

Super Library Agent: Joint Generation and Maintenance of Multiple Applications Beyond the Single Codebase

Daeyu Sung^{1*}, Yukyeong Lee^{1*}, Geon Park¹, Yumin Choi¹, Sung Ju Hwang^{1,2†}

¹KAIST, ²DeepAuto.ai

{sbigstar0310, yukyeong, geon.park, yumin.choi, sungju.hwang}@kaist.ac.kr

Abstract

Organizations often develop and maintain *portfolios* of related applications: independently deployable codebases that share substantial domain logic, interface patterns, or operational conventions. As LLM coding agents are increasingly used to generate and maintain such software, a naive application-by-application workflow duplicates shared logic across codebases and allows prolonged agentic maintenance to accumulate verbosity, dead code, and structural erosion. We introduce the **Super Library Agent** problem, where an agent sequentially generates a portfolio of N related applications while maintaining a shared *Super Library* of reusable cross-application components. A minimal sequential scaffold can in principle extract shared code and migrate applications to the evolving library, but in practice suffers from low extraction recall and fragile dependency migration. We address these failures with candidate-guided extraction over code chunk summaries, pre-extraction codebase consolidation, and context-aware migration using extraction traces and call-graph information. Across WebGen-Bench and PaperBench, our method preserves application functionality while significantly reducing redundancy and token footprint (verbosity, token length) over zero-shot, and avoiding the structural erosion introduced by naive library construction, with additional reductions in LOC and MDL. Our code is available at <https://github.com/sbigstar0310/super-library-agent>.

1 Introduction

LLM coding agents can now generate non-trivial applications end-to-end (Qian et al., 2024; Hong et al., 2024; Dong et al., 2024), but real-world software development rarely produces a single application in isolation. Organizations often maintain *portfolios* of related applications (Clements and

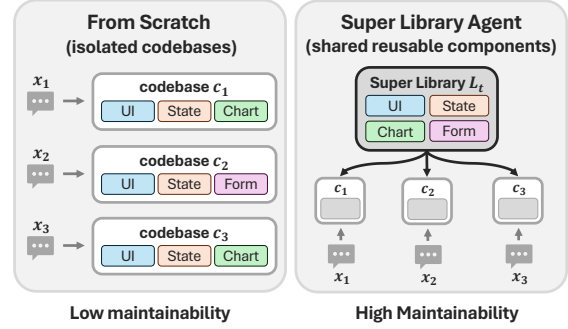


Figure 1: **Super Library Agent (SLA) problem.** Independent generation duplicates shared logic; SLA maintains a shared library across a sequence of codebases.

Northrop, 2001; Pohl et al., 2005): independently deployable codebases that share domain logic, interface patterns, data-processing routines, or operational conventions. Generating each portfolio member independently duplicates these shared concerns, violating DRY principles (Thomas and Hunt, 2019) and creating an inconsistency surface where every bug fix, design change, or policy update must be manually propagated across near-duplicates. Such duplication is already a well-known maintenance burden in software portfolios (van der Zwaag et al., 2025; Roy and Cordy, 2007; Juergens et al., 2009); LLM-generated portfolios make the problem easier to create at scale.

Furthermore, the issue is compounded by the maintenance behavior of LLM coding agents. LLM code generators are known to emit measurable code smells (Siddiq et al., 2022; Liu et al., 2026). SlopCodeBench (Orlanski et al., 2026) shows that prolonged agentic coding can produce *slop*: code that remains functionally correct while becoming increasingly verbose and structurally convoluted. Generating and maintaining a portfolio of N applications with N isolated coding agents compounds the effect: each agent re-derives shared logic in subtly different forms, accumulates its own dead code, and drifts further from its siblings with ev-

* Equal contribution. † Corresponding author.

ery maintenance cycle. A method that explicitly factors and re-factors shared concerns across the portfolio is therefore an essential structural defense against agent-induced slop accumulation.

Library learning with LLMs originated as a way to amortize reasoning and tool-creation costs across related tasks (Ellis et al., 2021; Bowers et al., 2023; Berlot-Attwell et al., 2026; Stengel-Eskin et al., 2024; Wang et al., 2023; Cai et al., 2024). Recent work, such as Librarian (Kovacic et al., 2025), assumes the input files live inside one logical project, and the authors explicitly leave “*large-scale multi-repo library creation*” as an open problem.

We introduce the *Super Library Agent* problem. Given a sequence of related application requests, an agent must generate each application while maintaining a shared *Super Library* of components reused across applications. A naive baseline generates each application independently, duplicating shared concerns. A stronger sequential scaffold generates applications one by one, extracts shared components into the Super Library, and migrates affected applications to use the updated library. Although this scaffold captures the intended functionality–maintainability trade-off, it fails in practice for two main reasons.

Challenge 1: Extraction Recall. The agent must discover reusable components across an expanding portfolio of codebases. This is difficult because functionally equivalent implementations may differ substantially in syntax and structure (Svajlenko et al., 2014; Saini et al., 2018). Retrieval mechanisms that rely primarily on surface similarity may therefore overlook valid reuse candidates. As the portfolio grows, these missed candidates increasingly limit the coverage and reuse of the Super Library.

Challenge 2: Migration Correctness. Extraction is not a local edit. Once a local implementation is moved into the Super Library, applications must update imports, call sites, dependent helper functions, and sometimes surrounding APIs. A naive migration agent may replace the visible local implementation but miss non-local dependency updates, leaving broken references, duplicated implementations, or dead code behind. Thus, reliable Super Library construction requires both discovering reusable abstractions and safely propagating their use through application codebases.

We propose an augmented Super Library Agent scaffold that targets these two failures. For ex-

traction recall, we build summarized code indexes and use an LLM-based selector to propose explicit cross-application reuse candidates, rather than relying only on implicit discovery. For migration correctness, we provide the dependency-migration agent with extraction traces and call-graph context, making explicit which local blocks were consolidated, which library components should replace them, and which surrounding call sites or dependencies must be updated.

We evaluate our method on WebGen-Bench (Lu et al., 2025) and PaperBench (Starace et al., 2025), reporting accuracy and various code maintainability metrics. Our contributions are:

- We formulate the Super Library Agent problem, in which an agent generates a sequence of related applications while maintaining a shared library of reusable cross-application components.
- We identify two key obstacles to naive library construction: low extraction recall and fragile dependency migration. We address them with candidate-guided extraction and context-aware migration, respectively.
- We evaluate on WebGen-Bench and PaperBench, showing that our method preserves functionality while improving maintainability metrics, library utilization, and abstraction quality over other baselines.

2 Related Work

Maintainable code generation. Recent code generation work increasingly treats maintainability as an explicit objective. MaintainCoder (Wang et al., 2025b) studies changing requirements, while SlopCodeBench (Orlanski et al., 2026), CodeTaste (Thillen et al., 2026), and Agentic Refactoring (Horikawa et al., 2025) show that agents can preserve behavior while accumulating verbosity, duplication, and architectural debt, and often perform only local cleanups. These works mainly study a single evolving codebase. In contrast, SLA targets a portfolio of independently deployable applications whose maintainability depends on the joint codebase: the shared library and all applications that import it.

Library learning and post-hoc refactoring. Library learning typically mines abstractions from fixed code corpora, from DreamCoder (Ellis et al., 2021) and Stitch (Bowers et al., 2023) in program synthesis to LILO (Grand et al., 2024)’s LLM-assisted interpretable libraries. However, Berlot-

Attwell et al. (2026) shows that LLM library-learning gains can be confounded by extra inference budget and that learned artifacts are often rarely reused. Kovacic et al. (2025) brings this line closer to software engineering by framing library construction as MDL-minimizing refactoring over existing code snippets. These methods treat library construction as a post-hoc compression or refactoring step over programs that already exist. SLA instead studies an online setting: applications arrive sequentially, the shared library is used when generating each new application, and updated abstractions are migrated back to prior applications.

Common-logic extraction and migration. Existing systems extract common logic from completed codebases through corpus-level grouping or explicit refactoring workflows (Meng et al., 2015; Yue et al., 2018; Pomian et al., 2024). RefAgent (Oueslati et al., 2025) decomposes project refactoring into planning, execution, and repair. These systems show that discovery and migration need more structure than direct rewriting, but they do not address online cross-application abstraction: SLA extracts code blocks from natural-language summaries, synthesizes parameterized APIs by turning codebase-specific values into arguments, and records source blocks in an extract map for migration back to affected applications.

3 Methods

3.1 Problem Definition

We introduce the **Super Library Agent** problem, a novel challenge for coding-specific agentic workflows that jointly generate and maintain N related applications with a shared *Super Library*. Unlike single-codebase generation, the agent must implement each new application while identifying reusable components across existing codebases and maintaining them through a common library.

Let $\mathbf{X} = (x_1, \dots, x_N)$ be a sequence of application implementation requests. The agent starts with an empty Super Library $\mathcal{L}_0 = \emptyset$. At each step t , given a new request x_t , previous codebases $\mathbf{C}_{<t} = \{c_1, \dots, c_{t-1}\}$, and the current library $\mathcal{L}_{t-1}$, the agent $\mathcal{A}$ produces

$$(c_t, \mathbf{C}'_{<t}, \mathcal{L}_t) = \mathcal{A}(x_t, \mathbf{C}_{<t}, \mathcal{L}_{t-1}),$$

where c_t is the new codebase, $\mathcal{L}_t$ is the updated Super Library, and $\mathbf{C}'_{<t}$ denotes previous codebases patched to use the updated library.

The ideal Super Library contains every component used by at least two implemented applications:

$$\mathcal{L}_t^* = \left\{ u \mid \sum_{i=1}^t \mathbf{1}[\text{use}(u, c_i)] \geq 2 \right\}.$$

Thus, application-specific components remain local, while shared components are extracted and consistently reused.

We consider a *sequential portfolio construction setting*, where applications arrive one at a time with a single initial request and no subsequent patches. The goal is to seek a **favorable Pareto trade-off between the functionality and joint-codebase maintainability** of the final applications $\mathbf{C}_N$ and Super Library $\mathcal{L}_N$:

$$\max_{\mathcal{A}} (S_{\text{func}}(\mathbf{C}_N, \mathbf{X}), S_{\text{maint}}(\mathbf{C}_N, \mathcal{L}_N)).$$

Here, functionality measures whether applications satisfy their requests, while maintainability measures whether shared components are properly extracted, deduplicated, reused, and updated.

3.2 A Minimal Agentic Scaffold

We instantiate Super Library Agent with a minimal sequential scaffold that maps $(x_t, \mathbf{C}_{<t}, \mathcal{L}_{t-1})$ to $(c_t, \mathbf{C}'_{<t}, \mathcal{L}_t)$ at each step t using two specialized phases: coding and library maintenance.

First, the *coding agent* generates the new codebase c_t^0 from x_t , reusing components from $\mathcal{L}_{t-1}$ when applicable. Second, a single *library agent* jointly extracts components shared across the implemented applications into the Super Library, yielding $\mathcal{L}_t$, and migrates previous codebases that contain duplicated local implementations or deprecated library dependencies, yielding c_t and $\mathbf{C}'_{<t}$.

This minimal scaffold assigns both Super Library construction and cross-codebase dependency migration to a single general-purpose library agent, without dedicated candidate selection or a structured hand-off between the two tasks. We provide the full procedure in Appendix G; our experiments in Section 4 use this scaffold as the SLA-NAIVE comparison baseline.

3.3 Library Extraction and Dependency Migration Strategies

The minimal scaffold in Section 3.2 has two practical limitations: (i) low recall in identifying reusable components across growing codebases, and (ii) the risk of breaking existing applications when migrating them to newly extracted library components.

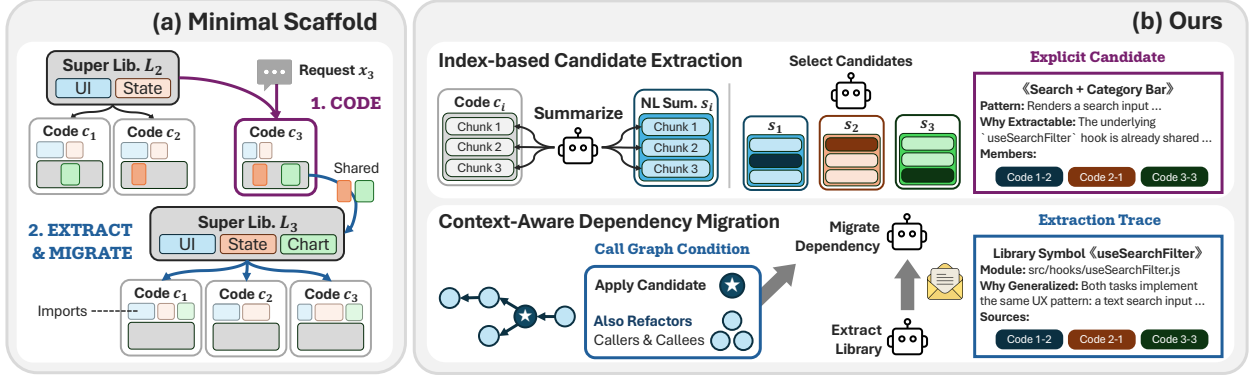


Figure 2: **Overview of the proposed workflow.** Minimal scaffold SLA-NAIVE (left) vs. SLA-FULL (right).

To address these limitations, we decompose the single library agent into two specialized agents: a *library-extraction agent* responsible for updating the shared library and a *dependency-migration agent* responsible for patching previous codebases. We then equip them with two complementary strategies: (i) index-based candidate extraction and (ii) context-aware dependency migration. See Algorithm 2 for our augmented scaffold.

3.3.1 Index-based Candidate Extraction

Functionally equivalent code blocks often share little surface form, so grouping them by code text or embedding similarity overlooks valid candidates. We therefore summarize each block in natural language and select over these compact *code-summary indexes*, matching blocks by what they do rather than by how they are written, for both library extraction and dependency migration.

Code Block Indexing. We maintain an up-to-date code-summary index for every application codebase and the Super Library. Each entry records the location of a semantically meaningful code block and its concise natural-language summary. We identify code blocks using abstract syntax tree (AST) boundaries, such as functions, classes, and modules,¹ and generate their summaries with an LLM.²

LLM-Based Candidate Selection. A dedicated candidate selector uses these indexes for two matching tasks.³ The library-extraction agent receives *extraction candidates*, code blocks from multiple applications that implement shared functionality.

¹We use cocoindex-code. <https://github.com/cocoindex-io/cocoindex-code>

²gpt-5.4-nano with medium reasoning effort is used.

³The candidate selector uses deepseek-v4-flash with high reasoning effort.

The dependency-migration agent receives *migration candidates*, library components paired with the application-local blocks they can replace. Detailed prompts and example outputs are provided in Appendix H.3.

Pre-Extraction Codebase Consolidation. Before cross-application candidate selection, we perform a one-time consolidation of each newly generated codebase. Using the same index-based selection procedure, the agent identifies duplicated or overlapping local implementations and refactors them into shared modules within the application. The consolidated codebase is then indexed for Super Library extraction.

3.3.2 Context-Aware Dependency Migration

Dependency migration is challenging because extraction and migration are performed by separate agents, and because replacing a local implementation requires coordinated updates to its import declarations and call sites. To address these challenges, we use extraction traces to bridge the two agents and call-graph context to guide dependent edits.

Extraction-Trace-Based Bridging. After each library extraction, the library-extraction agent creates a structured extraction trace for every new or updated Super Library symbol. Each trace records where the symbol came from, why the pattern generalizes across applications, and how to replace the original code with the library API. Passing this trace to the dependency-migration agent bridges the two phases without requiring it to rediscover the correspondence from scratch. Appendix H.4 provides an example.

Call Graph Conditioning. For each migration candidate, we attach an *also-refactor* list to the mi-

gration prompt containing the relevant import declarations and local caller/callee relationships (Ryder, 1979). This context directs the agent to update dependent call sites and imports while removing obsolete local implementations, reducing broken references, duplication, and dead code (Eder et al., 2012).

