD or confidence interval is inferred. The ReqWeaver trajectories contribute to the three-run functional results; this accounting reports their single-run token usage. The same-plan control disables binding and Projection while retaining the plan, and differs from the unmodified OpenHands baseline in Table 1.

Table 10: Exact same-plan token totals. Change is ReqWeaver relative to the control, calculated from aggregate counts. The 36-repository scopes include the incomplete `boltons` control; the 35-repository scope includes only pairs where both arms complete generation.

Scope	Same-plan control	ReqWeaver	Change
36 repos, including planning	130,082,041	116,342,765	−10.56%
36 repos, coding only	128,656,732	114,917,456	−10.68%
35 completed pairs, coding only	114,076,935	102,469,853	−10.17%

The `boltons` control reaches the context limit before generation completes; its tokens are consumption up to termination. Restricting accounting to the 32 pairs with parser-accepted scores gives 96,472,407 versus 89,240,256 coding tokens (−7.50%). That score-filtered scope is retained for reconciliation, not used as the primary token comparison. The score-filtered accounting does not replace the full 36-repository token comparison.

Coding input decreases from 127,661,725 to 113,743,955 tokens (−10.90%). Coding output, including reasoning, increases from 995,007 to 1,173,501 (+17.94%). The input reduction outweighs the output increase. Each metric in Figure 3 is normalized by its own control total; Total includes planning, while Input and Output exclude it.

Table 11: Token components for the same 36 selected runs. Output in Figure 3 is visible output plus reasoning. Cached input is already included in input.

Phase	Arm	Input	Visible output	Reasoning
Planning	Control	794,238	36,160	594,911
Planning	ReqWeaver	794,238	36,160	594,911
Coding	Control	127,661,725	909,950	85,057
Coding	ReqWeaver	113,743,955	1,054,671	118,830
Planning + coding	Control	128,455,963	946,110	679,968
Planning + coding	ReqWeaver	114,538,193	1,090,831	713,741

Table 12: Same-plan coding tokens by repository, excluding planning, in units of 10^4 tokens. Values and changes are rounded; aggregate changes use exact totals in Table 10. *The boltons control is incomplete after context-limit termination.

Repository	Same-plan control	ReqWeaver	Change
records	104.62	112.14	+7.19%
requests-html	324.29	109.52	−66.23%
decouple	221.40	113.51	−48.73%
pytorch-grad-cam	287.40	345.66	+20.27%
docopt-ng	209.24	171.80	−17.89%
python-jose	235.65	169.85	−27.92%
bleach	123.26	145.92	+18.39%
pypinyin	450.70	348.13	−22.76%
wsgidav	1,209.95	955.78	−21.01%
jsonlines	179.07	114.37	−36.13%
retrying	85.84	73.29	−14.62%
fuzzywuzzy	83.95	117.22	+39.63%
schedule-master	186.30	119.50	−35.86%
aiofiles	370.15	287.28	−22.39%
pytest-cov	289.13	131.72	−54.44%
xlrd	128.42	138.78	+8.07%
pyjwt	347.31	343.76	−1.02%
deepdiff	355.04	492.44	+38.70%
markupsafe	128.98	126.87	−1.64%
python-slugify	284.72	219.04	−23.07%
markdownify	301.61	197.23	−34.61%
python-dotenv	402.50	392.53	−2.48%
freezegun	302.36	235.83	−22.00%
ftfy	356.51	371.97	+4.34%
cerberus	298.87	180.70	−39.54%
cookiecutter	386.88	495.75	+28.14%
boltons*	1,457.98	1,244.76	−14.62%
pandarallel	113.53	232.65	+104.92%
ipytest	190.79	168.20	−11.84%
sklearn	146.05	179.74	+23.07%
python-fsutil	487.19	338.18	−30.59%
mechanicalsoup	168.33	259.04	+53.89%
databases	684.89	732.18	+6.91%
fastapi-users	352.36	295.66	−16.09%
sqlparse	370.66	762.48	+105.71%
ydata-profiling	1,239.76	768.25	−38.03%
Total (36 repositories)	12,865.67	11,491.75	−10.68%

D.3 Projection-Only Resource Diagnostics

This E4–E5 comparison holds Evidence Scheduling fixed and varies Projection. It is separate from the joint evidence-binding and Projection comparison in Appendix D.2; the two comparisons cannot be combined to decompose the contribution of evidence binding.

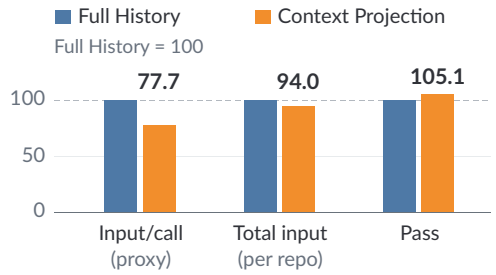

Figure 4: Separate E4–E5 Context Projection comparison on NL2RepoBench (GPT-5.4-mini/OpenHands), normalized to Full History = 100. Pass uses three-run means; input metrics use retained runs.

