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

Anonymous ACL submission

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 improving maintainability metrics such as LOC, MDL, structural erosion, and verbosity over zero-shot and naive library-construction baselines.

1 Introduction

LLM coding agents can now generate non-trivial applications end-to-end, but real-world software development rarely produces a single application in isolation. Organizations often maintain *portfolios* of related applications: 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 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); LLM-generated portfolios make the problem easier to create at scale.

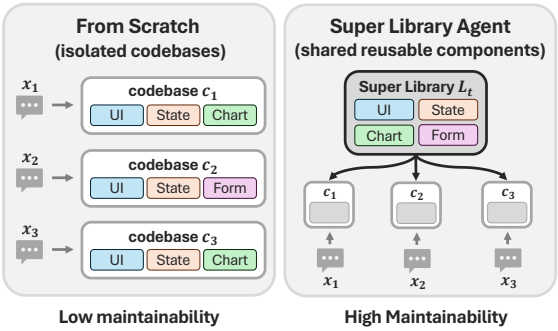


Figure 1: **Super Library Agent (SLA) problem.** Independent generation duplicates shared logic; SLA maintains a shared library across a sequence of codebases.

Furthermore, the issue is compounded by the maintenance behavior of LLM coding agents. 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 every maintenance cycle. A method that explicitly factors and re-factors shared concerns across the portfolio is therefore 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). Recent work, such as Librarian (Kovacic et al., 2025), as-

sumes 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 identify functionally reusable components across growing codebases. Embedding-based or implicit retrieval is often insufficient: code fragments with similar intent may differ substantially in surface form, while superficially similar fragments may implement different behavior. As the portfolio grows, the extraction agent therefore misses reusable components, limiting the quality and coverage 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 extraction recall, we build summarized code indices 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, where an agent sequentially generates related applications while maintaining a shared library of reusable cross-application components.
- We identify two key obstacles in naive library construction—low extraction recall and fragile dependency migration—and propose candidate-guided extraction and context-aware migration to address them.
- We evaluate on WebGen-Bench and PaperBench, showing that our method preserves functionality while improving maintainability metrics, library utilization, and abstraction quality over zero-shot and naive baselines.

2 Related Work

Maintainable code generation. Recent code generation work increasingly treats maintainability as an explicit objective. MaintainCoder (Wang et al., 2025) 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. Librarian (Kovacic et al., 2025) clusters model-written file summaries, samples candidate libraries and rewrites, and reranks them with MDL and tests; 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 $\mathbf{X} = (x_1, \ldots, x_N)$ 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 $\mathbf{C}_{<t} = \{c_1, \ldots, c_{t-1}\}$, and the current library $\mathcal{L}_{t-1}$, the agent $\mathcal{A}$ produces

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

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:

$$\mathcal{L}^\star_t = \left\{ u \,\middle|\, \sum_{i=1}^{t} \mathbf{1}[\mathrm{use}(u, c_i)] \geq 2 \right\}.$$

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

We consider a simplified sequential setting where each application has only one initial implementation request and no later issue patches. *The goal is to produce final applications* $\mathbf{C}_N$ *and library* $\mathcal{L}_N$ *that achieve a* ***Pareto-optimal trade-off between functionality and maintainability***:

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

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 $(c_t, \mathcal{L}_t, \mathbf{C}'_{<t})$ 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 scaffold is minimal in that all library updates and codebase migration are delegated to one agent per round, with no specialized candidate retrieval or hand-off mechanism between extraction and migration. We provide the full procedure in Appendix C; our experiments in Section 4 use this scaffold as the 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. To address both, we first decompose the single library agent into two specialized agents—a *library-update agent* responsible for extraction and a *dependency-migration agent* responsible for patching previous codebases—and 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.