4 Experiments

4.1 Experimental Setup

4.1.1 Benchmarks

We evaluate our method on two coding domains: WebGen-Bench and PaperBench. To adapt existing single-task coding benchmarks to the Super Library Agent setting, we sample disjoint task sequences of length N and treat each sequence as an evaluation suite. Given a suite $X = (x_1, \dots, x_N)$, the agent sequentially implements each task while maintaining a shared Super Library across the generated applications.

WebGen-Bench. WebGen-Bench (Lu et al., 2025) evaluates website generation from natural-language instructions using executable functionality tests and visual-quality judgments. We construct three disjoint 8-task suites. Because web applications share UI components, layout primitives, styling patterns, and interaction logic, this domain provides natural reuse opportunities. To reduce variance from underspecified visual design, we augment all methods’ inputs with the same *layout descriptions* generated from rendered reference pages; all the details are in Appendix A.2.

PaperBench. PaperBench (Starace et al., 2025) evaluates whether coding agents can reproduce AI research contributions from ICML 2024 Spotlight and Oral papers. We use the *Code-Dev* variant and construct five disjoint 4-task suites. Research implementations share abstractions such as training loops, data pipelines, model wrappers, and evaluation utilities, providing a distinct domain for evaluating library reuse. Suite details are provided in Appendix A.3.2.

4.1.2 Compared Methods

All methods use the same backbone LLM, deepseek-v4-flash (DeepSeek-AI, 2026), and the mini-SWE-agent harness (Yang et al., 2024). We distinguish SLA methods, which update the Super Library and migrate dependent applications as requests arrive, from baselines that either omit

library construction or perform it only after all applications are complete.

Non-SLA baselines. ZERO-SHOT generates each codebase independently, without a shared library or cross-codebase memory. LIBRARIAN (Kovacic et al., 2025) applies post-hoc library construction to the completed Zero-Shot codebases. It runs library construction $K=8$ times and selects the lowest-MDL candidate among those passing the functionality gate.

SLA methods. SLA-NAIVE uses the minimal scaffold introduced in Section 3.2. We evaluate two variants using implicit candidate discovery and Ward-clustering-based candidate selection. SLA-FULL augments SLA-Naive with index-based candidate extraction, pre-extraction codebase consolidation, extraction traces, and call graph-conditioned migration.

4.2 Evaluation Metrics

4.2.1 Application Functionality

WebGen-Bench. We use the two application-level metrics from WebGen-Bench (Lu et al., 2025). **Accuracy** ($\uparrow$) is the weighted pass rate over executable tests, assigning full credit to fully completed cases and half credit to partially completed ones; test operations are executed by a web-navigation UI agent (He et al., 2024).⁴ **Appearance Score** ($\uparrow$) is a 0–5 LLM judge score (Zheng et al., 2023) on the rendered entry page, measuring visual quality.⁵

PaperBench. We report the **Code-Dev Score** ($\uparrow$) (Starace et al., 2025), a normalized 0–1 LLM-judge score that checks the generated implementation against paper-specific rubric items.⁶

4.2.2 Code Maintainability

We evaluate maintainability over the final generated codebase, including all applications and the shared library, using five metrics.

Lines of Code (LOC, $\downarrow$). LOC is the total number of non-empty source lines across all generated applications and the shared library.

⁴The navigation agent is driven by gpt-5-mini with high reasoning effort.

⁵gpt-5-mini with default reasoning effort is used.

⁶deepseek-v4-flash with high reasoning effort is used as main evaluator, and gpt-5-mini with default reasoning effort is used as evaluation report parser.

Token Length (Tok, ↓). Token Length measures the total number of tokens in all generated applications and the shared library.⁷

Minimum Description Length (MDL, ↓). MDL (Kovacic et al., 2025) measures the negative log probability of the generated code under a reference model p_θ . We use Qwen-2.5-7B (Team, 2024) and adapt the original file-independent formulation to our setting with a dependency-aware variant:

$$\text{MDL}(\mathbf{C}) = -\log p_\theta(\mathcal{L}) - \sum_{i=1}^N \sum_{f \in c_i} \log p_\theta(f \mid \text{deps}(f)).$$

where $\mathbf{C}$ consists of codebases $c_1, \dots, c_N$ and the shared library $\mathcal{L}$, and $\text{deps}(f)$ denotes the direct imports of file f .

Structural Erosion (↓). Structural Erosion (Orlanski et al., 2026) measures the share of complexity mass in functions with cyclomatic complexity above 10, weighting each function by its complexity (McCabe, 1976) and the square root of its source lines of code.

Verbosity (↓). Verbosity (Orlanski et al., 2026) is the fraction of logical lines of code covered by duplicated code or rule-flagged redundant patterns.

4.3 Initial Portfolio Construction

WebGen-Bench. Table 1 (top) shows that all methods achieve comparable functionality, indicating that library use does not harm task performance. SLA-FULL obtains the best scores on LOC, token length, Erosion, and Verbosity, with statistically significant reductions in LOC, token length, and verbosity relative to zero-shot. LIBRARIAN attains the lowest MDL by selecting the lowest-MDL candidate among K sampled refactorings, but its Erosion rises above zero-shot rather than falling. In contrast, both SLA-NAIVE variants reduce some size metrics but increase Erosion, suggesting that unguided library extraction can concentrate complexity into shared components.

PaperBench. Table 1 (bottom) shows that SLA-FULL achieves the lowest values on all five maintainability metrics while retaining a comparable Code-Dev Score. In contrast, LIBRARIAN’s post-hoc MDL selection improves neither MDL nor functionality over Zero-Shot.

⁷Tokens are counted with the Qwen-2.5-7B (Team, 2024) tokenizer.

Further Evaluation. Paired t -tests show significant improvements in several maintainability metrics, while functionality differences are not statistically significant (Appendix A.1). We further report inference cost and evaluate two cost-reduction strategies (Appendix C) and compare against recent agentic SE scaffolds (Appendix F).

4.4 Post-Construction Maintenance

Our primary evaluation measures the functionality and structural maintainability of portfolios after initial construction. Existing benchmarks provide no shared follow-up changes for related applications. We therefore take each final portfolio from Section 4.3, consisting of the completed application codebases with or without a Super Library, and apply a shared policy update as an issue patch.

Shared policy updates. For each WebGen-Bench suite, we construct one cross-application policy update targeting behavior shared by multiple task specifications. A separate agent authors the update using only the original instructions and layout descriptions, without access to any generated portfolio or library.⁸ We instantiate the suite-level policy as application-specific issue-patch instructions and corresponding WebVoyager test cases for the target applications. Appendix B provides the policy construction and evaluation details.

Evaluation protocol. We apply the same issue patch to every method using the same edit agent.⁹ For methods with a shared library, a single agent jointly updates the library and all target applications. To preserve the independent-codebase setting, Zero-Shot assigns a separate agent to each application and disallows shared-library construction and cross-application imports. We measure **Patch Size (Added LOC, ↓)** from the resulting Git diff and report it separately for application and library code. We also evaluate whether the patched applications preserve their original behavior, satisfy the requested behavior, and maintain their visual quality.

Results. Table 2 shows that SLA-FULL achieves the smallest Patch Size overall and across all three suites. By centralizing shared changes in the Super Library, it substantially reduces application-level edits. This realizes the maintainability benefit that motivated our formulation of the Super Library

⁸We use claude-opus-4.8 with high reasoning effort.

⁹We use deepseek-v4-flash with high reasoning effort.

Method	Functionality		Maintainability					Library Size	
	Acc. $\uparrow$	Appr. $\uparrow$	LOC $\downarrow$	Tok $\downarrow$	MDL $\downarrow$	Eros. $\downarrow$	Verb. $\downarrow$	LOC	MDL
WebGen-Bench									
Zero-Shot	76.04	3.84	9393	70364	34875	0.1032	0.1603	–	–
LIBRARIAN ($K=1$)	75.09	3.90	9202	69492	34566	0.1266	0.1558	184	576
LIBRARIAN ($K=8$)	75.78	3.89	8973	68159	33919	0.1352	0.1472	222	639
NAIVE-IMPLICIT	76.95	3.89	9133	69777	35103	0.1567	0.1408	447	1212
NAIVE-WARD	76.07	3.90	8786	67774	34675	0.1250	0.1370	433	1175
SLA-FULL	77.21	3.86	8552	65633	34195	0.0987	0.0994	574	1630
Δ vs. Zero-Shot (%)	+1.5	+0.5	−9.0	−6.7	−1.9	−4.4	−38.0	–	–
PaperBench									
Zero-Shot	0.4687	–	6578	68603	30342	0.2558	0.8665	–	–
LIBRARIAN ($K=1$)	0.4741	–	6608	68859	30643	0.2534	0.8628	124	482
LIBRARIAN ($K=8$)	0.4591	–	6551	68297	30400	0.2527	0.8628	98	397
NAIVE-IMPLICIT	0.4802	–	6445	66539	29850	0.2882	0.8582	212	765
NAIVE-WARD	0.4720	–	6440	66028	29568	0.2761	0.8645	295	1027
SLA-FULL	0.4809	–	6252	63514	29489	0.2291	0.8490	336	1203
Δ vs. Zero-Shot (%)	+2.6	–	−5.0	−7.4	−2.8	−10.4	−2.0	–	–

Table 1: **Initial portfolio construction results.** Results are averaged over three 8-task WebGen-Bench suites and five 4-task PaperBench suites, each with three trials. PaperBench Acc. denotes the rubric-based Code-Dev Score, and appearance is not applicable. Maintainability metrics cover the complete portfolio, while Library Size covers only the Super Library. LIBRARIAN $K=1$ omits best-of- K selection.

Method	Patch Size			Functionality		
	Total $\downarrow$	App $\downarrow$	Lib	Orig. $\uparrow$	Req. $\uparrow$	Appr. $\uparrow$
Zero-Shot	936	936	0	77.9	81.9	3.93
LIBRARIAN	522	500	22	74.0	76.9	3.83
NAIVE-IMPLICIT	632	618	14	76.9	82.9	3.82
NAIVE-WARD	380	363	18	77.7	81.1	3.95
SLA-FULL	256	232	24	77.4	80.0	3.86

Table 2: **Maintenance under shared policy updates.** Results are averaged over three WebGen-Bench suites and three trials. Patch Size counts source lines added by the patch. Orig. and Req. report post-patch pass rates on the original and requested behaviors. Appr. denotes appearance quality.

Agent problem. The patched applications retain functionality and visual quality comparable to the baselines.

5 Analysis

Beyond the aggregate results in Section 4, we analyze how SLA-FULL improves reuse and maintainability over sequential application generation.

5.1 Ablation Study

In Table 3, we compare different candidate selection methods on SLA-NAIVE, and ablate the major components of SLA-FULL to isolate the contribu-

tion of each design choice.

Candidate selection for library update. In Table 3a, we compare three ways of providing reuse candidates to the library-extraction agent in the SLA-NAIVE setting: no explicit candidates, Ward cluster-based candidates, and our index-based candidate selection. We observe that our natural language (NL) summary-based approach gives the best scores in four out of five maintainability metrics. This is due to the NL-based approach extracting the largest number of shared components, as seen by the library size. The accuracy degrades in NL due to the additional shared component extraction introducing opportunities for the agent to introduce new bugs. We address this inaccuracy by using call graph conditioning in our SLA-FULL approach.

Pre-extraction codebase consolidation. We remove the pre-extraction codebase consolidation (Section 3.3.1) to test whether reducing intra-codebase redundancy before library extraction improves the quality of Super Library candidates. This isolates the effect of filtering duplicated or overly local implementations before they are considered for cross-application extraction. We observe that without local consolidation, the library

	Functionality		Maintainability					Library Size			Metric	w/o LC	w/o CG	SLA-Full
Sele.	Acc. $\uparrow$	Appr. $\uparrow$	LOC $\downarrow$	Tok $\downarrow$	MDL $\downarrow$	Eros. $\downarrow$	Verb. $\downarrow$	LOC	Tok	MDL	Acc. $\uparrow$	75.25	74.48	77.21
None	76.95	3.89	9133	69777	35103	0.1567	0.1408	447	3173	1212	Verb. $\downarrow$	0.1264	0.1215	0.0994
Ward	76.07	3.90	8786	67774	34675	0.1250	0.1370	433	3090	1175	Eros. $\downarrow$	0.1263	0.1409	0.0987
NL	72.95	3.89	8583	66727	32893	0.1122	0.1376	815	5944	2057	LOC $\downarrow$	8733	9068	8552
											MDL $\downarrow$	33880	34166	34195

(a) Candidate selection for SLA-NAIVE: **None**, embedding-based **Ward** clustering, and natural-language code-block summary matching (**NL**).

(b) **LC**: Local consolidation, **CG**: Call graph conditioning.

Table 3: **Ablations on WebGen-Bench.** We report the mean values over three suites $\times$ three trials. Maintainability metrics are computed over the final portfolio including the shared library.

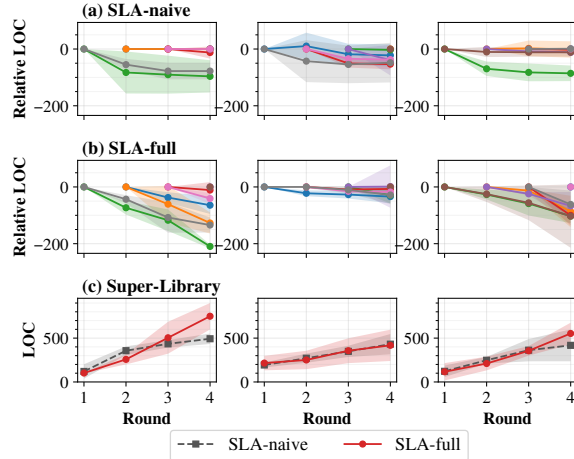
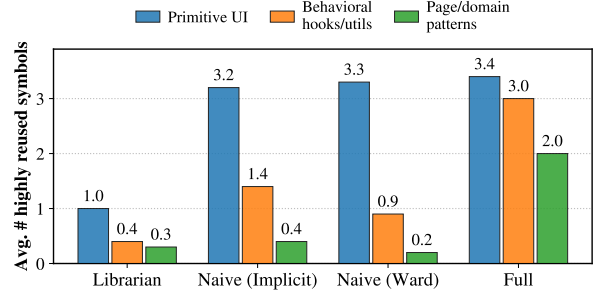


Figure 3: **LOC dynamics across sequential rounds.** Columns are WebGen suites 1–3. In (a) and (b), each colored line is one application. Values are relative to each app’s first round; shaded area shows ± 1 std over three trials.

agent fails to abstract shared components within a single codebase, since it is designed to extract shared components across multiple codebases. In Table 3b, this is reflected in the higher Verbosity, Erosion, and LOC metrics compared to SLA-FULL. The MDL metric fails to capture this, which is possibly caused by the quality of the training data of the LLM used for evaluating MDL.

Call Graph Conditioning. We ablate the migration context by removing call-graph (CG) conditioning, in order to test whether structural caller/callee context is necessary for reliably replacing local implementations with Super Library calls. In Table 3b, the accuracy of the no-CG variant is lower, which is due to the dependency-migration agent failing to properly handle the structural dependency updates. Also, without CG, the migration agent often leaves dead code after replacing the local implementation with a Super Library call, which is reflected in higher Verbosity, Erosion, and LOC.



(a) Average number of symbols by category.

Method	Primitive UI	Behavioral hooks/utils	Page/domain patterns
LIBRARIAN	Footer Header	useLocalStorage saveToStorage	ContactForm LoginForm
NAIVE-IMPLICIT	Header Footer	useLocalStorage useFormValidation	ContactForm KpiCard
NAIVE-WARD	Header Card	useLocalStorage createAuthContext	ContactForm
SLA-FULL	NavigationBar Section	useRouter useFilteredList	FeatureCardGrid MessageBanner

(b) Representative symbols (up to two examples per category; full per-run export inventories in Appendix I).

Figure 4: **Categories of highly reused Super Library symbols** (imported by ≥ 4 of 8 apps). SLA-FULL captures more behavioral and page/domain abstractions beyond primitives.

5.2 Maintainability Dynamics Across Rounds

We track each application’s LOC after the last phase of each round. This shows *when* reuse is accumulated, rather than only comparing final scores. In Figure 3, SLA-FULL tends to reduce app-local code as more applications are added to the stream. In contrast, the naive scaffold shows flatter trajectories, suggesting that a shared library alone is insufficient without reliable candidate discovery and migration.