Table 13 places three-run Pass beside the retained-run resources summarized in Figure 4, separating request size from total generation effort. Figure 5 shows the relationship between input changes and Pass differences within retained task pairs. Resource values are means per repository unless marked per call; accounting follows Appendix B.5. LOC counts nonblank source lines, excluding tests and generated, cache, and build directories.

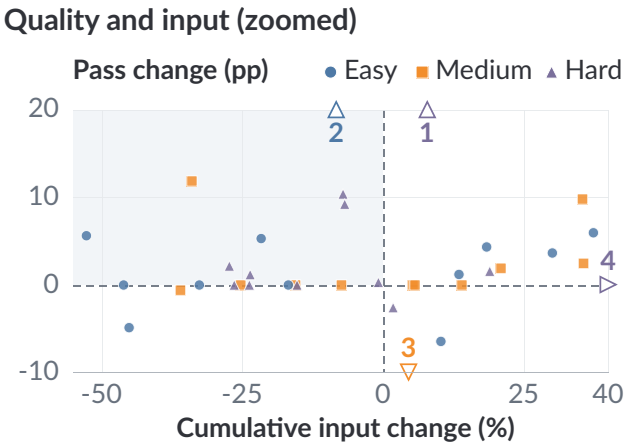

Figure 5: Retained-run Context Projection on NL2RepoBench: zoomed Pass versus input changes. Color and filled shape denote difficulty; numbered hollow arrows mark off-scale tasks, detailed below.

Off-scale observations. Figure 5 displays 32 tasks within $[-55, 40]\%$ input change and $[-10, 20]$ pp Pass change. Boundary arrows identify the other four at their actual (input, Pass) changes: (1) `bleach` ($+7.9\%$, $+38.8$ pp); (2) `markupsafe` (-8.3%, $+46.2$ pp); (3) `mechanicalsoup` ($+4.6\%$, -66.1 pp); (4) `ydata-profiling` ($+82.9\%$, $+0.0$ pp). Nine tasks have higher Pass and lower input, including `markupsafe`. No jitter is applied.

Task-level heterogeneity and failure mode. Projection's three-run Pass differences are $+5.09$, -3.38, and $+5.09$ points on Easy, Medium, and Hard tasks, with 95% CIs $[-7.94, +18.14]$, $[-17.09, +10.28]$, and $[-9.23, +19.43]$. Separately, five retained task pairs lose Pass; the largest loss, `mechanicalsoup`, is examined in Appendix D.7.

Table 13: Context Projection on NL2RepoBench. Pass means, SDs, and the paired Pass difference use the three-run summary; all resource and LOC columns use retained runs.

Context policy	Pass (%)	Input/repo (M)	Input/call (K)	Visible output/repo (K)	Calls/repo	Wall time/repo (min)	LOC/repo
Full History	$44.21_{\pm 2.48}$	10.08	95.0	45.34	106.2	21.42	1,188
Context Projection	$46.47_{\pm 2.49}$	9.48	73.8	49.91	128.5	25.72	1,266
Relative change	+2.27 pp	−6.0%	−22.3%	+10.1%	+21.0%	+20.1%	+6.6%

Input/call decreases from 95.0K to 73.8K. This ratio divides total input by total calls and serves as a proxy for average model-visible input size. The 22.3% reduction in input/call is partly offset by 21.0% more calls, leaving a 6.0% reduction in cumulative input, from 10.08M to 9.48M per repository. Visible output and wall time increase. The output column excludes reasoning and is not the complete token measure used in Appendix D.2. LOC measures implementation scale rather than correctness.

D.4 Requirement Access and Evidence Delivery

We examine whether raw agents access requirements before acting and whether scheduling ensures delivery without replacing agent-initiated reading. The raw-agent analysis covers one OpenHands and one Codex trajectory per task on 36 NL2RepoBench repositories with GPT-5.4-mini. Qualitative inspection identifies planning, runnable-skeleton construction, localized deepening, and integration as recurring regimes, motivating our four operational stages. For Table 14, coverage is the union of `start.md` lines reproduced exactly in supported `cat`/`sed` observations before the first deliverable-code write. Revisitation requires a later exact read to overlap that pre-write coverage.

Table 14: Retained raw-agent requirement access before the first repository write. Coverage measures exact source bytes visible in supported observations.

Harness	N	Mean	Median	Full	$< 50\%$	Revisit
OpenHands	36	63.2%	72.9%	12/36	12/36	0/36
Codex	36	69.4%	100.0%	19/36	12/36	0/36
Combined	72	66.3%	78.4%	31/72	24/72	0/72

Full pre-write coverage is observed in 31 of 72 trajectories, while 24 expose less than half the source. No trajectory revisits an observed range under this definition; a later read of new ranges, as in Codex's `boltons` run, counts as delayed coverage instead. Unsupported read forms may be missed. Exposure is not a measure of semantic use or whether a particular decision lacked prerequisites.