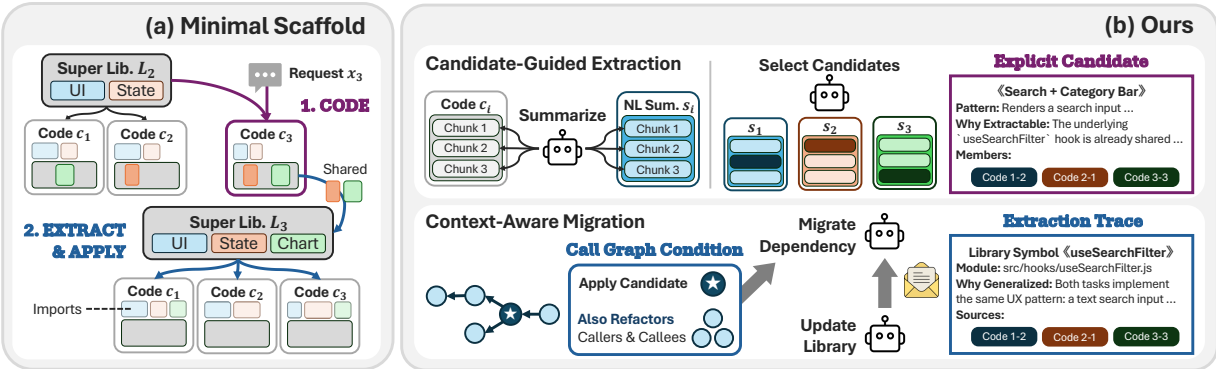


Figure 2: **Overview of the proposed workflow.** Minimal scaffold NAIVE (left) vs. FULL (right).

3.3.1 Index-based Candidate Extraction

To improve extraction and migration recall, we explicitly provide the library-update agent with reuse candidates selected from lightweight code indices, rather than relying only on the model's implicit retrieval ability.

Code Block Indexing. We maintain index files for both application codebases and the Super Library. Each repository is partitioned into semantically meaningful code blocks using abstract syntax tree (AST) boundaries, such as functions, classes, and modules.[1] An LLM then generates a concise natural-language summary for each block.[2] These summaries provide a compact representation of the codebase and allow agents to locate reusable functionality without repeatedly inspecting full source files.

LLM-based Candidate Selection. Given the up-to-date indices, a dedicated agent selects candidates for both library extraction and migration. For extraction, it identifies groups of code blocks from multiple applications that implement shared functionality and proposes them as Super Library candidates. For migration, the agent performs this matching separately for each codebase, identifying application-local blocks that can be replaced by existing Super Library components. We provide the detailed prompts and example outputs in Appendix E.

Pre-Extraction Codebase Consolidation. Because AI-generated code often contains redundancy even within a single codebase, we first consolidate local redundancy before selecting global Super Library candidates. For each newly generated codebase, the agent uses explicit candidate extraction to identify duplicated or overlapping implementations within the codebase, then performs a one-time extraction-and-migration step to refactor them into shared codebase-local modules. This does not create a separate local library: the consolidated components remain inside the application codebase. The consolidated codebase is then indexed and used for global Super Library extraction, improving candidate recall while reducing noise from redundant local implementations.

3.3.2 Context-Aware Dependency Migration

After shared components are extracted into the Super Library, existing applications must be migrated to use the updated library. This is non-trivial because extraction and migration are performed by separate agents, and replacing a local component may also require updating its imports, callers, and dependent helper functions. We therefore provide the dependency-migration agent with explicit migration context.

Extraction Trace-based Bridging. After each global extraction, the library-update agent records an extraction trace that maps each new or updated Super Library component to the original code blocks it consolidated. This trace serves as a handoff between the extraction and migration phases, allowing the dependency-migration agent to identify where the new library component should be applied without rediscovering the correspondence from scratch. We provide an example extraction trace in Appendix F.

Call Graph Conditioning. For each selected application candidate, we expose file-level imports and local caller/callee relationships as an *also-refactor* context. This call-graph context guides the agent to update surrounding call sites, adjust imports, and remove obsolete local implementations. It reduces migration failures caused by bro-

[1] We use `cocoindex-code` for this. https://github.com/cocoindex-io/cocoindex-code

[2] gpt-5.4-nano with high reasoning effort is used.

ken references, duplicated code, or dead code.