5.3 Library Utilization and Abstraction

We analyze whether the final Super Library is both reused and meaningfully abstracted.

Method	No. of Apps			Total Exports
	1–2	3–5	6–8	
LIBRARIAN	1.2	1.6	1.0	4.2
NAIVE-IMPLICIT	1.3	3.6	2.7	8.2
NAIVE-WARD	1.4	3.3	2.0	7.2
SLA-FULL	2.2	6.0	4.8	13.3

Table 4: **Super Library utilization**, averaged over three suites $\times$ three trials.

5.3.1 Library Utilization

In Table 4, we measure library utilization by counting how many exported Super Library components are reused by different numbers of applications. The Exports column reports all exported library components, while the remaining columns group those exports by the number of generated applications that import them: 1–2, 3–5, and 6–8 apps. The result shows that our SLA-FULL builds a larger reusable library rather than merely increasing reuse density over a small set of components. SLA-FULL exposes 13.3 components, 5–6 more than the naive variants on average. SLA-FULL also increases the number of broadly reused exports, with 6.0 exports used by 3–5 apps and 4.8 exports used by 6–8 apps, compared to 3.3–3.6 and 2.0–2.7 for the naive variants.

5.3.2 Abstraction Quality

We analyze what types of abstractions are captured by highly reused Super Library symbols, defined as symbols imported by at least four of eight applications in a run. We categorize them into primitive UI components, behavioral hooks/utilities, and page- or domain-level patterns.

Figure 4a shows that, apart from LIBRARIAN, all methods extract comparable numbers of primitive UI components, but SLA-FULL captures substantially more reusable symbols beyond primitives, including behavioral hooks/utilities and page/domain patterns. Figure 4b provides representative examples, showing that the naive variants mostly reuse shallow widgets or thin utilities, whereas SLA-FULL additionally extracts interaction/state-management hooks and composed page/domain components.

This suggests summary-guided extraction broadens the library’s semantic coverage rather than merely raising primitive counts. Per-suite and per-trial export inventories are in Appendix I.

Metric	w/o library	w/ library		Δ
		App	Lib	
UI acc. (%) $\uparrow$	80.95	84.35		+3.40
Appear. (0–5) $\uparrow$	3.83	3.92		+0.09
LOC $\downarrow$	8,256	6,567	750	−940
MDL $\downarrow$	29,291	27,442	1,985	+135

Table 5: **Library-as-prior ablation**. 8 tasks $\times$ 3 trials. Δ on total (App + Lib).

5.4 Mature Libraries Enable Richer Application Generation

We test whether a Super Library can also serve as a prior for application generation. In this task-informed ablation, a vanilla coding agent generates eight Content-Presentation applications with or without access to SLA-FULL’s final library, while extraction, migration, candidate selection, and detailed layout inputs are disabled. Table 5 shows that library access improves average accuracy from 80.95 to 84.35 and appearance while reducing application-local code. Even after including the library itself, total LOC decreases by approximately 11%, with only a marginal change in MDL. Figure 7 shows the per-task comparisons.

6 Conclusion

As LLM coding agents build portfolios of related applications at scale, independent generation duplicates shared logic and prolonged agentic maintenance accumulates slop. We formulate this as the Super Library Agent problem, in which an agent develops related applications while maintaining a shared library of reusable components. SLA addresses missed abstractions and incomplete migration through candidate-guided extraction and context-aware dependency migration. Across WebGen-Bench and PaperBench, SLA preserves functionality while improving maintainability, library utilization, and abstraction quality. Post-patch results further show fewer cross-application edits, pointing toward more sustainable LLM coding for larger, evolving codebases.

Limitations

Lack of SLA-native benchmarks. A major limitation of this work is the absence of benchmarks specifically designed for the Super Library Agent setting. We adapt existing single-task coding benchmarks into sequential suites, but they were not originally built to evaluate long-horizon library growth, cross-application reuse, or codebase maintenance. Section 4.4 measures one round of post-deployment maintenance, but two gaps remain: the applications being maintained are themselves benchmark artifacts, generated from task specifications with no users, commit history, or production constraints, so it is untested whether the reduction in patch size survives on real applications; and we measure a single round rather than a sequence, so we do not observe how the shared library holds up as requirements keep drifting or how errors compound across rounds. Future benchmarks should include related application streams with enough shared structure for meaningful reuse, while also testing how the shared library supports evolving maintenance requests.

Limitations of maintainability metrics. Our maintainability evaluation relies on proxy metrics such as code size, duplication, verbosity, and library utilization. These metrics capture useful signals about abstraction and reuse, but they do not fully measure whether a codebase is easier to understand, modify, or extend. Moreover, maintainability is ultimately revealed through repeated future changes, whereas our evaluation combines static properties of the final codebases with a single maintenance round. Future work should extend this to multi-round maintenance on real applications, cross-application update consistency, regression frequency, and human judgments of abstraction quality.

Ethical Considerations

Automated code modification. Our system edits its code without a human in the loop: it rewrites existing applications to import from a shared library, and it applies a maintenance patch across a whole portfolio in a single pass. A migration that passes our functional checks can still alter behavior those checks do not cover, and because one library symbol is imported by many applications, a single faulty edit propagates to all of them at once. This concentration is the same property that makes the

shared library useful for maintenance, so it cannot be separated from the method. Applying this to software that people depend on would require human review of the generated diffs and stronger regression testing than a benchmark setting provides.

Released artifacts. The application codebases and Super Libraries we release are generated from benchmark task specifications and are intended as evidence for the claims in this paper, not as deployable software; they have not been reviewed for security, accessibility, or data-handling practices. The backbone models we use are trained on public code, so generated components may resemble existing implementations, and downstream users should verify licensing before reuse. Our experiments use only public coding benchmarks and involve no human subjects or personal data (Appendix J).

Acknowledgements

This work was supported by Institute for Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2019-II190075, Artificial Intelligence Graduate School Program (KAIST)), National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2023-00256259), the Institute of Information & Communications Technology Planning & Evaluation (IITP) with a grant funded by the Ministry of Science and ICT (MSIT) of the Republic of Korea in connection with the Global AI Frontier Lab International Collaborative Research (No. RS-2024-00469482 & RS-2024-00509279), Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. RS-2022-II220713, Meta-learning Applicable to Real-world Problems), and the “Advanced GPU Utilization Support Program” funded by the Government of the Republic of Korea (Ministry of Science and ICT).

References

- Ian Berlot-Attwell, Tobias Sesterhenn, Frank Rudzicz, and Xujie Si. 2026. Is this LLM library learning? evaluation must account for compute and behaviour. In *Proceedings of EACL 2026*.
- Matthew Bowers, Theo X. Olausson, Catherine Wong, Gabriel Grand, Joshua B. Tenenbaum, Kevin Ellis, and Armando Solar-Lezama. 2023. [Top-down syn-](#)

- thesis for library learning. *Proceedings of the ACM on Programming Languages*, 7(POPL).
- Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. 2024. [Large language models as tool makers](#). In *The Twelfth International Conference on Learning Representations*.
- Paul Clements and Linda Northrop. 2001. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional.
- DeepSeek-AI. 2026. Deepseek-v4: Towards highly efficient million-token context intelligence.
- Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. [Self-collaboration code generation via chatgpt](#). *ACM Trans. Softw. Eng. Methodol.*, 33(7).
- Sebastian Eder, Maximilian Junker, Elmar Jürgens, Benedikt Hauptmann, Rudolf Vaas, and Karl-Heinz Prommer. 2012. [How much does unused code matter for maintenance?](#) In *2012 34th International Conference on Software Engineering (ICSE)*, pages 1102–1111.
- Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sablé-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B. Tenenbaum. 2021. [DreamCoder: Bootstrapping inductive program synthesis with wake-sleep library learning](#). In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*.
- Gabriel Grand, Lionel Wong, Maddy Bowers, Theo X. Olausson, Muxin Liu, Joshua B. Tenenbaum, and Jacob Andreas. 2024. [LILO: Learning interpretable libraries by compressing and documenting code](#). In *The Twelfth International Conference on Learning Representations*.
- Hongliang He, Wenlin Yao, Kaixin Ma, Wenhao Yu, Yong Dai, Hongming Zhang, Zhenzhong Lan, and Dong Yu. 2024. [WebVoyager: Building an end-to-end web agent with large multimodal models](#). In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 6864–6890. Association for Computational Linguistics.
- Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. [Metagpt: Meta programming for a multi-agent collaborative framework](#). *Preprint*, arXiv:2308.00352.
- Kosei Horikawa, Hao Li, Yutaro Kashiwa, Bram Adams, Hajimu Iida, and Ahmed E. Hassan. 2025. [Agentic refactoring: An empirical study of AI coding agents](#). *Preprint*, arXiv:2511.04824.
- Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. 2009. [Do code clones matter?](#) In *2009 IEEE 31st International Conference on Software Engineering*, pages 485–495.
- Ziga Kovacic, Justin T. Chiu, Celine Lee, Wenting Zhao, and Kevin Ellis. 2025. Refactoring Codebases through Library Design. *arXiv preprint arXiv:2506.11058v1*. Version 1.
- Xunhao Lai, Weiqi Xu, Yufeng Yang, Yuxuan Zhang, Zehan Zhu, Chengyue Chen, Ziyi Wang, Yuchen Zhou, Hao Li, Zhenhua Xie, Yifan Wu, Jian Chen, Yu Zhao, Peng Sun, Xin Liu, Jie Zheng, and Bo Wang. 2026. MiniMax sparse attention. *arXiv preprint arXiv:2606.13392*.
- Yue Liu, Ratnadira Widyasari, Yanjie Zhao, Ivana Clairine Irsan, Junkai Chen, and David Lo. 2026. [Debt behind the ai boom: A large-scale empirical study of ai-generated code in the wild](#). *Preprint*, arXiv:2603.28592.
- Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. 2025. [Webgen-bench: Evaluating LLMs on generating interactive and functional websites from scratch](#). In *The Thirtieth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.
- Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320.
- Na Meng, Lisa Hua, Miryung Kim, and Kathryn S. McKinley. 2015. [Does automated refactoring obviate systematic editing?](#) In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 392–402.
- Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Nicholas Roberts, Frederic Sala, and Aws Albarghouti. 2026. [Slopcodebench: Benchmarking how coding agents degrade over long-horizon iterative tasks](#).
- Khouloud Oueslati, Maxime Lamothe, and Foutse Khomh. 2025. [RefAgent: A multi-agent LLM-based framework for automatic software refactoring](#). *Preprint*, arXiv:2511.03153.
- Klaus Pohl, Günter Böckle, and Frank Van Der Linden. 2005. *Software product line engineering: foundations, principles, and techniques*, volume 1. Springer.
- Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Andrey Sokolov, Timofey Bryksin, and Danny Dig. 2024. [Em-assist: Safe automated extractmethod refactoring with llms](#). In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, FSE 2024, pages 582–586, New York, NY, USA. Association for Computing Machinery.

- Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. [Chatdev: Communicative agents for software development](#). *Preprint*, arXiv:2307.07924.
- Chanchal Kumar Roy and James R. Cordy. 2007. A survey on software clone detection research. Technical Report 2007-541, School of Computing, Queen's University at Kingston, Kingston, Ontario, Canada.
- B.G. Ryder. 1979. [Constructing the call graph of a program](#). *IEEE Transactions on Software Engineering*, SE-5(3):216–226.
- Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. 2018. [Oreo: detection of clones in the twilight zone](#). In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE '18*, pages 354–365. ACM.
- Mohammed Latif Siddiq, Shafayat H Majumder, Maisha R Mim, Sourov Jajodia, and Joanna CS Santos. 2022. An empirical study of code smells in transformer-based code generation techniques. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 71–82. IEEE.
- Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. 2025. [Paperbench: Evaluating ai's ability to replicate ai research](#). *Preprint*, arXiv:2504.01848.
- Elias Stengel-Eskin, Archiki Prasad, and Mohit Bansal. 2024. Regal: refactoring programs to discover generalizable abstractions. In *Proceedings of the 41st International Conference on Machine Learning, ICML'24*. JMLR.org.
- Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. 2014. [Towards a big data curated benchmark of inter-project code clones](#). In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 476–480.
- Qwen Team. 2024. Qwen2.5 technical report. *arXiv preprint arXiv:2412.15115*.
- Alex Thillen, Niels Mündler, Veselin Raychev, and Martin Vechev. 2026. [CodeTaste: Can LLMs generate human-level code refactorings?](#) *Preprint*, arXiv:2603.04177.
- David Thomas and Andrew Hunt. 2019. *The pragmatic programmer, 20th anniversary edition: journey to mastery*, second edition edition. Addison-Wesley, Boston.
- Jesper van der Zwaag, Frank Driesens, Bouwe Postma, and Andrea Capiluppi. 2025. [Refactoring cross-project code duplication in an industrial software product line: A case study from RDW](#). *J. Syst. Softw.*, 230(C).
- Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. [Voyager: An open-ended embodied agent with large language models](#). *Preprint*, arXiv:2305.16291.
- Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, and 5 others. 2025a. OpenHands: An open platform for AI software developers as generalist agents. In *International Conference on Learning Representations (ICLR)*.
- Zhengren Wang, Rui Ling, Chufan Wang, Yongan Yu, Sizhe Wang, Zhiyu Li, Feiyu Xiong, and Wentao Zhang. 2025b. [MaintainCoder: Maintainable code generation under dynamic requirements](#). In *Advances in Neural Information Processing Systems*.
- John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. 2024. [SWE-agent: Agent-computer interfaces enable automated software engineering](#). In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*.
- Ruru Yue, Zhe Gao, Na Meng, Yingfei Xiong, Xiaoyin Wang, and J. David Morgenthaler. 2018. [Automatic clone recommendation for refactoring based on the present and the past](#). *Preprint*, arXiv:1807.11184.
- Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, Hao Zhang, Joseph E. Gonzalez, and Ion Stoica. 2023. Judging llm-as-a-judge with mt-bench and chatbot arena. In *Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23*, Red Hook, NY, USA. Curran Associates Inc.

Appendix Table of Contents

A	Initial-Construction Results	14
A.1	Paired Statistical Tests	14
A.2	WebGen-Bench Setup & Results .	14
A.3	PaperBench Setup & Results . . .	14
A.4	Qualitative Example	14
B	Post-Construction Maintenance	14
B.1	Policy Updates	14
B.2	Expanded Results	15
B.3	Qualitative Example	15
B.4	Change-Specific Test Cases	16
C	Inference Cost and Cost Reduction	24
C.1	Inference Cost	24
C.2	Migration-Session Reuse	24
C.3	Fewer Migration Rounds	24
D	Backbone Robustness	24
E	Low-Reuse Suite Stress Test	24
F	Agentic SE Scaffold Comparison	24
G	Algorithms	25
H	Prompts and Auxiliary Inputs	28
H.1	Agent Prompts	28
H.2	Code-Index Summarization	28
H.3	Candidate-Selection Prompts . . .	28
H.4	Extraction Trace	30
H.5	Call-Graph Context	31
I	Library Export Listings	31
J	Responsible NLP Checklist Details	31
J.1	Scientific Artifacts	31
J.2	AI Assistants	32

A Initial-Construction Results

A.1 Paired Statistical Tests

To assess whether the maintainability differences between SLA-FULL and each baseline are statistically reliable rather than artifacts of a few suites, we run paired significance tests at the level of individual tasks. For each metric we form one paired observation per task, taking the unit of pairing to be a (suite, task) pair and averaging over the three trials to reduce run-to-run noise. This yields $n = 20$ paired units for PaperBench (five suites of four papers) and $n = 24$ for WebGen-Bench (three suites of eight tasks). For every method pair we compute the mean paired difference Δ (oriented so that a positive value favors SLA-FULL on lower-is-better metrics), a bootstrap 95% confidence interval over Δ (10,000 resamples), and a two-sided paired t -test p -value. Table 6 reports the results. The reductions in token length and Verbosity over zero-shot are significant on both benchmarks, and SLA-FULL significantly reduces Erosion relative to the naive baselines. No functionality metric differs significantly between methods, so the maintainability gains do not come at the cost of task performance. Against LIBRARIAN, SLA-FULL obtains significant reductions in Erosion and Verbosity on WebGen-Bench and in token length and Verbosity on PaperBench, while no metric significantly favors LIBRARIAN.