All 591 critical bindings across 223 scheduled E4 stages arrive before the first write, or before completion for stages without writes. An `ipytest` repair stage has no bindings and receives no packet. Agents also make 311 source reads, including 284 after Plan, versus 434 and 353 in E1. Scheduled delivery therefore coexists with self-directed retrieval; these stage-based counts differ from the raw audit's first-write measure.

Implementation before scheduled delivery. One retained `python-fsutil` E4 run (NL2RepoBench, GPT-5.4-mini/OpenHands, full history) illustrates this interaction. The case was selected from an exploratory three-task audit whose tasks were chosen by a fixed hash ordering, not by performance. Broad Pass targets the package shell and core file/path/check interfaces, but the agent also implements hashing and ZIP creation. Table 15 traces their exact source signatures, an independent read, and a confirmed file-creation patch. Earlier packets name both functions but do not include these exact signatures; the corresponding packets arrive in later Module Deepening stages.

This case shows self-directed reading supporting implementation before the corresponding scheduled delivery. It verifies the selected signatures and defaults, not complete functionality or the prevalence of this behavior.

Table 15: Source-to-implementation trace for one retained `python-fsutil` E4 run. Sequence numbers identify records in the saved debug trace.

Evidence	Verified record
Public source	`start.md`:318 specifies `get_file_hash` with `func="md5"`; line 381 specifies `create_zip_file` with `overwrite=True` and `compression=zipfile.ZIP_DEFLATED`.
Independent read	Broad Pass action 33 reads source lines 292–321 and 373–430, among other lines; observation 34 returns both exact signatures.
Initial implementation	Action 47 adds `src/fsutil/info.py` and `src/fsutil/archives.py`; observation 48 confirms both file additions, including function bodies and the specified defaults.
Later delivery	Prompt 253 delivers lines 292–321 at information-utility deepening; prompt 280 delivers lines 373–388 at archive deepening. Both follow the Broad Pass implementation.

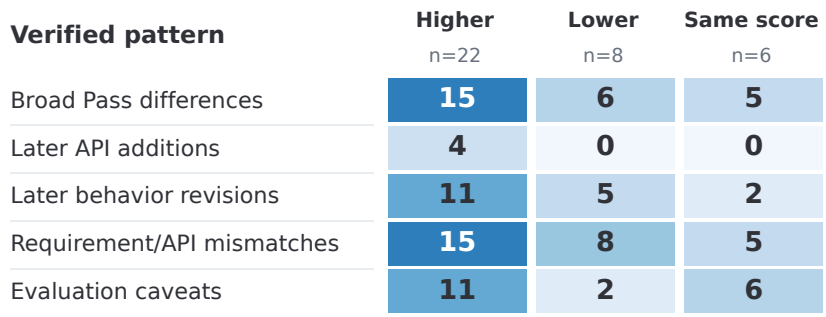

Figure 6: Retained ReqWeaver–OpenHands audit by Pass difference. Counts share a 0–15 color scale. Codes describe the principal evidence chains defined in this subsection and can overlap.

D.5 Implementation Trajectory Audit

We inspect one retained raw OpenHands run (E0) and one complete ReqWeaver run (E5) for each of the 36 NL2RepoBench repositories, using GPT-5.4-mini. Each pair shares its public source, generation parent, and recorded model/environment metadata. These historical diagnostics are separate from the three-run estimates. For each repository, we trace one principal finding through recorded actions, successful or independently confirmed partial writes, code, and evaluator assertions or startup failures. Twelve findings have additional bounded host-environment replay evidence; replay does not replace the official evaluation. The requirement-access and delivery measurements in Appendix D.4 remain separate.

Figure 6 screens every principal evidence chain against five common codes. *Broad Pass differences* locate a relevant implementation difference in the first implementation stage. *Later API additions* identify a named interface, export, or module added after Broad Pass. *Later behavior revisions* identify subsequent semantic changes, including incomplete or harmful revisions rather than only successful repairs. *Requirement/API mismatches* identify a final behavior/API mismatch or implementation blocker. *Evaluation caveats* flag collection/startup/build failures, backend or skip coverage, or timeouts that affect interpretation. Codes can overlap. A zero means no coded positive within these principal chains, not that the behavior was absent from all trajectories; this is not an exhaustive prevalence estimate. Table 16 lists all assignments.

Across 36 retained pairs, 15 of 22 higher-scoring repositories have Broad Pass differences, but 15 retain requirement/API mismatches. Later revisions also occur in five of eight lower-scoring repositories. These overlapping patterns describe implementation changes without isolating component effects or additional computation.