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 $X = (x_1, \ldots, x_N)$, 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 B.1.

PaperBench. PaperBench (Starace et al., 2025) evaluates whether coding agents can reproduce published AI research contributions from paper descriptions. We use the *Code-Dev* variant, which uses an LLM judge to grade code-development criteria without execution or result matching. We construct three disjoint 4-task suites from ICML 2024 Spotlight and Oral papers, grouped into RL, VLM, and NLP; the per-suite task list is given in Appendix B.3.1. PaperBench tests reuse in research-code abstractions such as training loops, data pipelines, model wrappers, tokenizers, and evaluation utilities.

4.1.2 Compared Methods

All methods use the same backbone LLM, `deepseek-v4-flash` (DeepSeek-AI, 2026), and `mini-SWE-agent` harness (Yang et al., 2024). [3]

- **Zero-shot** generates each codebase independently, without a shared library or cross-codebase memory.
- **SLA-Naive** uses the minimal sequential scaffold, including library update and dependency migration, but without our proposed extraction and migration strategies; we report both implicit and Ward-clustering candidate variants.
- **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 (↑)** 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). **Appearance Score (↑)** is a 1-5 LLM judge score on the rendered entry page, measuring visual quality.[4]

PaperBench. We report the **Code-Dev Score (↑)** (Starace et al., 2025), a normalized 0-1 LLM-judge score that checks the generated implementation against paper-specific rubric items.[5]

4.2.2 Code Maintainability

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

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

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

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 and adapt the original file-independent formulation to our setting with a dependency-aware variant:

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

[3] `mini-SWE-agent` is a minimal-scaffold variant maintained by the SWE-agent authors; per the official guidance, we cite the SWE-agent paper.

[4] `gpt-5.4-mini` with high reasoning effort is used.

[5] `deepseek-v4-flash` with high reasoning effort is used as main evaluator, and `gpt-5.4-mini` with default reasoning effort is used as evaluation report parser.

	Zero-Shot	Naive		Full
		Implicit	Ward-selected	
Functionality				
Acc. ↑	76.24	76.95	76.07	**77.21**
Appear. ↑	3.84	3.89	**3.90**	3.86
Maintainability				
LOC ↓	9393	9133	8786	**8552**
Tok ↓	70364	69777	67774	**65633**
MDL ↓	34870	35103	34675	**34195**
Eros. ↓	0.1032	0.1567	0.1250	**0.0987**
Verb. ↓	0.1603	0.1408	0.1370	**0.0994**
Library Size				
LOC	–	447	433	574
Tok	–	3173	3090	4143
MDL	–	1212	1175	1630

Table 1: **Main results on WebGen-Bench**, mean over three disjoint suites of 8 tasks (24 total).

	Zero-Shot	Naive		Full
		Implicit	Ward-selected	
Functionality				
Score ↑	0.4088	0.4095	0.4144	**0.4145**
Maintainability				
LOC ↓	6763	6561	**6507**	6555
Tok ↓	69115	66786	65187	**64641**
MDL ↓	31357	30405	**29628**	30656
Eros. ↓	0.2324	0.2529	0.2525	**0.2282**
Verb. ↓	0.8609	0.8505	0.8615	**0.8434**
Library Size				
LOC	–	197	298	345
Tok	–	1826	2712	3161
MDL	–	672	980	1138

Table 2: **Main results on PaperBench,** averaged over three disjoint suites of 4 tasks each. We report mean scores over 12 tasks without standard deviations.

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

Structural Erosion (↓). Structural Erosion (Orlanski et al., 2026) is the fraction of total cyclomatic-complexity mass concentrated in high-complexity functions, defined as functions with cyclomatic complexity greater than 10.