A.2 WebGen-Bench Setup & Results

A.2.1 About Layout Descriptions

For fair comparison, we augment the original instructions with *detailed layout descriptions*. Because WebGen-Bench instructions are often brief, agents can produce substantially different initial designs, making maintainability metrics sensitive to arbitrary design complexity rather than library reuse. We therefore use Claude Opus 4.7 to generate an initial webpage from each raw instruction and describe the rendered layout, then provide this description to all methods together with the original instruction. This controls design variance and allows us to more directly evaluate code abstraction, migration, and reuse.

A.2.2 Evaluation Suites

Table 7 lists the tasks in each suite and how the suites were constructed.

A.2.3 Per-Suite Results

See Table 9 for WebGen-Bench per-suite results.

A.3 PaperBench Setup & Results

A.3.1 Agent Environment

We pre-install 27 packages in the agent container: PyTorch, NumPy, SciPy, scikit-learn, and pandas, along with common plotting, I/O, and configuration utilities. Domain frameworks such as stable-baselines3, transformers, and timm are left out on purpose. Such a framework already implements a paper’s method, so it would hand the agent the very code that a shared library is meant to capture. We would then be measuring whether a method calls the framework, not how it organizes its own code. The container also runs with `-network=none`, so no agent can install anything mid-run. The list of available packages is given to the agent in its prompt. Every method runs in this same container, Zero-Shot included.

A.3.2 Evaluation Suites

Table 8 lists the papers in each suite and how the suites were constructed.

A.3.3 Per-Suite Results

See Table 10 for PaperBench per-suite results.

A.4 Qualitative Example

Figure 5 shows a detailed example from WebGen-Bench Suite 1. Four of the eight applications persist state to `localStorage` in both portfolios. Zero-Shot writes the same load-and-persist cycle into each of them, each with its own `try/catch` and JSON round-trip. SLA-FULL extracts the cycle into one library hook, which the four applications reach through one `import` and one `call` each.

B Post-Construction Maintenance

B.1 Policy Updates

We construct each maintenance request in two isolated stages. First, an agent observes only the task metadata for a suite and writes one policy update grounded in behavior shared across its applications. Second, isolated sessions use each application’s original instruction and layout description to instantiate the policy for that application. Neither stage observes generated code, Super Libraries, or evaluation results. We retain the first policy produced for each suite without selecting among alternatives. This appendix reproduces the suite-level instructions and the resulting change-specific Web-Voyager tests. The tests also identify every target application.

Metric	vs. Zero-Shot		vs. Librarian		vs. Naive-Implicit		vs. Naive-Ward	
	Δ [95% CI]	p	Δ [95% CI]	p	Δ [95% CI]	p	Δ [95% CI]	p
WebGen-Bench ($n = 24$)								
Acc. $\uparrow$	-0.85 [-3.24, +1.38]	0.484	-1.14 [-3.07, +0.78]	0.274	+0.23 [-2.38, +2.60]	0.864	-1.04 [-3.23, +1.06]	0.358
Appear. $\uparrow$	-0.028 [-0.125, +0.042]	0.539	+0.028 [-0.069, +0.125]	0.575	+0.028 [-0.056, +0.125]	0.575	+0.042 [-0.028, +0.125]	0.328
LOC $\downarrow$	+105 [+54, +154]	0.001	+53 [+0, +102]	0.059	+73 [+28, +126]	0.009	+29 [+3, +56]	0.044
Tok $\downarrow$	+591 [+275, +891]	0.001	+316 [-1, +633]	0.068	+518 [+232, +884]	0.006	+268 [+60, +492]	0.028
MDL $\downarrow$	+85 [-43, +206]	0.207	-34 [-170, +92]	0.623	+113 [-57, +308]	0.256	+60 [-77, +207]	0.434
Eros. $\downarrow$	+0.0213 [-0.0072, +0.0535]	0.189	+0.1338 [+0.0836, +0.1860]	4.5e-05	+0.1322 [+0.0595, +0.2113]	0.002	+0.1249 [+0.0502, +0.2037]	0.004
Verb. $\downarrow$	+0.0582 [+0.0315, +0.0867]	0.001	+0.0334 [+0.0101, +0.0594]	0.017	+0.0240 [-0.0021, +0.0609]	0.174	+0.0455 [+0.0251, +0.0719]	0.001
PaperBench ($n = 20$)								
Score $\uparrow$	-0.0122 [-0.0437, +0.0182]	0.458	-0.0218 [-0.0545, +0.0107]	0.216	-0.0007 [-0.0242, +0.0274]	0.959	-0.0089 [-0.0416, +0.0240]	0.607
LOC $\downarrow$	+81 [-15, +165]	0.097	+75 [-19, +158]	0.120	+48 [-37, +128]	0.271	+47 [-56, +146]	0.388
Tok $\downarrow$	+1272 [+357, +2071]	0.010	+1196 [+325, +1979]	0.012	+756 [+134, +1369]	0.029	+628 [-543, +1735]	0.304
MDL $\downarrow$	+213 [-171, +560]	0.273	+221 [-157, +568]	0.250	+90 [-203, +345]	0.531	+20 [-542, +551]	0.945
Eros. $\downarrow$	+0.0362 [-0.0104, +0.0804]	0.148	+0.0291 [-0.0157, +0.0705]	0.219	+0.0496 [+0.0091, +0.0935]	0.038	+0.0347 [-0.0032, +0.0736]	0.101
Verb. $\downarrow$	+0.0342 [+0.0226, +0.0460]	2.5e-05	+0.0267 [+0.0149, +0.0387]	4.0e-04	+0.0189 [+0.0041, +0.0333]	0.022	+0.0210 [+0.0064, +0.0339]	0.008

Table 6: **Paired significance tests** comparing SLA-FULL against each baseline (protocol in the text). Δ is the compared method minus SLA-FULL, so SLA-FULL is favored by a positive Δ on lower-is-better ($\downarrow$) metrics and by a negative Δ on higher-is-better ($\uparrow$) ones; **bold** marks $p < 0.05$. WebGen LOC/Tok/MDL are per-task.

Suite 1: Engagement-form standardization.

Our organization has standardized how all customer-facing engagement forms behave across its web properties. For every contact, lead, newsletter, reservation, or similar primary engagement form:

1. Required text inputs must reject empty or whitespace-only values.
2. If an email field exists, it must require a value containing both “@” and “.”.
3. An invalid submit must show a clear inline error message and must not show a success state.
4. An invalid submit must preserve the user’s current input values.
5. A valid submit must show an in-page confirmation without navigating away.
6. The confirmation must include either “Received” or “Submitted”, and must include the submitted name or email when such a field exists.

Original navigation and content behavior must not be removed or weakened.

Suite 2: Record-audit policy.

Our organization has adopted a record-audit policy for its data-management applications. For each app’s primary create/update workflows:

1. When a new domain record is created, display a human-readable record ID with a domain prefix and a 3+ digit/character suffix where possible (e.g., CALL-001, LEAD-001, PAT-001, TASK-001). Where the app already exposes a domain-prefixed ID scheme, keep that scheme.
2. Created or updated records must display a “Last updated” timestamp or date, refreshed on update.
3. Create and update actions must show a confirmation message.
4. The created/updated record must be findable from the app’s relevant list, search, filter, tracking, or status view.

Original CRUD, report, and navigation behavior must not be removed or weakened.

Suite 3: Trust, safety, and search quality.

Our platform group has adopted a trust-and-safety policy for user-generated content, plus a search-quality baseline.

Part A: Moderation status. For each publicly visible user-generated content submission flow—submissions that appear in lists, feeds, or search results seen by other users (e.g., shared promotions, job postings, job/internship applications appearing in a reviewable list, tool comments/ratings, uploaded pictures, newly registered public profile cards):

1. After successful submission, show a confirmation message.
2. The submitted item must display a visible “Pending review” badge or status text.
3. Pending items must remain visible in the relevant list/detail/search views—do not hide them.

Explicitly excluded from moderation (keep original immediate behavior): edits to one’s own profile fields, one-to-one chat messages, friend invitations.

Part B: Search normalization. Search must trim leading/trailing whitespace, match case-insensitively, and show a clear empty-state message when there are no matches (instead of a blank area or silently showing all items).

Original registration, login, navigation, and search behavior must not be removed or weakened.

B.2 Expanded Results

Table 11 supplements the aggregate results in Table 2 with the number of files touched, changes in portfolio-level structural metrics, and Patch Size for each suite.

B.3 Qualitative Example

Figure 6 shows a detailed example from WebGen-Bench Suite 1. Zero-Shot and SLA-FULL receive

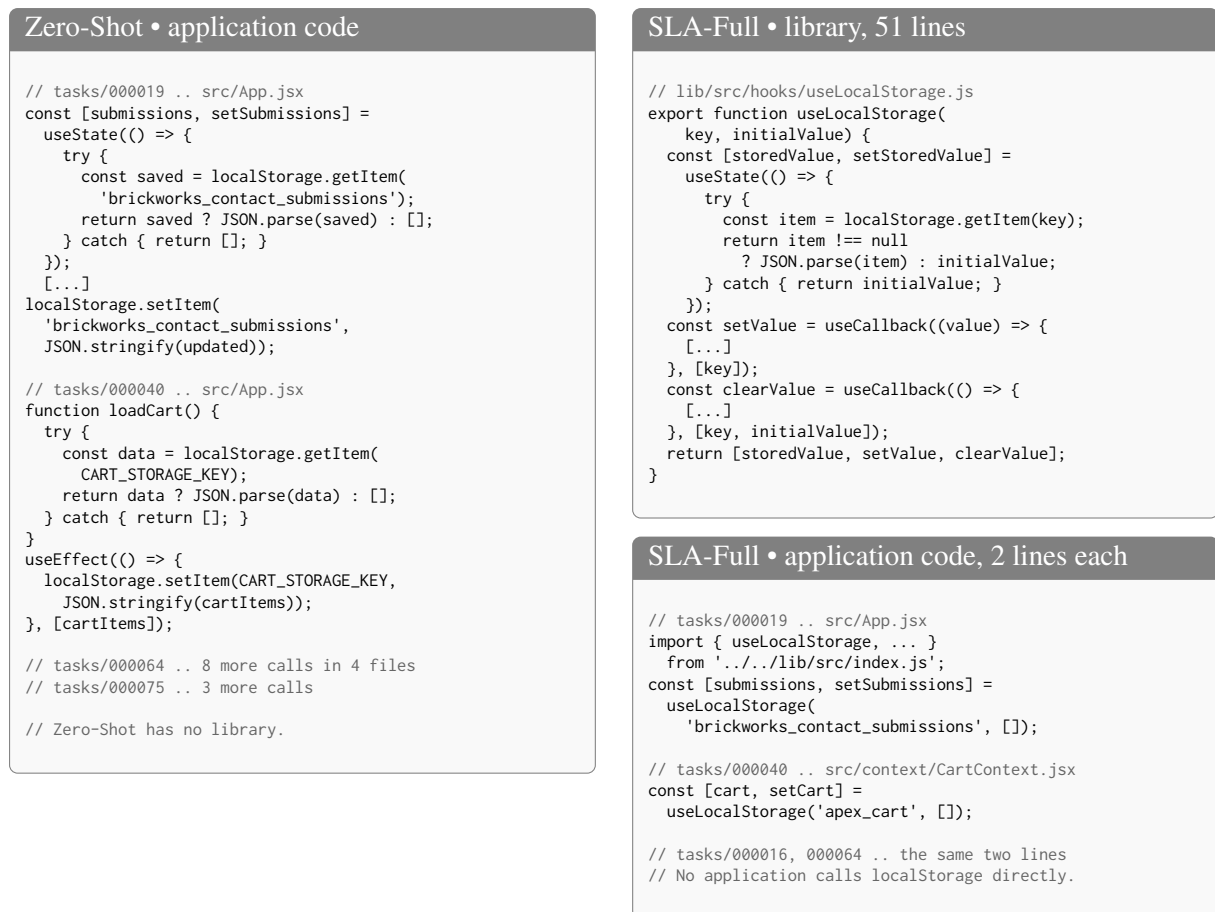


Figure 5: **Initial-construction example.** Zero-Shot repeats the same persistence cycle in every application that needs it, while SLA-FULL defines it once in the library.

the same policy (Section B.1), whose first two items require that an email field contain both @ and a dot. Zero-Shot implements the rule independently in four of the six applications, each with its own message. SLA-FULL adds it once to a library component, the shared validator that the shared contact form already calls, so five of the six applications need no edit at all.

B.4 Change-Specific Test Cases

Tables 12–14 report the WebVoyager action and expected result used to evaluate each target application.

Zero-Shot • application code, +186

```
// tasks/000019 .. src/pages/Contact.jsx
+ } else if (!email.includes('@'))
+   || !email.includes('.') {
+   newErrors.email = <app-specific text>;
+ }

// tasks/000071 .. src/pages/Contact.jsx
+ } else if (!email.includes('@'))
+   || !email.includes('.') {
+   newErrors.email = <app-specific text>;
+ }

// tasks/000075 .. src/App.jsx
+ if (!email.includes('@'))
+   || !email.includes('.') {
+   setError(<app-specific text>);
+   return;
+ }

// tasks/000016 .. a fourth copy of the check

// Zero-Shot has no library.
```

SLA-Full • library, +76

```
// lib/src/utils/validate.js
+export function validateEmailField(
+  value, label = 'Email') {
+  if (!value || !value.trim()) return null;
+  const trimmed = value.trim();
+  if (!trimmed.includes('@'))
+    || !trimmed.includes('.') {
+    return `${label} must contain both
+      "@" and "."`;
+  }
+  return null;
+}

// lib/src/components/ContactForm.jsx
+ const emailError =
+   validateEmailField(values[f.name],
+     f.label);
+   [...] // +53 -10 in this file

// lib/src/index.js
+export { validateEmailField }
+ from './utils/validate.js';
```

SLA-Full • application code, +1

```
// tasks/000019 .. src/pages/Contact.jsx
- { name: 'company', label: 'Company' },
+ { name: 'company', label: 'Company',
+   optional: true },

// the other five applications: unchanged
```

Figure 6: **Post-construction maintenance example.** Zero-Shot repeats the requested rule in every application it applies to, while SLA-FULL adds it once to the library.

Suite	WebGen-Bench tasks
S1 <i>Content Presentation Brochures & Portfolios</i>	1. [000019] SW solutions brochure. 2. [000075] Team projects & skills. 3. [000015] Clinical office info. 4. [000016] Solar company products. 5. [000040] Consulting & training. 6. [000072] Personal CV/resume. 7. [000071] Designer portfolio. 8. [000064] Shipping news blog.
S2 <i>Data Management CRUD / ERP / CRM</i>	1. [000085] Construction project mgmt. 2. [000022] Customer-service call log. 3. [000046] Hospital admin & inventory. 4. [000083] Detective agency case mgmt. 5. [000047] E-government office system. 6. [000044] Supply-chain traceability. 7. [000023] CRM sales lead tracking. 8. [000080] Advanced to-do list app.
S3 <i>User Interaction Social & Job Platforms</i>	1. [000052] Blue-collar job board. 2. [000090] Social networking site. 3. [000051] Driver recruitment listings. 4. [000077] Engineer tool info site. 5. [000092] Skill-sharing platform. 6. [000053] Student internship portal. 7. [000091] Matrimonial matching site. 8. [000027] Promotions & discounts forum.

Table 7: **WebGen-Bench evaluation suites.** We cluster the 101 WebGen-Bench tasks with Ward agglomerative clustering ($k=14$, Euclidean) over text-embedding-3-small embeddings of each task’s instruction, category metadata, and UI-test interactions, and select three of the resulting clusters as evaluation suites. From each selected cluster we take the eight tasks closest to the centroid; each suite is then used as a sequential application stream.