Table 16: Principal-chain coding for Figure 6. B: Broad Pass difference; A: later API addition; R: later behavior revision; P: requirement/API mismatch; E: evaluation caveat. +/−/= denote higher/lower/unchanged retained Pass, not causal effects. Unlisted codes are not absence claims.

Repository	Direction	Codes	Repository	Direction	Codes
aiofiles	+	A	pandarallel	+	BRE
bleach	−	BRP	pyjwt	+	ARP
boltons	+	BR	pypinyin	+	BE
cerberus	+	BRP	pytest-cov	=	BPE
cookiecutter	−	RP	python-dotenv	−	BP
databases	−	BPE	python-fsutil	−	BRP
decouple	+	BRP	python-jose	+	BAPE
deepdiff	+	BPE	python-slugify	+	RP
docopt-ng	+	RPE	pytorch-grad-cam	=	BRE
fastapi-users	=	PE	records	−	BP
freezegun	=	BRPE	requests-html	+	BPE
ftfy	+	BE	retrying	+	BRP
fuzzywuzzy	−	BRPE	schedule-master	+	BPE
ipytest	−	RP	sklearn	+	BR
jsonlines	+	BP	sqlparse	+	RP
markdownify	+	R	wsgidav	+	BPE
markupsafe	+	APE	xlrd	+	BPE
mechanicalsoup	=	BPE	ydata-profiling	=	BPE

Columns preserve the historical score direction: 22 higher, eight lower, and six unchanged. Equal Pass scores need not imply identical implementations, and higher-scoring repositories can retain requirement/API mismatches.

The chains distinguish several routes to outcome differences. In `python-slugify`, a Module Deepening check exposes a separator/truncation error, a patch changes the operation order, and repeated checks pass; bounded replay confirms the selected behavior persists. In `jsonlines`, per-record prefix handling already differs during Broad Pass. Conversely, `fuzzywuzzy` introduces an optional-dependency binding error during Broad Pass that later work does not eliminate. These explain specific behaviors, not every test difference. ReqWeaver uses more recorded model calls in all 36 pairs (4,625 versus 1,684 in total), so this audit does not distinguish component benefits from additional computation.

D.6 Task-Level and Difficulty-Stratified Effects

Figure 7 and Table 17 show heterogeneous retained-run effects. Scheduling improves 20 of 36 tasks over E1, with a median gain of $+1.10$ points. Matched-content timing has a negative retained-run median and more losses than wins, contrasting with the positive three-run mean in the main results.

Table 17: Retained-run paired Pass differences on 36 NL2RepoBench repositories. Medians are in percentage points; W/T/L counts positive, zero, and negative differences.

Contrast	Median pp	W/T/L
Evidence Scheduling increment (E4−E1)	$+1.10$	20/7/9
Scheduled vs. upfront source (E4−E2)	0.00	14/8/14
Matched-content timing (T0−TU)	-1.20	13/5/18
Context Projection increment (E5−E4)	$+0.15$	18/13/5
Scheduling vs. raw agent (E4−E0)	$+1.38$	19/7/10
Full method vs. raw agent (E5−E0)	$+4.43$	22/6/8

The historical retained-run E4−E0 diagnostic has a mean Pass difference of $+7.01$ points (task-level 95% CI, $[-1.40, +15.05]$); this is not an additional three-run comparison. In the retained Projection pairs, five repositories lose Pass: `mechanicalsoup`, `records`, `requests-html`, `pypinyin`, and `ftfy`. The largest loss is -66.12 points on `mechanicalsoup`; Appendix D.7 examines its failure path.