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 Main Results

WebGen-Bench. Table 1 shows that all methods achieve comparable functionality, indicating that library use does not harm task performance. SLA-FULL obtains the best scores on all maintainability metrics, reducing total LOC, token length, MDL, Erosion, and Verbosity relative to zero-shot. In contrast, both SLA-NAIVE variants reduce some size metrics but increase Erosion, suggesting that unguided library extraction can concentrate complexity into shared components. Thus, targeted extraction and migration are necessary to improve maintainability without functional regression.

PaperBench. Table 2 shows a similar but weaker trend. SLA-FULL matches or slightly improves Code-Dev Score while achieving the best structural metrics, especially Erosion and Verbosity. The size-based gains are smaller and less uniform than in WebGen-Bench, consistent with research-code tasks offering fewer repeated UI-style abstractions. Overall, SLA-FULL provides the most stable functionality–maintainability trade-off across both domains.

5 Analysis

Beyond the aggregate results in Section 4, we analyze how FULL improves reuse and maintainability over sequential application generation. We focus on four aspects: component ablations, maintainability dynamics, final library quality, and the effect of mature libraries on coding behavior.

5.1 Ablation Study

In Table 3, we compare different candidate selection methods on SLA-NAIVE, and ablate the major components of FULL to isolate the contribution of each design choice.

Candidate selection for library update. In Table 3a, we compare three ways of providing reuse candidates to the library-update agent in the SLA-NAIVE setting: no explicit candidates, Ward-clustering-based candidates, and our index-based candidate selection. This ablation tests whether explicit retrieval is necessary for discovering reusable cross-application patterns as the number of codebases grows. 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 functionality scores are down in NL due to the additional shared component extraction introducing opportunities for the agent to introduce new bugs. We ad-

	Functionality		Maintainability					Library Size		
Selec.	Acc. ↑	Appr. ↑	LOC ↓	Tok ↓	MDL ↓	Eros. ↓	Verb. ↓	LOC	Tok	MDL
None	**76.95**	3.89	9133	69777	35103	0.1567	0.1408	447	3173	1212
Ward	76.07	**3.90**	8786	67774	34675	0.1250	**0.1370**	433	3090	1175
NL	72.95	3.89	**8583**	**66727**	**32893**	**0.1122**	0.1376	815	5944	2057

(a) Candidate selection: **None**, **Ward** (hierarchical clustering over code-block embeddings), and **NL** (natural-language code-block summary matching) on three SLA-NAIVE variants.

Metric	no LC	no CG	SLA-Full
Acc. ↑	75.25	74.48	**76.90**
Verb. ↓	0.1264	0.1215	0.0994
Eros. ↓	0.1263	0.1409	**0.0987**
LOC ↓	8733	9068	8552
MDL ↓	33880	34166	34195

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

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

dress this inaccuracy by using call graph conditioning in our 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 global 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 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 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.

Context-aware dependency migration. We ablate the migration context by removing extraction traces and call-graph (CG) conditioning, in order to test whether explicit bridging information and structural caller/callee context are necessary for reliably replacing local implementations with SL calls. In Table 3b, as expected, the accuracy of the no-CG variant is lower, which is due to the apply agent failing to properly handle the structural dependency updates. Also, without CG, the apply agent often leaves dead code after replacing the local implementation with a SL call, which is reflected in higher verbosity, erosion, and LOC.

5.2 Maintainability Dynamics Across Rounds

We track maintainability metrics such as LOC and MDL after coding and dependency-application phases across sequential rounds. This analysis shows when reuse is accumulated, rather than only comparing final scores.

SLA-FULL tends to reduce application-local code as more applications are added to the stream. The largest decrease often occurs immediately after an application first enters the extract-and-apply workflow, where shared components can be moved into the Super Library and local implementations can be replaced. In contrast, the naive scaffold shows flatter trajectories, suggesting that a shared library alone is insufficient without reliable candidate discovery and migration.