Suite	PaperBench tasks
S1	1. fre: Functional Reward Encodings. 2. rice: RICE. 3. sapg: Split and Aggregate Policy Gradients. 4. ftrl: RL Fine-tuning as Forgetting Mitigation.
S2	1. robust-clip: Robust CLIP. 2. sample-specific-masks: Sample-specific Masks. 3. lca-on-the-line: OOD Generalization. 4. test-time-model-adaptation: Test-Time Adaptation.
S3	1. mechanistic-understanding: DPO and Toxicity. 2. what-will-my-model-forget: Forgotten Examples. 3. stay-on-topic-with-classifier-free-guidance: CFG. 4. adaptive-pruning: Adaptive Pruning and Tuning.
S4	1. bridging-data-gaps: Diffusion Transfer Learning. 2. sequential-neural-score-estimation: Neural Score Estimation. 3. stochastic-interpolants: Data-Dependent Couplings. 4. all-in-one: Simulation-Based Inference.
S5	1. bam: Batch-and-Match Variational Inference. 2. pinn: Training PINNs Loss Landscapes. 3. lbcs: Refined Coreset Selection. 4. bbox: BBox-Adapter for Black-Box LLMs.

Table 8: **PaperBench evaluation suites.** We construct five disjoint suites of four tasks each. Suites S1–S3 are topically coherent (RL, VLM, NLP), S4 is a diffusion/score-based inference suite, and S5 is a heterogeneous low-reuse suite included as a robustness probe. Each suite is used as a sequential task stream for evaluating Super Library Agent methods.

Metric	Suite	Zero	Librarian	Naive		Full
				Implicit	Ward	
Functionality						
Acc. $\uparrow$	S1	82.99 $\pm$ 2.12	81.97 $\pm$ 1.56	83.67 $\pm$ 0.00	83.67 $\pm$ 3.68	83.33 $\pm$ 2.12
	S2	70.28 $\pm$ 2.55	70.83 $\pm$ 2.21	70.83 $\pm$ 3.00	70.00 $\pm$ 4.64	72.22 $\pm$ 3.37
	S3	74.85 $\pm$ 0.52	74.55 $\pm$ 1.82	76.36 $\pm$ 1.57	74.54 $\pm$ 3.96	76.06 $\pm$ 2.62
	Avg.	76.04 $\pm$ 5.82	75.78 $\pm$ 5.17	76.96 $\pm$ 5.83	76.07 $\pm$ 7.00	77.21 $\pm$ 5.44
Appear. $\uparrow$	S1	3.84 $\pm$ 0.08	3.96 $\pm$ 0.07	4.00 $\pm$ 0.00	3.96 $\pm$ 0.07	3.92 $\pm$ 0.07
	S2	3.92 $\pm$ 0.07	3.84 $\pm$ 0.08	3.84 $\pm$ 0.08	3.96 $\pm$ 0.07	3.96 $\pm$ 0.07
	S3	3.75 $\pm$ 0.00	3.88 $\pm$ 0.13	3.84 $\pm$ 0.08	3.79 $\pm$ 0.08	3.71 $\pm$ 0.08
	Avg.	3.84 $\pm$ 0.09	3.89 $\pm$ 0.10	3.89 $\pm$ 0.10	3.90 $\pm$ 0.10	3.86 $\pm$ 0.13
Maintainability						
LOC $\downarrow$	S1	7129 $\pm$ 235	6623 $\pm$ 302	6809 $\pm$ 83	6459 $\pm$ 42	6299 $\pm$ 245
	S2	9834 $\pm$ 463	9437 $\pm$ 705	9890 $\pm$ 741	9668 $\pm$ 768	9689 $\pm$ 576
	S3	11215 $\pm$ 51	10858 $\pm$ 207	10700 $\pm$ 734	10232 $\pm$ 665	9669 $\pm$ 445
	Avg.	9392 $\pm$ 1819	8973 $\pm$ 1909	9133 $\pm$ 1853	8787 $\pm$ 1834	8552 $\pm$ 1733
MDL $\downarrow$	S1	31474 $\pm$ 1691	30378 $\pm$ 1886	30556 $\pm$ 136	29927 $\pm$ 614	30072 $\pm$ 1193
	S2	36138 $\pm$ 3020	35371 $\pm$ 3372	37607 $\pm$ 2778	38014 $\pm$ 2299	36924 $\pm$ 3397
	S3	37013 $\pm$ 553	36008 $\pm$ 1281	37144 $\pm$ 125	36082 $\pm$ 2663	35589 $\pm$ 1884
	Avg.	34875 $\pm$ 3118	33919 $\pm$ 3357	35102 $\pm$ 3689	34674 $\pm$ 4070	34195 $\pm$ 3745
Eros. $\downarrow$	S1	0.0113 $\pm$ 0.0196	0.0948 $\pm$ 0.1179	0.1476 $\pm$ 0.0819	0.1406 $\pm$ 0.0670	0.0102 $\pm$ 0.0177
	S2	0.0817 $\pm$ 0.0101	0.0842 $\pm$ 0.0189	0.1151 $\pm$ 0.0717	0.0613 $\pm$ 0.0253	0.0673 $\pm$ 0.0319
	S3	0.2167 $\pm$ 0.0463	0.2264 $\pm$ 0.0578	0.2075 $\pm$ 0.0312	0.1730 $\pm$ 0.0358	0.2185 $\pm$ 0.0335
	Avg.	0.1032 $\pm$ 0.0939	0.1352 $\pm$ 0.0954	0.1567 $\pm$ 0.0697	0.1250 $\pm$ 0.0639	0.0986 $\pm$ 0.0964
Verb. $\downarrow$	S1	0.0670 $\pm$ 0.0189	0.0514 $\pm$ 0.0244	0.0148 $\pm$ 0.0056	0.0277 $\pm$ 0.0192	0.0361 $\pm$ 0.0251
	S2	0.2261 $\pm$ 0.0336	0.2187 $\pm$ 0.0318	0.1986 $\pm$ 0.0610	0.1995 $\pm$ 0.0166	0.1382 $\pm$ 0.0212
	S3	0.1878 $\pm$ 0.0234	0.1715 $\pm$ 0.0120	0.2090 $\pm$ 0.0335	0.1838 $\pm$ 0.0142	0.1238 $\pm$ 0.0127
	Avg.	0.1603 $\pm$ 0.0754	0.1472 $\pm$ 0.0776	0.1408 $\pm$ 0.1008	0.1370 $\pm$ 0.0836	0.0994 $\pm$ 0.0510
Library Size						
LOC	S1	–	247 $\pm$ 41	493 $\pm$ 59	510 $\pm$ 62	750 $\pm$ 151
	S2	–	225 $\pm$ 252	431 $\pm$ 116	499 $\pm$ 169	418 $\pm$ 179
	S3	–	194 $\pm$ 80	418 $\pm$ 182	290 $\pm$ 67	553 $\pm$ 123
	Avg.	–	222 $\pm$ 136	447 $\pm$ 117	433 $\pm$ 144	573 $\pm$ 196
Tok.	S1	–	1849 $\pm$ 357	3529 $\pm$ 399	3666 $\pm$ 336	5281 $\pm$ 1103
	S2	–	1579 $\pm$ 1663	3123 $\pm$ 753	3592 $\pm$ 750	3070 $\pm$ 1207
	S3	–	1334 $\pm$ 554	2867 $\pm$ 1161	2012 $\pm$ 607	4077 $\pm$ 989
	Avg.	–	1587 $\pm$ 922	3173 $\pm$ 776	3090 $\pm$ 957	4143 $\pm$ 1353
MDL	S1	–	640 $\pm$ 130	1297 $\pm$ 289	1445 $\pm$ 155	1985 $\pm$ 308
	S2	–	752 $\pm$ 753	1193 $\pm$ 299	1332 $\pm$ 286	1214 $\pm$ 383
	S3	–	526 $\pm$ 203	1147 $\pm$ 318	749 $\pm$ 141	1693 $\pm$ 305
	Avg.	–	639 $\pm$ 407	1212 $\pm$ 270	1175 $\pm$ 369	1631 $\pm$ 444

Table 9: **Per-suite results on WebGen-Bench.** We report functionality and maintainability metrics across three disjoint suites of eight tasks each, as mean $\pm$ std over three independent trials per suite. Avg. is pooled across all 9 trials (3 suites $\times$ 3 trials).

Metric	Suite	Zero	Librarian	Naive		Full
				Implicit	Ward	
Functionality						
Score ↑	S1	0.3409 ±0.0918	0.3539 ±0.0476	0.3393 ±0.0822	0.3918 ±0.0594	0.3690 ±0.0456
	S2	0.4394 ±0.0127	0.3924 ±0.0137	0.4885 ±0.0382	0.4557 ±0.0269	0.4584 ±0.0104
	S3	0.4461 ±0.0193	0.4420 ±0.0406	0.4006 ±0.0154	0.3958 ±0.0799	0.4160 ±0.0597
	S4	0.5909 ±0.0564	0.5801 ±0.0772	0.6071 ±0.0701	0.5392 ±0.0650	0.5751 ±0.0683
	S5	0.5261 ±0.0674	0.5271 ±0.0206	0.5655 ±0.0440	0.5773 ±0.0654	0.5860 ±0.0174
	Avg.	0.4687 ±0.1004	0.4591 ±0.0949	0.4802 ±0.1132	0.4720 ±0.0936	0.4809 ±0.0975
Maintainability						
LOC ↓	S1	7128 ±891	7125 ±874	7155 ±582	7156 ±602	7080 ±526
	S2	6239 ±1006	6209 ±961	6288 ±754	6381 ±820	6387 ±1109
	S3	6922 ±86	6928 ±120	6309 ±518	5992 ±337	6198 ±381
	S4	6057 ±421	5998 ±407	6041 ±455	6134 ±1059	5604 ±528
	S5	6544 ±488	6496 ±437	6430 ±140	6536 ±286	5993 ±441
	Avg.	6578 ±701	6551 ±697	6445 ±594	6440 ±715	6252 ±748
Tok. ↓	S1	69723 ±6679	69730 ±6560	70292 ±4967	70942 ±5114	67897 ±5531
	S2	64270 ±10973	64100 ±10502	65198 ±6768	64823 ±9037	63029 ±12116
	S3	73351 ±2528	73147 ±1806	64868 ±6821	59797 ±4782	62997 ±5168
	S4	64699 ±4074	64160 ±3727	63268 ±6188	64403 ±10701	59037 ±4488
	S5	70974 ±7082	70350 ±6461	69066 ±1807	70176 ±2483	64613 ±4752
	Avg.	68603 ±6900	68297 ±6644	66539 ±5506	66028 ±7346	63514 ±6636
MDL ↓	S1	28483 ±2831	28724 ±2727	29080 ±2283	28671 ±2073	29006 ±3820
	S2	29510 ±4585	29656 ±4605	29216 ±1649	30483 ±4302	30444 ±7164
	S3	36079 ±3979	36140 ±3409	32919 ±2949	29731 ±2625	32518 ±624
	S4	27663 ±1995	27541 ±1892	27665 ±4704	28750 ±4919	27281 ±1832
	S5	29977 ±3606	29941 ±3555	30369 ±1720	30207 ±3485	28198 ±1530
	Avg.	30342 ±4283	30400 ±4201	29850 ±3045	29568 ±3166	29489 ±3728
Eros. ↓	S1	0.2070 ±0.0202	0.2133 ±0.0290	0.2536 ±0.0835	0.2585 ±0.0159	0.1552 ±0.0244
	S2	0.2342 ±0.0777	0.2350 ±0.0784	0.2394 ±0.0460	0.2281 ±0.0769	0.2678 ±0.0348
	S3	0.2560 ±0.0660	0.2383 ±0.0519	0.2656 ±0.0509	0.2709 ±0.0995	0.2616 ±0.0347
	S4	0.2012 ±0.1213	0.1974 ±0.1132	0.2978 ±0.0476	0.2916 ±0.0856	0.1641 ±0.0555
	S5	0.3804 ±0.0646	0.3795 ±0.0657	0.3849 ±0.1232	0.3314 ±0.0611	0.2967 ±0.1291
	Avg.	0.2558 ±0.0939	0.2527 ±0.0915	0.2882 ±0.0840	0.2761 ±0.0717	0.2291 ±0.0828
Verb. ↓	S1	0.8889 ±0.0042	0.8865 ±0.0040	0.8707 ±0.0216	0.8860 ±0.0135	0.8552 ±0.0064
	S2	0.8482 ±0.0231	0.8435 ±0.0258	0.8413 ±0.0197	0.8582 ±0.0102	0.8356 ±0.0154
	S3	0.8455 ±0.0102	0.8404 ±0.0120	0.8394 ±0.0217	0.8404 ±0.0278	0.8393 ±0.0221
	S4	0.8645 ±0.0097	0.8618 ±0.0117	0.8719 ±0.0090	0.8646 ±0.0048	0.8518 ±0.0222
	S5	0.8853 ±0.0153	0.8818 ±0.0173	0.8675 ±0.0112	0.8734 ±0.0251	0.8630 ±0.0132
	Avg.	0.8665 ±0.0221	0.8628 ±0.0238	0.8582 ±0.0212	0.8645 ±0.0222	0.8490 ±0.0177
Library Size						
LOC	S1	–	152 ±53	216 ±91	308 ±83	541 ±73
	S2	–	48 ±37	156 ±87	355 ±109	277 ±150
	S3	–	105 ±48	220 ±49	231 ±167	217 ±128
	S4	–	163 ±43	320 ±25	364 ±79	524 ±15
	S5	–	20 ±5	146 ±80	215 ±70	118 ±34
	Avg.	–	98 ±68	212 ±88	295 ±111	336 ±192
Tok.	S1	–	1445 ±501	1960 ±821	2914 ±712	4872 ±754
	S2	–	472 ±364	1414 ±629	3132 ±963	2733 ±1786
	S3	–	950 ±397	2104 ±587	2089 ±1418	1879 ±1135
	S4	–	2017 ±346	3385 ±401	4048 ±899	5309 ±184
	S5	–	171 ±48	1347 ±653	1757 ±415	1050 ±416
	Avg.	–	1011 ±752	2042 ±929	2788 ±1155	3168 ±1927
MDL	S1	–	516 ±208	650 ±190	901 ±125	1583 ±263
	S2	–	201 ±126	511 ±218	1212 ±381	1043 ±441
	S3	–	368 ±130	854 ±131	825 ±561	786 ±379
	S4	–	779 ±240	1137 ±148	1300 ±149	2084 ±125
	S5	–	121 ±26	673 ±327	896 ±41	520 ±197
	Avg.	–	397 ±280	765 ±287	1027 ±332	1203 ±637

Table 10: **Per-suite results on PaperBench.** We report functionality and maintainability metrics across five disjoint suites of four tasks each, as mean $\pm$ std over three independent trials per suite. Avg. is pooled across all 15 trials (5 suites $\times$ 3 trials). Library Size reports the size of the final Super Library only.

Method	Patch				Change in maintainability					Patch Size by suite		
	Added LOC ↓	App ↓	Lib	Files ↓	ΔLOC ↓	ΔTok ↓	ΔMDL ↓	ΔEros. ↓	ΔVerb. ↓	S1	S2	S3
Zero-Shot	936	936	0	29.8	+605	+5101	+2565	−0.004	+0.007	182	1804	822
LIBRARIAN	522	500	22	27.0	–	–	–	–	–	151	717	699
NAIVE-IMPLICIT	632	618	14	27.4	+278	+2874	+2078	+0.019	+0.005	97	1328	471
NAIVE-WARD	380	363	18	22.2	+251	+2779	+1909	+0.030	+0.011	103	574	463
SLA-FULL	256	232	24	19.9	+171	+2073	+1377	+0.001	+0.003	79	422	268

Table 11: **Post-construction maintenance, full results.** The table extends Table 2 with the number of files touched, changes in portfolio-level structural metrics, and Patch Size for each suite. Δ columns report post-patch minus pre-patch values. They are unavailable for LIBRARIAN, whose pre-patch portfolio is a post-hoc refactoring of the Zero-Shot codebases rather than a round-structured run.

Table 12: **Change-specific tests for Suite 1.** Each row reproduces one WebVoyager action and its expected result.