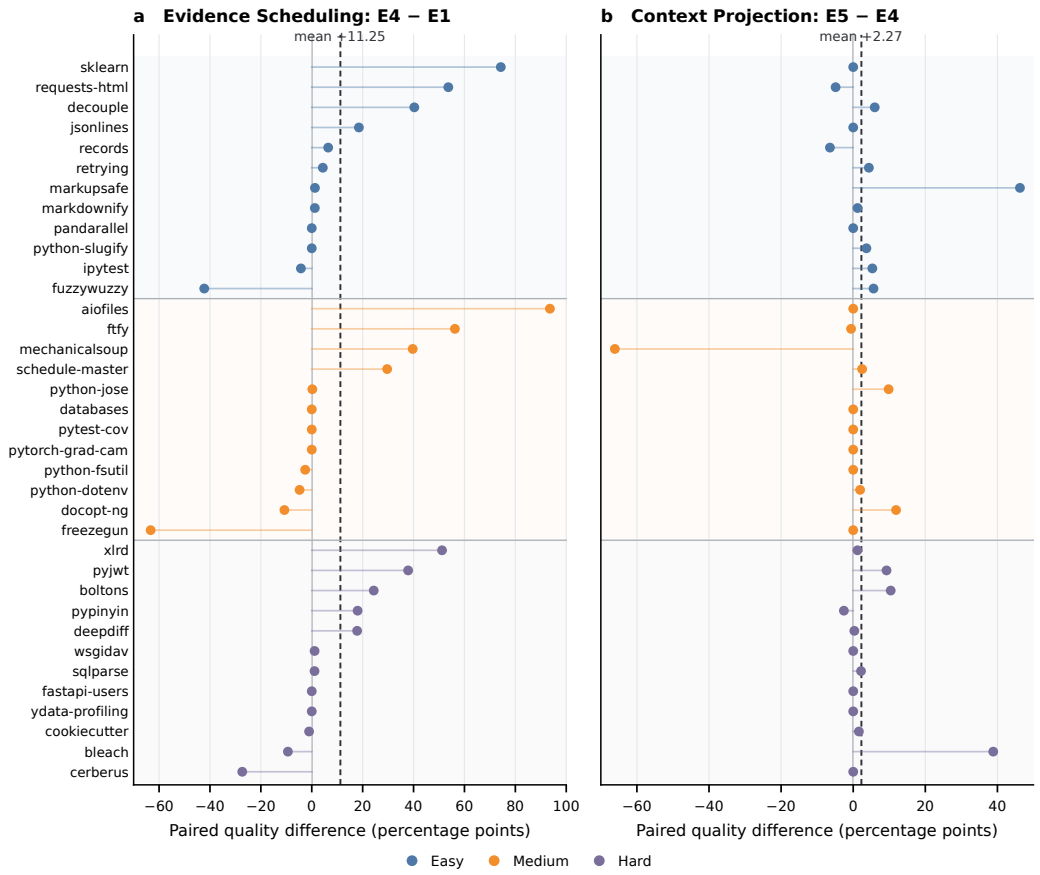

Figure 7: Retained-run paired effects on 36 NL2RepoBench repositories. Evidence Scheduling has a positive but heterogeneous E4−E1 effect (left); Context Projection has a smaller E5−E4 effect with five negative task-level differences (right). Colors denote benchmark difficulty, and dashed lines show retained-run task-macro means.

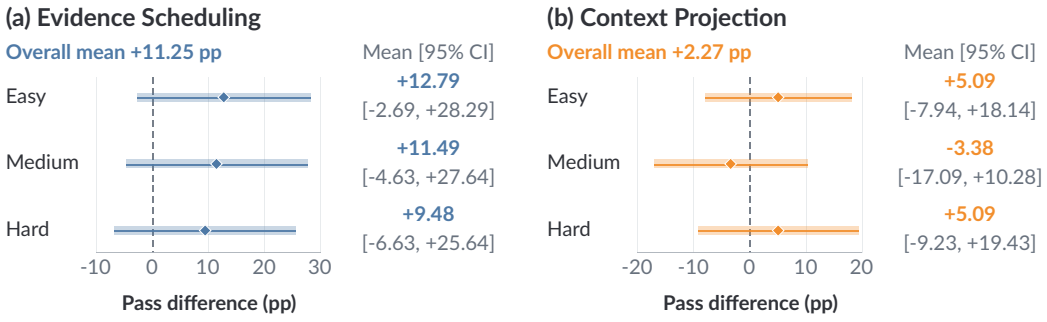

Figure 8: Three-run difficulty-stratified Pass effects on NL2RepoBench (GPT-5.4-mini/OpenHands): (a) Scheduling, E4−E1; (b) Projection, E5−E4. Diamonds and bands show means and crossed-bootstrap 95% CIs. Both axes span 40 percentage points; dashed lines mark zero.

Figure 8 reports three-run Scheduling increments of $+12.79$, $+11.49$, and $+9.48$ points for Easy, Medium, and Hard tasks, with 95% CIs $[-2.69, +28.29]$, $[-4.63, +27.64]$, and $[-6.63, +25.64]$. All include zero.

Table 18 combines three-run Pass summaries with retained-run resources. Projection lowers cumulative input on Easy and Medium tasks but not Hard tasks; only Medium has a negative three-run mean Pass change. These separate scopes do not estimate a paired multi-run quality–resource relationship.

Table 18: Difficulty-stratified summaries with separate scopes: Pass differences are three-run task-macro means in percentage points; input and calls are retained-run changes from E4 to E5. Input is the stratum-level cumulative percentage change; calls are counts.

Stratum	N	E4−E1	E5−E4	Input Δ	Calls Δ
Easy	12	+12.79	+5.09	−17.7%	+112
Medium	12	+11.49	−3.38	−8.7%	+282
Hard	12	+9.48	+5.09	+0.5%	+409

D.7 Success and Failure Cases

We select one large scheduling gain and the largest projection loss from the retained runs to illustrate contrasting execution paths, not to estimate their frequency across runs. Both use GPT-5.4-mini and OpenHands; the comparisons are E4 versus E1 for `xlrd` and E5 versus E4 for `mechanicalsoup`.

Evidence Scheduling on `xlrd`. The evidence chain below connects a public requirement to the delivered stage view, the resulting implementation difference, and final test outcomes. E1 uses Stage Trajectory without scheduled packets; E4 adds Evidence Scheduling.

Evidence link	Observed requirement, implementation, or outcome
Public requirement	Open binary `.xls` files and expose their worksheets. Source units S010–S013 cover imports, `open_workbook`, `sheet_by_index`, and `sheet_by_name`.
Stage delivery	E4 receives the exact `start.md` span at lines 96–170 at Broad Pass entry, before coding. The saved plan and source mapping appear in Appendix C.2.1.
E1 implementation	`open_workbook` recognizes the file type but assigns an empty worksheet list to the `.xls` branch. The returned `Book` has no sheets; downstream lookup raises `XLRDError`.
E4 implementation	Extracts the OLE `Workbook` stream, falls back to `Book`, and parses workbook globals and worksheets before returning the object.
Worksheet tests	E1: 0/24 passed; E4: 23/24 passed. The remaining E4 failure concerns `cell_xf_index`.
Repository Pass	E1: 18/84 (21.43%); E4: 61/84 (72.62%).

This retained pair links scheduled requirements to a substantive parsing implementation rather than an API stub. It illustrates the combined Scheduling intervention; the source-to-packet construction appears in Appendix C.2.1.

D.7.1 Illustrative Stage-Assignment Case: `deepdiff`

In the retained `deepdiff` pair, Aligned delivery precedes implementation of a shared helper module that is absent from the Shuffled repository. Aligned passes 305/980 tests (31.12%), versus 21/970 (2.16%) for Shuffled, using the observed evaluator denominators. This historical case is separate from the three-run Aligned–Shuffled effect estimate.

From packet assignment to implementation. The Broad Pass packet P_2 spans source lines 206–2282, beginning with public module imports. Lines 233–240 explicitly import shared types and utilities from `deepdiff.helper`. Aligned receives this packet before constructing the package; Shuffled receives the Delta/path packet P_5 at Broad Pass and receives P_2 one stage later, during value-comparison deepening. Integration reuses S010 from P_2 alongside new cross-module requirements, illustrating reuse for a different stage goal (Table 19).

In the Aligned Broad Pass trace, action 33 creates the module set including `helper.py`, and action 35 writes its implementation. The final Shuffled package lacks `deepdiff/helper.py`;

Table 19: Complete seven-stage `deepdiff` schedule and packet reassignment. P_i denotes the intact packet originally assigned to row i; Shuffled changes packet donors, not stage goals or order. Sizes are source characters in thousands, excluding wrappers. Goal descriptions are shortened from the saved plan.

Stage	Goal	Aligned source aliases	Aligned packet (K)	Shuffled packet (K)
1	Plan module ownership and dependencies	S001–S005, S007	P_1: 11.45	P_4: 4.54
2	Build package structure and public API	S010–S017	P_2: 75.64	P_5: 14.90
3	Implement value/container comparisons	S209–S213	P_3: 8.52	P_2: 75.64
4	Implement datetime, filtering, typing, and caching	S214–S218	P_4: 4.54	P_6: 4.48
5	Implement Delta and path operations	S170, S221, S234–S235	P_5: 14.90	P_1: 11.45
6	Implement search, hashing, and serialization	S188–S189, S219, S222–S223	P_6: 4.48	P_7: 33.18
7	Integrate exports and cross-module behavior	S010, S206–S207, S224–S233	P_7: 33.18	P_3: 8.52

its evaluator output contains 12 missing-module collection errors naming that module. Neither arm records an additional source read. The trace links earlier package-level evidence with shared-module coverage in this run.

D.7.2 Other Failure Cases

Context Projection on `mechanicalsoup`. Successful integration requires the generated library to coexist with its installed dependencies. The E5 workspace instead contains a local `requests` package. During evaluator startup, `requests_mock` imports `BaseAdapter`, but Python resolves `requests.adapters` to the local `requests/adapters.py`, which lacks that class. Pytest stops during plugin import, before test execution. E4 passes 80/121 tests (66.12%); E5 receives zero under the failure-scoring policy, not 121 observed assertion failures. The trace identifies dependency shadowing as the immediate failure, not evidence that projection removed a necessary observation. It motivates final package-boundary checks alongside stage-local context control.

External-system failure cases. The three-run external-baseline results include code incompatibility, startup/collection blockers, and workflow failures. In representative GPT-5.4-mini runs, ZeroRepo's `records` lacks expected `Database`, `Record`, and `RecordCollection` attributes, and `markupsafe` cannot import `_native`. Zero2Paper's RepoGenesis `Blog` completes generation but fails service startup and test collection. MetaGPT likewise has low NL2RepoBench Pass despite completed generation and evaluation; DeepSeek runs also contain MetaGPT workflow aborts and Zero2Paper output-parsing failures. These cases explain why workflow completion need not imply benchmark compatibility, without attributing all low scores to adaptation or infrastructure.