App ID	Action	Expected result
000015	Go to the Contact Us page. In the contact form, fill the Name and Message fields but leave Email empty (or whitespace only), then submit.	An inline error message appears, no success/confirmation state is shown, and the values already typed in Name and Message remain in the fields.
000015	On the same contact form, fill Name, a valid Email containing “@” and “.”, and Message, then submit.	An in-page confirmation appears on the Contact Us page without navigating away, containing the word “Received” or “Submitted” and including the submitted name or email.
000016	Go to the Contact page. In the contact form, enter a Name and Message but type an invalid email without “@” or “.” (e.g., “testmail”), then submit.	A clear inline error appears for the email field, no confirmation banner is shown, and the typed Name, Email, and Message values remain in the inputs.
000016	Correct the email to a valid address containing “@” and “.”, then submit the form.	An in-page confirmation banner appears on the Contact page without navigating away, containing “Received” or “Submitted” together with the submitted name or email.
000019	Go to the Contact page. Submit the contact form with the “How can we help?” field empty while Name and Email are filled.	An inline error message appears, no confirmation is shown, and all values typed so far, including Company if entered, are preserved in the form.
000019	Fill Name, a valid Email containing “@” and “.”, and the “How can we help?” field, then submit.	An in-page confirmation appears on the Contact page without navigating away, including “Received” or “Submitted” plus the submitted name or email.
000064	Go to the Newsletter page. Enter an invalid email without “@” or “.” (e.g., “newsfan”) into the subscription input and click Subscribe.	A clear inline error message appears, no success state is shown, and the typed value remains in the input.
000064	Replace it with a valid email containing “@” and “.”, then click Subscribe.	An in-page confirmation appears on the same page without navigating away, containing “Received” or “Submitted” along with the submitted email address.
000071	Go to the Contact page’s “Get in Touch” form. Submit it with the Message field empty (or spaces only) while Name and Email are filled.	A clear inline error message appears, no saved/confirmation line is shown, and the typed Name and Email values are preserved in the form.
000071	Fill all three fields with valid values (email containing “@” and “.”) and submit.	An in-page confirmation appears beneath the form without navigating away, including “Received” or “Submitted” together with the submitted name or email.
000075	Scroll to the Contact Us section. Submit the contact form with all fields empty.	A clear inline error message appears within the Contact Us section, and no success state is shown.
000075	Fill name, a valid email containing “@” and “.”, and a message, then submit.	An in-page confirmation appears within the Contact Us section without navigating away, and includes “Received” or “Submitted” plus the submitted name or email.

Table 13: **Change-specific tests for Suite 2.** Each row reproduces one WebVoyager action and its expected result.

App ID	Action	Expected result
000022	Create a new call log via the New Call form with required fields filled.	A confirmation message appears, and the new call log displays a human-readable ID with the CALL- prefix (e.g., CALL-001) and a “Last updated” timestamp. The call appears in the Open Calls or To-Do Calls view.
000022	Modify an existing client’s information (or close/mark done an existing call).	A confirmation message appears and the record’s “Last updated” timestamp refreshes to reflect the change.
000023	Create a new lead via the New Lead form/modal with valid details.	A confirmation message appears, and the new lead is shown with a LEAD- prefixed ID (e.g., LEAD-001) and a “Last updated” timestamp or date, visible in the leads table.
000023	Change that lead’s status (or edit it), then locate it using the leads search or filters.	A confirmation appears, the “Last updated” value refreshes, and the lead is findable via search/filter with its LEAD- ID still displayed.
000044	Register a new product via the registration form.	A confirmation message appears and the product’s SCT- prefixed ID is displayed immediately upon registration.
000044	Add a checkpoint to that product on the Track screen, then look it up on the Inquiry screen.	A confirmation appears for the checkpoint, the product’s “Last updated” line refreshes, and the product is findable via the Inquiry search with its updated status.
000046	Create a new patient electronic file in the patient management section.	A confirmation message appears and the new patient record displays a PAT- prefixed ID (e.g., PAT-001) and a “Last updated” timestamp or date, visible in the patient files list.
000046	Add a new medicine in the pharmacy section (or collect a patient fee in finances).	A confirmation appears and the new record shows its prefixed ID (MED- or FEE-) with a “Last updated” value, findable in the corresponding section list.
000080	Add a new task via the Add Task input.	A confirmation message appears and the new task row displays a TASK- prefixed ID (e.g., TASK-001) and a “Last updated” timestamp or date.
000080	Edit that task’s description (or toggle it complete), then find it via search and the matching filter.	A confirmation appears, the “Last updated” value refreshes, the ID persists, and the task is findable via search and under the correct All/Active/Completed filter.
000083	Submit a real-time report on a case.	A confirmation message appears and the new report entry displays an RPT- prefixed ID with a timestamp, visible in the selected case’s report history.
000083	Log work hours for the same case.	A confirmation appears, the new hours entry shows an HRS- prefixed ID with a “Last updated” timestamp, and the case’s total hours update.
000085	Create a new project via the New Project action.	A confirmation message appears and the new project row displays a PRJ- prefixed ID (e.g., PRJ-001) and a “Last updated” date, visible in the Projects table and reflected on the dashboard.
000085	Add a schedule task (or a document), then mark the task complete.	A confirmation appears on create, and the record shows its prefixed ID (TASK- or DOC-) with a “Last updated” value that refreshes on the completion toggle.

Table 14: **Change-specific tests for Suite 3.** Each row reproduces one WebVoyager action and its expected result.

App ID	Action	Expected result
000027	Share a new promotion using the share form with all necessary details.	A confirmation message appears, and the new promotion is visible in the browse list carrying a visible “Pending review” badge or status text. It is not hidden.
000027	Search for that promotion using its title typed in mixed case with leading and trailing spaces, then search for a nonsense string such as “zzqqxx”.	The mixed-case padded query still returns the promotion. The nonsense query shows a clear empty-state message instead of a blank area or all deals.
000051	Post a new job as a client with all required details.	A confirmation appears, and the new listing shows a visible “Pending review” badge while still appearing on the jobs board and in the client’s active-postings list.
000051	Search the jobs page for that job’s title typed in mixed case with surrounding spaces, then search for a nonsense string.	The padded mixed-case query returns the job. The nonsense query shows a clear empty-state message instead of a blank list or all jobs.
000052	On the jobs page, search for “ELECTRICIAN ” (upper case with trailing spaces), then search for a nonsense string such as “zzqqxx”.	The padded upper-case query returns the electrician listings. The nonsense query shows a clear empty-state message such as “No jobs match your search” instead of a blank grid or all listings.
000053	As a company, post a new internship position.	A confirmation appears, and the new listing shows a visible “Pending review” badge while remaining visible on the student browse list and in the company’s listings.
000053	On the student panel, search for that internship’s role typed in mixed case with surrounding spaces, then search for a nonsense string.	The padded mixed-case query returns the internship. The nonsense query shows a clear empty-state message instead of a blank list or all internships.
000077	As a logged-in user, add a comment and rating to a tool.	A confirmation message appears, and the review is visible under the tool’s reviews section with a visible “Pending review” badge or status text.
000077	Search for a tool by its name typed in mixed case with leading and trailing spaces, then search for a nonsense string.	The padded mixed-case query returns the tool. The nonsense query shows a clear empty-state message instead of a blank grid or all tools.
000090	As a logged-in user, upload a picture post.	A confirmation message appears, and the new post is visible in the updates feed carrying a visible “Pending review” badge or status text. It is not hidden from the feed or the profile’s own updates.
000091	Register a new member profile with valid details.	A confirmation appears, and the newly registered profile appears in partner search results carrying a visible “Pending review” badge or status text. It is not hidden from matches.
000091	Search for that member using a keyword typed in mixed case with surrounding spaces, then search with criteria that match no one.	The padded mixed-case query still returns the member. The no-match search shows a clear empty-state message instead of a blank panel or all profiles.
000092	Join as a new member with skills filled in.	A confirmation appears, and the new member’s card shows a visible “Pending review” badge or status text in the browse-members grid. It is not hidden.
000092	Search members for one of that member’s skills typed in mixed case with surrounding spaces, then search for a nonsense skill.	The padded mixed-case query returns the member. The nonsense query shows a clear empty-state message instead of a blank result area or all members.

C Inference Cost and Cost Reduction

C.1 Inference Cost

Maintaining a shared library is not free. Table 15 reports the total LLM spend to produce one portfolio, relative to Zero-Shot. The difference comes from how dependencies are migrated (Section 3.3). SLA-FULL runs a separate migration agent per application, whereas the naive variants extract and migrate in a single call. We consider two ways to reduce this cost, migration-session reuse and fewer migration rounds. The rest of this section evaluates each.

C.2 Migration-Session Reuse

SLA-FULL gives each application its own dependency-migration agent and starts it with a fresh context every round, so an application’s code is read again in every round it survives, whether the previous round changed it or not. We therefore compare fresh sessions against reused sessions on cost, functionality, and maintainability, with everything else held fixed. Both arms start from the same round-1 portfolio and run over three suites $\times$ three trials. Table 16 shows a significant drop in both total cost and migration turns, with all six quality metrics statistically indistinguishable. The saving depends on prompt caching. Session reuse sends about 6% *more* input, but that prefix is stable across rounds and 97% of input tokens are cache hits. Repriced without a cache discount the saving falls to 1.9% and is no longer significant, whereas the drop in migration turns does not depend on pricing.

C.3 Fewer Migration Rounds

The number of migration cycles is itself a knob. Holding the portfolio fixed at eight applications, we vary m , the number of applications generated per round. The paper’s configuration is $m=2$ over four rounds, and we additionally run $m=4$ over two rounds and $m=8$ in a single round. Table 17 shows cost falling steeply as rounds are removed, from $8.3\times$ to $4.9\times$ Zero-Shot, since fewer rounds mean fewer migration passes over the portfolio. The saving comes with a loss in quality. At $m=8$, accuracy drops by 3.6%, LOC rises by 4.4%, and Verbosity rises by 11.1%.

D Backbone Robustness

To test whether the main result is specific to the coding backbone, we rerun the WebGen-Bench pro-

Method	WebGen	PaperBench
Zero-Shot	1.0	1.0
LIBRARIAN ($K=8$)	5.9	3.2
NAIVE-IMPLICIT	3.1	1.7
NAIVE-WARD	2.9	1.7
SLA-FULL	8.3	3.1

Table 15: **Cost of producing one portfolio**, as total LLM spend relative to Zero-Shot ($\times$ ZS, lower is better); same runs as Table 1. LIBRARIAN ($K=1$) is omitted: it reuses a sample already drawn for $K=8$ and has no independent cost.

tolocol with minimax-m3 (Lai et al., 2026), keeping the three $N=8$ suites, $m=2$, four-round schedule, and evaluation harness unchanged.

Table 18 shows that the core result transfers. SLA-FULL again produces the most compact portfolio, with the lowest LOC, token length, and Verbosity, and its appearance score is indistinguishable from the best.

E Low-Reuse Suite Stress Test

Each suite in the main results is taken from a single topical group of the benchmark tasks, so its applications have much in common, which is a favorable setting for a shared library. To build the opposite, we cluster all 101 WebGen-Bench tasks by embedding similarity into 14 topical groups, set aside the 24 tasks already used by the three reported suites, and take one or two of the remaining tasks from each group until we reach $N=16$. The result spans 13 groups, one having been fully consumed by the reported suites. With $m=2$, its 16 applications run over eight rounds.

Even in this unfavorable setting, SLA-FULL still produces the smallest portfolio in Table 19, with the lowest LOC, token length, and Verbosity. Its MDL is level with NAIVE-WARD, and it has the lowest Erosion of the library-building methods.

Functionality degrades in this setting, with SLA-FULL falling below Zero-Shot on both accuracy and appearance. This is what the stress test was built to create, since the applications share as little as the benchmark allows and a shared library therefore has little to reuse.

F Agentic SE Scaffold Comparison

All experiments in the paper use the minimal mini-SWE-agent scaffold. As a reference point we also run two recent agentic SE systems,

Migration configuration	Cost		Functionality		Maintainability			
	Total (\$) ↓	Migr. turns ↓	Acc. ↑	Appr. ↑	LOC ↓	MDL ↓	Eros. ↓	Verb. ↓
Fresh sessions	0.453	485	75.4	3.78	8598	34473	0.094	0.104
Reused sessions	0.400	354	76.0	3.88	8760	35122	0.104	0.114
Δ (%)	-11.7	-27.0	+0.8	+2.6	+1.9	+1.9	+11.1	+10.0
p	0.017	0.002	0.749	0.064	0.348	0.553	0.410	0.536

Table 16: **Reusing the dependency-migration session across rounds (WebGen-Bench).** Mean over three suites $\times$ three trials ($n=9$), both arms seeded from the same round-1 portfolio. p is from a two-sided paired t -test, with $p < 0.05$ in bold. Cost is the total spend to produce one portfolio, as in Table 1. Reusing the session cuts cost and migration turns significantly while every functionality and maintainability metric is statistically indistinguishable.

m	Rounds	Cost ↓	Acc. ↑	LOC ↓	Verb. ↓
2	4	8.30	77.21	8552	0.099
4	2	6.55	73.97	8643	0.095
8	1	4.93	74.44	8932	0.110

Table 17: **Applications per round (WebGen-Bench).** The portfolio is fixed at eight applications; m sets how many arrive per round, so larger m means fewer extract-and-migrate cycles. $m=2$ is the configuration used everywhere else in the paper. Cost is relative to Zero-Shot, as in Table 15; mean over three suites $\times$ three trials.

	Zero-Shot	NAIVE-I	NAIVE-W	SLA-FULL
<i>Functionality</i>				
Acc. ↑	79.17	74.51	81.68	77.45
Appr. ↑	3.92	3.86	3.90	3.91
<i>Maintainability</i>				
LOC ↓	10,957	11,305	11,152	10,581
Tok ↓	80,796	85,795	85,930	80,185
MDL ↓	46,124	51,578	49,975	48,910
Eros. ↓	0.112	0.152	0.159	0.127
Verb. ↓	0.125	0.099	0.134	0.095

Table 18: **Results with the Minimax-M3 backbone (WebGen-Bench).** Mean over three suites $\times$ three trials; **bold** marks the best method per row. Standard deviations are omitted for space and follow the same pattern as Table 1.

OPENHANDS (Wang et al., 2025a) and CLAUDE CODE,¹⁰ in the Zero-Shot slot, where each generates the eight applications of a suite independently with no shared library and no cross-codebase memory. Only the scaffold differs, with the backbone, reasoning-effort setting, task specifications, and evaluation harness as in the main results.

As Table 20 shows, the recent scaffolds are strong on some metrics. CLAUDE CODE attains the best accuracy and MDL, and OPENHANDS the

	Zero-Shot	NAIVE-I	NAIVE-W	SLA-FULL
<i>Functionality</i>				
Acc. ↑	69.02	62.58	63.37	65.88
Appr. ↑	3.66	3.48	3.52	3.29
<i>Maintainability</i>				
LOC ↓	23,263	20,744	19,900	18,557
Tok ↓	177,022	163,063	157,096	147,844
MDL ↓	76,131	72,606	67,277	67,943
Eros. ↓	0.189	0.303	0.282	0.248
Verb. ↓	0.179	0.157	0.165	0.121

Table 19: **Stress test: the hardest suite of $N=16$ tasks (WebGen-Bench).** Mean over three trials; **bold** marks the best method per row. Metrics are computed with the same evaluation harness as the main results. Standard deviations are omitted for space.

best appearance and Erosion. But SLA-FULL still leads on LOC, token length, and Verbosity, because an agent working on one codebase at a time cannot see redundancy across applications. Removing that redundancy needs a shared library, not a stronger scaffold. SLA is a layer over a scaffold, so it could be instantiated over these systems as well.

G Algorithms

Algorithm 1 gives the full procedure for the minimal scaffold (Section 3.2), used as our SLA-NAIVE comparison baseline, and Algorithm 2 gives the full procedure for SLA-FULL (Section 3.3).