D.8 Exploratory Contract-Binding Analysis

A separate development comparison on 30 RepoGenesis tasks with GPT-5.4-mini and OpenHands holds the joint trajectory, public-source bindings, and post-Broad-Pass projection policy fixed while removing contract bindings from one arm. This tests delivery of compiled evidence, not removal of the compiler from preflight. The base matched results are 77.61% Pass without bindings and 76.14% with them, providing no positive point estimate for the added bindings in this comparison.

D.9 Requirement-Length Analysis

D.9.1 Grouping and Main Findings

We tokenize the original public requirements with `o200k_base` for retained GPT-5.4-mini/OpenHands pairs: 36 NL2RepoBench and 30 RepoGenesis tasks. Source hashes are matched across compared runs. RepoGenesis length counts README content only; neither benchmark includes generated artifacts or interaction history. Tasks are sorted within each benchmark into equal-sized token tertiles, independently of outcomes. Table 20 and Figure 9 report ranges and counts.

Group means equally weight E5−E0 task differences. Pointwise 95% CIs use 10,000 paired-task bootstrap resamples within each fixed group (seed: 20260907), with percentile endpoints and linear interpolation. All selected tasks remain included under the evaluation policy.

Table 20: Requirement-length breakdown of retained GPT-5.4-mini/OpenHands results. Each benchmark is divided into three equal-sized groups by original document token count, with non-overlapping ranges. Pass is in percent; Δ is ReqWeaver minus OpenHands in percentage points. Intervals are pointwise paired task-bootstrap 95% CIs, not across-run uncertainty.

Group	n	Length (tokens)	OpenHands	ReqWeaver	Δ [95% CI]
NL2RepoBench					
Short	12	3,612–7,638	51.47	65.25	$+13.78$ $[+2.81, +25.30]$
Medium	12	8,983–16,330	42.65	43.62	$+0.97$ $[-14.95, +13.92]$
Long	12	16,445–90,503	17.48	30.55	$+13.07$ $[+4.47, +23.47]$
RepoGenesis					
Short	10	358–1,139	62.73	80.17	$+17.44$ $[-20.00, +52.92]$
Medium	10	1,244–1,752	56.04	72.39	$+16.35$ $[-23.44, +54.64]$
Long	10	1,768–3,364	46.33	82.80	$+36.47$ $[+12.31, +61.93]$

Results. Mean gains are positive in all six groups. NL2RepoBench's Short and Long gains are similar ($+13.78$ and $+13.07$ points), while Medium is near zero ($+0.97$). RepoGenesis's Long group has the largest mean gain ($+36.47$ points), but its Short and Medium intervals span zero. Benefits are therefore not confined to long documents, and there is no consistent increasing trend across benchmarks.

Length groups overlap unevenly with difficulty: NL2RepoBench's Short/Medium/Long groups contain 8/3/1, 4/6/2, and 0/3/9 Easy/Medium/Hard tasks; RepoGenesis has 8/2/0, 2/5/3, and 1/4/5 under its proxy. All RepoGenesis documents are shorter than the shortest NL2RepoBench input, so length-group names are benchmark-relative.

D.9.2 Difficulty Adjustment

Long-minus-Short mean differences are -0.71 points on NL2RepoBench (95% CI, $[-15.28, +14.13]$) and $+19.03$ on RepoGenesis ($[-23.75, +62.28]$), using the fixed-group bootstrap. To adjust for difficulty, we separately fit each benchmark with equally weighted OLS. For task gain $\Delta_i = 100(\mathrm{Pass}_{E5,i} - \mathrm{Pass}_{E0,i})$ and centered log-length $\ell_i = \log_2(L_i) - \overline{\log_2(L)}$,

$$\Delta_i = \alpha + \beta\ell_i + \gamma_{\mathrm{M}}\mathbf{1}[D_i = \mathrm{Medium}] + \gamma_{\mathrm{H}}\mathbf{1}[D_i = \mathrm{Hard}] + \varepsilon_i. \tag{6}$$

Here β is the gain difference per doubling of length, conditional on difficulty, with Easy as reference. The unadjusted model omits difficulty indicators. Both use the same 10,000 paired-task bootstrap draws within difficulty strata (seed: 20260907); intervals are pointwise and not multiplicity-adjusted.

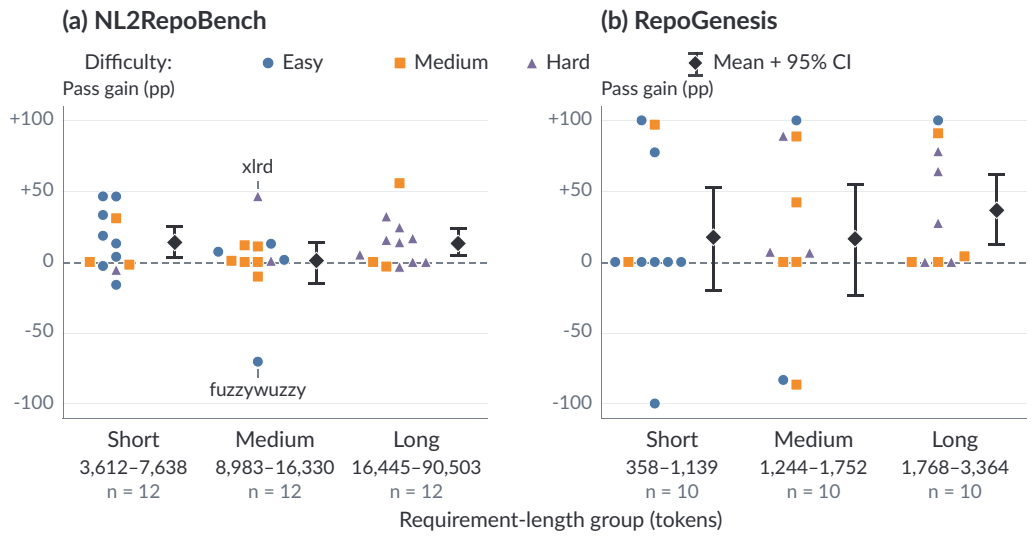

Figure 9: Retained-run E5−E0 Pass gains by original requirement length on NL2RepoBench and RepoGenesis. Shapes and colors encode difficulty (a public-test-load proxy for RepoGenesis); diamonds and bars show length-group means and pointwise paired task-bootstrap 95% CIs. Horizontal offsets separate coincident marks. Highlighted tasks support the Medium-group sensitivity analysis.

Table 21: Exploratory length–gain slopes (percentage points per doubling of original document length). Brackets are pointwise 95% task-bootstrap CIs. Adjusted OLS includes difficulty-category indicators; VIF describes length–difficulty collinearity, not causal identification.

Benchmark	n	Unadjusted [95% CI]	Adjusted [95% CI]	VIF
NL2RepoBench	36	$+0.45$ $[-5.25, +6.38]$	-1.48 $[-11.73, +7.52]$	2.05
RepoGenesis	30	$+1.13$ $[-20.12, +28.81]$	-7.35 $[-55.50, +39.94]$	1.86

Table 22: Length-group stability using all retained task pairs. Mean, median, and leave-one-out (LOO) means are E5−E0 Pass differences in percentage points. W/T/L counts task-level wins/ties/losses. LOO ranges omit each task once with fixed group membership; they are not confidence intervals.

Group	n	Mean	Median	W/T/L	LOO mean range
NL2RepoBench					
Short	12	$+13.78$	$+8.35$	7/1/4	$[+10.82, +16.50]$
Medium	12	$+0.97$	$+1.15$	8/2/2	$[-3.16, +7.46]$
Long	12	$+13.07$	$+9.55$	7/3/2	$[+9.20, +14.57]$
RepoGenesis					
Short	10	$+17.44$	$+0.00$	3/6/1	$[+8.27, +30.49]$
Medium	10	$+16.35$	$+6.83$	6/2/2	$[+7.05, +27.80]$
Long	10	$+36.47$	$+15.77$	6/4/0	$[+29.41, +40.52]$

Adjustment changes both slope estimates from positive to negative, but all four CIs include zero (Table 21). Difficulty explains 51.3% and 46.3% of log-length variance, with VIFs 2.05 and 1.86. Leave-one-out adjusted slopes range from -4.01 to $+0.87$ and -24.65 to $+6.63$, respectively. Sparse overlap and the common log-linear slope assumption limit interpretation; these results establish neither a causal length effect nor equivalence to zero.

The original NL2RepoBench difficulty-relative selection groups provide a separate check, with pooled Short/Medium/Long gains $+11.84$/$+9.48$/$+6.51$ points. Within Easy they

are $+20.45/+15.05/-12.21$; within Medium, $+4.66/+2.94/+16.05$; and within Hard, $+10.41/+10.44/+15.69$. Each cell contains four tasks and is resampled within its original group. This grouping also shows no uniform length trend.

D.9.3 Sensitivity to Individual Tasks

Table 22 reports medians, W/T/L counts, and leave-one-out mean ranges with group membership fixed. NL2RepoBench's Medium group has eight wins, two ties, and two losses, with median $+1.15$ points. Its extremes are `fuzzywuzzy` (-70.42) and `xlrd` ($+46.43$). Omitting either changes the mean from $+0.97$ to $+7.46$ or -3.16, respectively. The near-zero mean reflects heterogeneous outcomes rather than a general medium-length failure regime; these omission ranges are sensitivity summaries, not confidence intervals.