¹⁰<https://www.anthropic.com/claude-code>

Method	Scaffold	Functionality		Maintainability				
		Acc. $\uparrow$	Appr. $\uparrow$	LOC $\downarrow$	Tok $\downarrow$	MDL $\downarrow$	Eros. $\downarrow$	Verb. $\downarrow$
Zero-Shot	mini-SWE-agent	76.04	3.84	9393	70364	34875	0.1032	0.1603
Zero-Shot	OPENHANDS	77.34	3.92	8948	68528	34926	0.0732	0.1489
Zero-Shot	CLAUDE CODE	78.20	3.86	8712	67857	34009	0.1125	0.1349
SLA-FULL	mini-SWE-agent	77.21	3.86	8552	65633	34195	0.0987	0.0994

Table 20: **Scaffold comparison (WebGen-Bench)**. All rows use the same backbone (deepseek-v4-flash), task specifications, and reasoning-effort setting, and only the scaffold differs. Mean over three suites $\times$ three trials.

Algorithm 1 MINIMAL AGENTIC SCAFFOLD: per round, m codebases are generated in parallel by the Coding agent, and a single Library agent then both extracts shared symbols into L_r and migrates the existing codebases to use L_r in one call.

Require: Test instances $\{x_i\}_{i=1}^N$, batch size m (with N divisible by m)

```

1:  $L_0 \leftarrow \emptyset$ 
2: for  $r = 1$  to  $N/m$  do
3:    $\mathcal{B}_r \leftarrow \{x_{(r-1)m+1}, \dots, x_{rm}\}$  ▷ batch of  $m$  instances
4:   Stage 1. Coding agent — Generate codebases
5:   in parallel for all  $i$  with  $x_i \in \mathcal{B}_r$ :
6:      $c_i \leftarrow \text{CODING}(x_i \mid L_{r-1})$ 
7:   end parallel
8:   Stage 2. Library agent — Joint extraction and migration
9:    $(\{c_j\}_{j=1}^{rm}, L_r) \leftarrow \text{LIBRARY}(\{c_t\}_{t=1}^{rm}, L_{r-1})$ 
10:  ▷ single agent jointly extracts shared symbols into  $L_r$  and rewrites local implementations as imports from  $L_r$ 
11: end for
12: return  $\{c_j\}_{j=1}^N, L_{N/m}$ 

```

Algorithm 2 SLA-FULL (ours): two-level library extraction with code-block index $\mathcal{N}_r$ and accumulated extraction trace T_r , followed by trace-guided dependency migration.

Require: Test instances $\{x_i\}_{i=1}^N$, batch size m (with N divisible by m)

```

1:  $L_0 \leftarrow \emptyset, \mathcal{N}_0 \leftarrow \emptyset, T_0 \leftarrow \emptyset$ 
2: for  $r = 1$  to  $N/m$  do
3:    $\mathcal{B}_r \leftarrow \{x_{(r-1)m+1}, \dots, x_{rm}\}$ 
4:   Stage 1. Coding agent — Generate codebases
5:   in parallel for all  $i$  with  $x_i \in \mathcal{B}_r$ :
6:      $c_i \leftarrow \text{CODING}(x_i \mid L_{r-1}, \mathcal{N}_{r-1})$ 
7:   end parallel
8:    $\mathcal{N}_r \leftarrow \text{INDEXBLOCKS}(\mathcal{N}_{r-1}, \{c_i\}_{x_i \in \mathcal{B}_r})$ 
9:   Stage 2a. Two-level library extraction (consolidation then update)
10:  in parallel for all  $i$  with  $x_i \in \mathcal{B}_r$ :
11:     $c_i \leftarrow \text{CONSOLIDATE}(c_i, \mathcal{N}_r)$  ▷ intra-codebase pre-extraction consolidation
12:  end parallel
13:   $(L_r, T_r) \leftarrow \text{LIBRARYUPDATE}(\{c_t\}_{t=1}^{rm}, L_{r-1}, T_{r-1}, \mathcal{N}_r)$ 
14:  Stage 2b. Dependency migration with extraction trace
15:  in parallel for all  $j \in \{1, \dots, rm\}$ :
16:     $c_j \leftarrow \text{MIGRATE}(c_j, L_r, T_r)$  ▷  $T_r$  pre-localizes migration sites in  $c_j$ 
17:  end parallel
18: end for
19: return  $\{c_j\}_{j=1}^N, L_{N/m}$ 

```

H Prompts and Auxiliary Inputs

H.1 Agent Prompts

This appendix summarizes what each agent is instructed to do. The verbatim text of the four agent prompts, for both benchmarks, is in our released code. Here we record the division of labor and the constraints that matter for interpreting the results.

Stage structure. SLA-FULL runs four sub-agents per round. The *coding* agent implements the round’s tasks against the benchmark specification. The *local extract* agent then consolidates duplication *within* each new codebase into a codebase-local module (the pre-extraction consolidation of Section 3.3.1). The *global extract* agent (the *library-extraction agent* of Section 3.3) grows the shared Super Library from patterns recurring *across* codebases, and the *migration* agent (the *dependency-migration agent*) rewrites each application to consume the resulting library. SLA-NAIVE omits local extraction and collapses global extraction and migration into a single-shot library agent. This difference in decomposition, not in prompt wording, is what the two conditions compare.

Cross-benchmark scope. The three library-manipulating prompts—local extract, global extract, and migration—are byte-identical across WebGen-Bench and PaperBench: they refer to code structure rather than to any domain concept. Only the coding agent and the SLA-NAIVE library agent are benchmark-specific, since they must state the target stack and the run/build contract.

Shared rule blocks. Every agent that can touch application or library code receives the same two rule blocks, which encode the constraints assumed throughout the paper: application code may import from the library but the library may never import from an application; the library must remain free of application-specific state; and each benchmark’s build and entry-point conventions must be preserved so that a refactored portfolio still runs under the evaluation harness unchanged.

H.2 Code-Index Summarization

The candidate-selection prompts in Appendix H.3 are built from natural-language summaries of code rather than from source files.

Every application codebase and the Super Library carry an index that pairs each AST-delimited

chunk¹¹ with a one-line natural-language summary. One LLM call summarizes one chunk, using the following instruction.

System Prompt (chunk summarization)

You summarize a code chunk in ONE concise English sentence (≤ 160 chars). Describe what the chunk does -- its purpose and notable mechanism if non-obvious. No quotes, no preamble like 'This chunk'. Output only the summary.

The user message carries only the chunk itself, prefixed by its file path, line span, and language. An index entry stores the language, file path, line span, a content hash, and the summary. The hash makes the index incremental. A chunk whose hash is unchanged keeps its stored summary, so each round pays only for code that actually changed. Chunks shorter than five lines are skipped.

An index is rendered into a prompt as one line per chunk, the format shown under ## LIBRARY and ## APP in Appendix H.3. Two example entries follow.

Rendered index lines

```
- chunk_id=6 [40L] src/components/Button.jsx:1
  -- Defines a reusable themed Button
  component that composes CSS classes from
  variant/size/block props (with defaulting)
  and renders a native button.
- chunk_id=23 [23L] src/hooks/useSearchFilter.
  js:1 -- Defines a React hook scaffold that
  will track searchQuery and activeCategory
  state and use filterBySearch to compute
  filtered items.
```

H.3 Candidate-Selection Prompts

H.3.1 Extraction Candidates

System Prompt

You are a senior library architect deciding which code patterns to extract from multiple Python codebases into a shared library. You will be given one-line summaries of every chunk across all apps, plus summaries of the EXISTING library (treat lib symbols as 'already extracted' -- do NOT propose duplicates).

Identify the TOP-K distinct, high-value cross-app patterns that appear in TWO OR MORE apps and would be genuinely worth extracting (not trivial helpers, not app-specific glue, not

¹¹We use cocoindex-code. <https://github.com/cocoindex-io/cocoindex-code>

duplicates of existing lib).

For each candidate, output exactly:

```
### C<n>. <short pattern name>
**Pattern**: <1 sentence describing the
shared behavior>
**Why extractable**: <1 sentence on why this
generalizes>
**Members**:
- <app>::<file_path>:<start>-<end>::<
chunk_id=<id>> -- <summary>
- ... (>=2 distinct apps required)
```

Be conservative -- false positives matter more than coverage.

User Prompt (with sample indices)

```
### EXISTING LIBRARY (already extracted -- do
NOT duplicate)
```

```
## LIBRARY
- chunk_id=1 [8L] src/index.js:1 -- Re-
exports custom hooks and React components
from internal files so consumers can import
them via this package entry point.
- chunk_id=2 [42L] src/components/Header.css:1
-- Defines a sticky dark header bar layout
using flexbox, styling the brand for focus
accessibility, and setting up a horizontally
spaced navigation button area.
```

(skip)

```
### APPS (8 apps, 794 chunks total)
```

```
## App: 000027
- chunk_id=1 [21L] src/App.jsx:14 -- Defines
the App component with persisted currentPage
and isSignedIn state via useLocalStorage,
plus navigate/signOut/signIn handlers
updating both.
- chunk_id=2 [28L] src/App.jsx:36 -- Maps the
currentPage string to the corresponding
React component, passing navigate (and signIn
for auth-login), defaulting to Home.
```

```
## App: 000051
```

(skip)

Identify the top-10 cross-app extraction candidates.

Assistant Response (with sample candidates)

```
### C1. Search+Category Bar (query + category
+ results count)
**Pattern**: Renders a controlled search
input plus a category selector, derives/filter
results, and displays a count-based summary
(often with an empty state).
```

```
**Why extractable**: The underlying `
useSearchFilter` hook is already shared, but
the *UI shell* around it (input + category
control + count/empty messaging) is still
duplicated across apps with only small
behavioral differences.
```

```
**Members**:
- 000027::src/components/BrowseScreen.jsx
:17-43::chunk_id=75 -- Renders a "Find a Deal"
search UI with controlled input and category
dropdown, populating options from categories.
```

```
- 000051::src/components/JobScreen/
JobScreen.jsx:13-39::chunk_id=167 -- Renders
a Jobs screen header with controlled search
input and clickable category filter pills
updating selected category.
```

```
- 000053::src/Home.jsx:21-36::chunk_id=158
-- Renders a controlled search input and
category dropdown, updating state on typing/
selection changes.
```

```
- 000092::src/components/Search.jsx:1-7::
chunk_id=149 -- Initializes search filters (
search term + category selection defaulting
to "All").
```

```
### C2. App Shell / Page Layout (Header +
main content + Footer)
```

```
**Pattern**: Standardizes each app's top-
level layout by composing a shared Header, a
main content region (page switch/routing),
and a shared Footer.
```

```
**Why extractable**: Even with shared `Header
`/`Footer` components already extracted, the *
shell composition* (main container,
renderPage wiring, screen switching placement)
is still duplicated across apps.
```

```
**Members**:
```

```
- 000027::src/App.jsx:110-142::chunk_id=59
-- Renders the app layout with a shared header
/footer and conditionally shows Browse/Share/
Admin screens.
```

```
- 000051::src/App.jsx:65-74::chunk_id=58 --
Renders the app layout with Header (
navigation/auth props), main content from `
renderPage()`, and a Footer.
```

```
- 000092::src/App.jsx:116-149::chunk_id=62
-- Wraps profile-related screen content with
Header and a main layout region.
```

(skip)

H.3.2 Migration Candidates

System Prompt

You are a senior engineer migrating an app to use a shared utility library. Given (a) one-line summaries of every symbol in the library and (b) summaries of every chunk in ONE specific app, pick the TOP-K library symbols this app should adopt -- i.e., places where the app currently has its own implementation that the library symbol could replace.

For each candidate, output exactly:

```

### A<n>. <lib symbol>
**Library**: <lib_file>:<start>-<end>::<
chunk_id=<id>> -- <lib summary>
**Replaces in this app**:
- <app_file>:<start>-<end>::<chunk_id=<id>>
-- <reason>
- ... (one or more places)

```

Be conservative. If fewer than the requested number of strong matches exist, output only the strong ones. If NONE, output exactly 'NONE'.

User Prompt (with sample indices)

```

### Library (37 symbols)

## LIBRARY
- chunk_id=11 [20L] src/hooks/useLocalStorage.
js:15 -- Implements a React hook that syncs
state with localStorage by reading/parsing an
initial value and persisting updates as JSON
with error-suppressed failures.
- chunk_id=23 [28L] src/hooks/useSearchFilter.
js:70 -- Memoizes and returns a filtered data
array by lowercasing/trimming the search
query, matching it against configured fields,
and applying an optional category filter.

```

(skip)

Target App: 000027 (94 chunks)

```

## APP
- chunk_id=160 [28L] src/App.jsx:1 -- Imports
localStorage helpers and page components,
then declares the app-level persistence
approach.
- chunk_id=162 [27L] src/App.jsx:64 --
Manages and persists promotions and app state
via localStorage with custom update logic.

```

(skip)

Pick the top-10 migration candidates.

Assistant Response (with sample candidates)

```

### A1. useLocalStorage
**Library**: src/hooks/useLocalStorage.js
:15-34::chunk_id=11 -- Implements a React hook
that syncs state with localStorage by
reading/parsing an initial value and
persisting updates as JSON with error-
suppressed failures.
**Replaces in this app**:
- src/App.jsx:64-90::chunk_id=162 -- App
manages and persists promotions/app state via
localStorage with custom update logic (can
be replaced by the shared useLocalStorage
contract).
- src/App.jsx:1-28::chunk_id=160 -- App

```

imports localStorage helpers and owns the persistence approach (can be replaced by adopting the shared useLocalStorage hook).

```

### A2. useSearchFilter
**Library**: src/hooks/useSearchFilter.js
:70-97::chunk_id=23 -- Memoizes and returns a
filtered data array by lowercasing/trimming
the search query, matching it against
configured fields, and applying an optional
category filter.
**Replaces in this app**:
- src/components/BrowseScreen.jsx:1-18::
chunk_id=167 -- BrowseScreen initializes its
own search/category state and performs
filtering logic over promotions (directly
matches the hook's responsibility).
- src/components/BrowseScreen.jsx:17-43::
chunk_id=75 -- BrowseScreen builds the
controlled "Find a Deal" search + category UI
and derives counts from filtered results (
should instead consume the hook's derived
categories + filtered list).

```

(skip)

H.4 Extraction Trace

After each library extraction, the library-extraction agent writes a markdown document for every new or updated library symbol, recording where the symbol came from, why the pattern generalizes, and how to replace the original code with the library API. The file is cumulative with later rounds appending newly extracted symbols and revising existing entries whose lib-side implementation changed.

The example below generalizes two independent localStorage usage patterns into a single useLocalStorage hook.

Sample Extract Trace Entry

```

## library symbol: `useLocalStorage`

**Module**: `src/hooks/useLocalStorage.js`

**Sources** (tasks the pattern was
generalized from)

| Task | File | Original Symbol |
|-----|-----|-----|
| 000019 | App.jsx (L18-25, 102-119)
| inline useState init + handler |
| 000075 | ContactUs/ContactUs.jsx (L23-25)
| inline localStorage get/set |

**Why generalized**
Both tasks read and write contact-submission
data to localStorage with the
same load-modify-save cycle. 000019 wraps the
pattern in a React lazy-
initializer state and a callback that appends

```

```

to the stored array; 000075
directly calls localStorage.getItem and
setItem from the submit handler. The
common skeleton -- load JSON from a key, fall
back to a default, persist
updates -- is captured by this hook so that
any component can declare
`const [data, setData, clearData] =
useLocalStorage(key, [])`.

**Apply guidance**
Replace inline `useState(() => { try {...} })`
+ manual `localStorage.setItem`
calls with `useLocalStorage(key, initialValue)`
. The hook returns a state
tuple where `setValue` automatically persists
the serialised value. Apps no
longer need `JSON.parse` / `JSON.stringify` /
`try-catch` boilerplate.

```

Every entry has the same four parts, namely the library symbol with its module path, a **Sources** table keyed by task ID and file location, the justification under **Why generalized**, and the migration instructions under **Apply guidance**.

H.5 Call-Graph Context

A file cannot be migrated in isolation. Other files import it, and they break if they are not updated too. We therefore append the related files to each migration candidate of Appendix H.3.2 (Section 3.3.2).

No LLM is involved. We build a file-level dependency graph from the import statements of the application and the library. For each file the candidate cites, we then list the files that import it and the files it imports. Modules that only re-export are skipped, and each side is capped at eight files.

In the candidate below, the library’s NotificationBanner replaces the application’s duplicated copy. Call-graph injection then appends the callers of that file (AdminPage.jsx and SharePage.jsx) and the file it imports (NotificationBanner.css), so the agent is asked to update those too. Long summaries are elided here, and arrows appear as ^ and v.

Migration candidate, as returned by the selector

```

### A1. NotificationBanner
**Library**: src/components/NotificationBanner
.jsx:24-52::chunk_id=44 -- Renders a typed
notification alert with message text and an
optional dismiss button [...]
**Replaces in this app**:
- src/NotificationBanner.jsx:1-18::chunk_id
=63 -- The app defines its own simple
notification banner component (green / red)
[...]

```

The same candidate after call-graph injection

```

### A1. NotificationBanner
**Library**: src/components/NotificationBanner
.jsx:24-52::chunk_id=44 -- Renders a typed
notification alert with message text and an
optional dismiss button [...]
**Replaces in this app**:
- src/NotificationBanner.jsx:1-18::chunk_id
=63 -- The app defines its own simple
notification banner component (green / red)
[...]
**Also refactor** (neighbors of src/
NotificationBanner.jsx):
^ Callers (import this file) : src/
AdminPage.jsx, src/SharePage.jsx
v Imports (used by this file) : src/
NotificationBanner.css

```

I Library Export Listings

We report the complete Super Library export inventory underlying the abstraction-quality analysis in Figure 4. Each table corresponds to one WebGenBench suite, which contains $N = 8$ related applications, and each row corresponds to one trial of a method variant. For every trial, we list all exported symbols together with their reuse rate, defined as the percentage of applications in the suite that import the symbol. Exports are sorted by reuse rate in descending order. We include low- and zero-reuse exports as well, since they reveal whether a method produces unused or overly specific library code in addition to broadly reused abstractions.

J Responsible NLP Checklist Details

This appendix expands on the items listed in the Responsible NLP Checklist that did not fit in the main text.

J.1 Scientific Artifacts

Artifacts we use. Our experiments rely on the following external artifacts: the **WebGenBench** benchmark (Lu et al., 2025), the **PaperBench** Code-Dev benchmark (Starace et al., 2025), the **WebVoyager** UI agent used as WebGenBench’s executable test runner (He et al., 2024), the **mini-SWE-agent** harness (Yang et al., 2024), the **cocoindex-code** code-indexing tool used by our Global Extract / Apply candidate selectors, and the **Qwen-2.5-7B** reference model used to compute MDL. We additionally call hosted LLM APIs (deepseek-v4-flash, gpt-5-mini, gpt-5.4-nano, Claude Opus 4.7) for code generation and judging. License information is summarized in Table 24.

Method	Trial	#	Exported symbols with reuse rate
NAIVE-IMPLICIT	T1	6	Footer (100%), Header (88%), ContactForm (62%), PillNav (50%), useLocalStorage (50%), SkillBar (25%).
NAIVE-IMPLICIT	T2	5	Footer (100%), Header (88%), ContactForm (62%), useLocalStorage (62%), SkillBar (25%).
NAIVE-IMPLICIT	T3	6	Footer (100%), ContactForm (50%), NavBar (50%), FilterGroup (38%), useLocalStorage (38%), SkillBar (25%).
NAIVE-WARD	T1	4	Footer (100%), Header (62%), ContactSection (38%), SkillsSection (25%).
NAIVE-WARD	T2	5	Footer (100%), Header (88%), ContactForm (62%), SkillsChart (25%), useLocalStorage (25%).
NAIVE-WARD	T3	7	Footer (88%), Header (62%), ContactForm (50%), Section (38%), loadStorage (38%), saveStorage (38%), SkillBar (25%).
SLA-FULL	T1	13	Footer (100%), NavigationBar (100%), Section (100%), TagPill (62%), Hero (50%), useLocalStorage (50%), validateContactFields (50%), FilterBar (38%), saveSubmission (38%), ProgressBar (25%), createSubmission (25%), validateRequiredFields (25%), loadSubmissions (0%).
SLA-FULL	T2	14	NavHeader (100%), FeatureCardGrid (62%), FormField (62%), HeroSection (62%), Tag (62%), useContactForm (62%), ProjectCards (50%), FilterGroup (38%), StatsGrid (38%), saveToStorage (38%), SkillBars (25%), loadFromStorage (25%), defaultContactValidator (12%), removeFromStorage (0%).
SLA-FULL	T3	11	Footer (100%), ContactForm (75%), Header (62%), Section (62%), PillNav (50%), Timeline (50%), useLocalStorage (50%), NewsCard (38%), SkillBar (25%), FormField (12%), validateRequiredFields (12%).

Table 21: Complete Super Library export inventory for WebGen-Bench Suite 1. Reuse rate denotes the percentage of the eight applications in the suite that import each symbol.

Method	Trial	#	Exported symbols with reuse rate
NAIVE-IMPLICIT	T1	10	Button (100%), FormField (88%), Card (75%), useLocalStorage (75%), ErrorMessage (62%), NavBar (50%), StatusPill (50%), SearchInput (38%), loadStorage (0%), saveStorage (0%).
NAIVE-IMPLICIT	T2	10	useLocalStorage (100%), Button (88%), Card (88%), FormField (75%), KpiCard (75%), Footer (50%), NavBar (50%), SearchBar (38%), PageHeader (12%), DataTable (0%).
NAIVE-IMPLICIT	T3	9	Footer (100%), Header (100%), NavBar (88%), Badge (62%), useLocalStorage (62%), Card (50%), loadStorage (38%), saveStorage (12%), StatCard (0%).
NAIVE-WARD	T1	7	Button (88%), Card (88%), useLocalStorage (62%), Footer (50%), Header (50%), NavBar (50%), Sidebar (25%).
NAIVE-WARD	T2	9	Button (100%), Card (88%), useLocalStorage (75%), FormField (62%), StatusPill (62%), StatCard (50%), Table (50%), NavTabs (38%), Alert (12%).
NAIVE-WARD	T3	14	Button (88%), useLocalStorage (75%), Card (62%), StatCard (62%), Badge (50%), FormGroup (50%), NavBar (50%), DataTable (38%), PageHeader (25%), SearchBar (25%), Sidebar (25%), EmptyState (12%), loadData (0%), saveData (0%).
SLA-FULL	T1	20	NavBar (88%), generateId (88%), useLocalStorage (88%), Badge (75%), EmptyState (75%), KpiCard (75%), Selector (75%), ErrorMessage (62%), formatCurrency (50%), updateItemById (50%), useFilteredList (50%), ProgressBar (38%), deleteItemById (38%), filterByKey (38%), formatDate (38%), FilterBar (25%), StepProgress (25%), formatDateTime (25%), seedLocalStorage (25%), todayISO (12%).
SLA-FULL	T2	11	addItem (100%), useLocalStorage (100%), EmptyState (88%), NavBar (75%), sumBy (62%), formatCurrency (50%), updateItem (50%), useFilter (50%), formatDate (38%), toggleField (38%), removeByIndex (25%).
SLA-FULL	T3	10	ControlledSelect (88%), NavigationBar (88%), StatCard (75%), useLocalStorage (75%), Badge (62%), EmptyState (50%), SearchFilterBar (50%), StatusMessage (38%), loadFromLocalStorage (12%), saveToLocalStorage (0%).

Table 22: Complete Super Library export inventory for WebGen-Bench Suite 2. Reuse rate denotes the percentage of the eight applications in the suite that import each symbol.

Consistency with intended use. All listed artifacts are released for open use in research on automated code generation, web agent evaluation, or language modeling. Our use — evaluating an LLM-based multi-agent coding system on the public task splits, without redistributing benchmark data, grader rubrics, or ground-truth code — is consistent with each artifact’s stated intended use. We do not modify any benchmark task descriptions or graders; we only adapt them into our *suite* format by sampling disjoint task subsequences (§4.1.1).

Artifacts we create. The artifacts produced by this work — the source code of the system, the metric and evaluation result files behind every reported number, and the final-round application codebases together with the Super Libraries extracted from them — are released under the MIT license. The generated codebases are produced from benchmark

task specifications and are intended as evidence for the claims in this paper rather than as deployable applications.

J.2 AI Assistants

We disclose the use of AI assistants in two distinct roles.

As research subjects. LLMs and LLM-based coding agents are the *object of study* in this paper. Specifically, deepseek-v4-flash is used as the backbone for all compared agents, gpt-5.4-nano produces per-block code summaries inside our pipeline, gpt-5-mini serves as the appearance judge and the report parser, and Claude Opus 4.7 is used to render the reference layout descriptions provided to all methods. These usages are described in §4.1.2 and §4.2 and are integral to the experimental setup, not authoring aids.

Method	Trial	#	Exported symbols with reuse rate
NAIVE-IMPLICIT	T1	8	Alert (100%), loadStorage (100%), saveStorage (100%), Footer (62%), FileInput (38%), SearchBar (38%), useFileUpload (25%), useLocalStorage (0%).
NAIVE-IMPLICIT	T2	12	Alert (100%), FormField (100%), useLocalStorage (100%), validateRequired (50%), FilePicker (38%), Button (25%), SearchBar (25%), validateEmail (25%), validateMinLength (25%), Modal (12%), Pill (12%), Footer (0%).
NAIVE-IMPLICIT	T3	8	useFormValidation (75%), Footer (62%), createKeyedStorage (50%), loadStorage (50%), saveStorage (50%), Chip (38%), FileUpload (38%), SearchFilter (38%).
NAIVE-WARD	T1	5	Header (100%), loadStorage (100%), saveStorage (100%), Footer (75%), FormField (75%).
NAIVE-WARD	T2	8	Button (62%), Footer (62%), FormField (62%), createAuthContext (62%), Modal (25%), useLocalStorage (12%), getStorage (0%), setStorage (0%).
NAIVE-WARD	T3	6	loadJSON (100%), saveJSON (100%), Footer (62%), ImageUpload (38%), LoginForm (38%), RegisterForm (25%).
SLA-FULL	T1	13	loadStorage (100%), saveStorage (100%), validateRequired (100%), Header (88%), MessageBanner (88%), useFilteredList (75%), Footer (62%), Pill (62%), EmptyState (50%), TabBar (38%), fileToDataURL (38%), validateMinLength (38%), renderPills (12%).
SLA-FULL	T2	17	useLocalStorage (100%), useRouter (100%), validateRequired (100%), useFormError (88%), FormField (75%), NotificationBanner (75%), Pill (75%), validateAll (75%), EmptyState (62%), Footer (50%), TabBar (50%), useSearchFilter (50%), usePhotoUpload (38%), validateEmail (38%), getTodayDate (25%), useFieldErrors (25%), filterBySearch (12%).
SLA-FULL	T3	11	FormField (100%), Card (88%), ErrorMessage (88%), loadFromStorage (88%), saveToStorage (88%), usePersistentList (88%), EmptyState (75%), useFilteredList (75%), Pill (62%), Grid (62%), useImageUpload (38%).

Table 23: Complete Super Library export inventory for WebGen-Bench Suite 3. Reuse rate denotes the percentage of the eight applications in the suite that import each symbol.

Artifact	License	Use
WebGen-Bench	MIT	eval suite
PaperBench	MIT	eval suite
WebVoyager	Apache 2.0	UI runner
mini-SWE-agent	MIT	harness
cocoindex-code	Apache 2.0	indexing
Qwen2.5-7B	Apache 2.0	MDL ref.

Table 24: External artifacts and their licenses. All are permissive open-source licenses, and our use is consistent with each artifact’s intended purpose.

As research and engineering aids. During this project we used **ChatGPT (GPT-5 family)** and **Claude (Opus 4.7)** as auxiliary tools in three bounded roles. They sanity-checked the logical flow and consistency of arguments in draft sections, stress-tested research ideas and surfaced counter-examples or related work we might have missed, and assisted with implementation-level code generation such as boilerplate analysis scripts and plotting code. All scientific claims, experimental designs, analyses, and final interpretations are authored and verified by the human authors; AI assistants did not autonomously produce any of the experimental results, ablations, or conclusions reported in this paper.

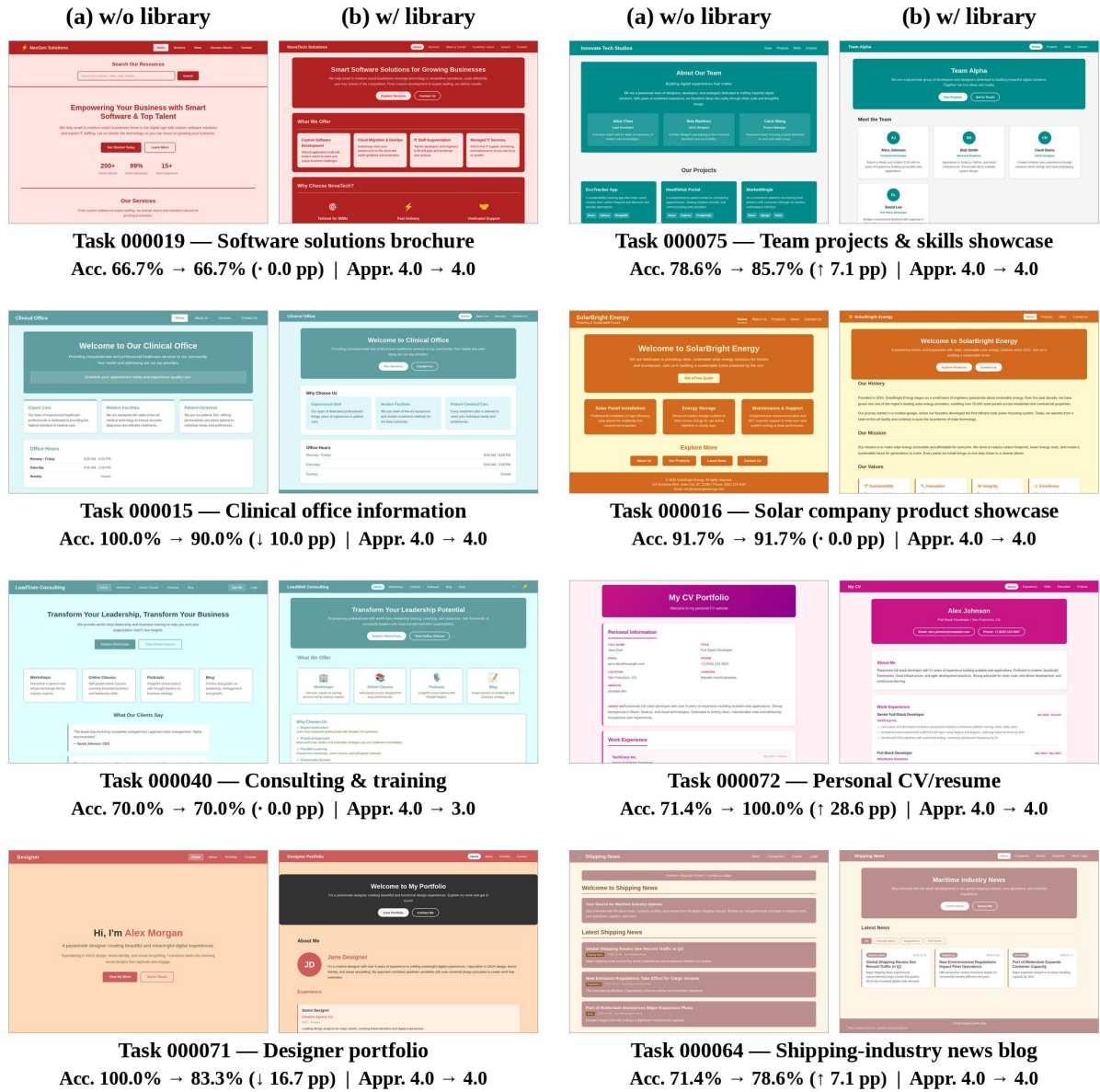


Figure 7: **All eight Content-Presentation tasks under trial 1.** Each task is shown as a (without library, with library) pair; the eight tasks are tiled as four rows of two tasks. UI-test accuracy and appearance grade are reported below each pair.