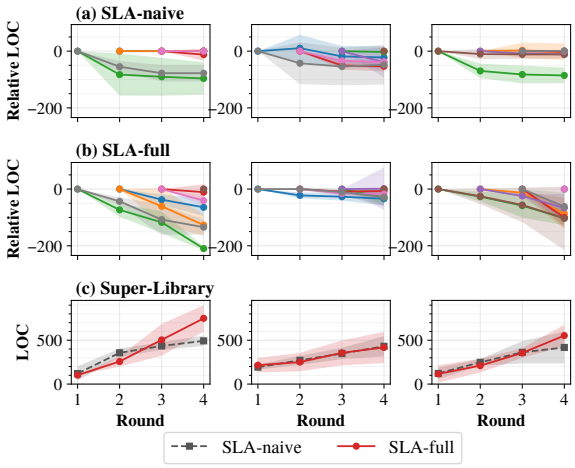


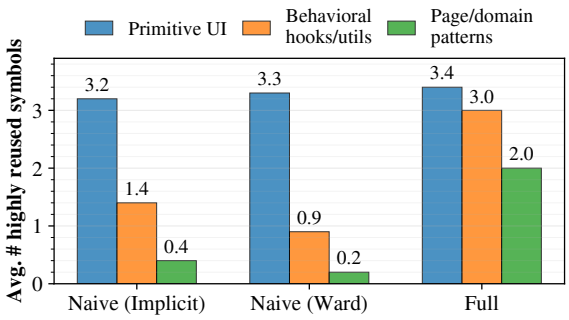
Figure 3: **LOC dynamics across sequential rounds.** Values are relative to each app's first round; shaded area shows ±1 std over three trials.

5.3 Library Utilization and Abstraction

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

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



(a) Average number of symbols by category.

Method	Primitive UI	Behavioral hooks/utils	Page/domain patterns
Naive (Implicit)	`Header` `Footer`	`useLocalStorage` `useFormValidation`	`ContactForm` `KpiCard`
Naive (Ward)	`Header` `Card`	`useLocalStorage` `createAuthContext`	`ContactForm`
Full	`NavigationBar` `Section`	`useRouter` `useFilteredList`	`FeatureCardGrid` `MessageBanner`

(b) Representative symbols.

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

larger reusable library rather than merely increasing reuse density over a small set of components. FULL exposes 13.3 components, 5-6 more than the naive variants on average. FULL also increases the number of broadly reused exports, with 6.0 exports used by 3–5 apps and 4.1 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 4 shows that all methods extract comparable numbers of primitive UI components, but FULL captures substantially more reusable symbols beyond primitives, including behavioral hooks/utilities and page/domain patterns. Table 4b provides representative examples, showing that the naive variants mostly reuse shallow widgets or thin utilities, whereas 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 G.

Method	No. of Apps 1–2	3–5	6–8	Total Exports
Naive (Implicit)	1.3	3.6	2.7	8.2
Naive (Ward)	1.4	3.3	2.0	7.2
Full	2.7	6.0	4.1	13.3

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

Metric	w/o library	w/ library App	w/ library Lib	Δ
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 isolate the library-as-prior effect with a controlled ablation: a vanilla coding agent runs with or without access to FULL's final SL. No extract/apply rounds, candidate retrieval, or detailed layout (8 Content-Presentation tasks $\times$ 3 trials).

Library access lifts UI-test accuracy $80.95 \rightarrow 84.35\%$ ($+3.40$ pp) and appearance $3.83 \rightarrow 3.92$. Total LOC and MDL stay flat: the library footprint offsets the drop in app-local code. Per-task before/after pairs in Figure 5.

6 Conclusion

We introduce the Super Library Agent problem: generating and maintaining application portfolios with a shared library. SLA addresses two failures of naive library construction, missed cross-application abstractions and incomplete migration, via summary extraction and call-graph apply. Across WebGen-Bench and PaperBench, SLA preserves functionality while improving maintainability over zero-shot and naive baselines, reducing duplication and verbosity and producing reusable abstractions. These results suggest that shared libraries can improve maintainability for portfolios of agent-generated applications.

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. In particular, our current setting only considers initial application implementation, without later issue patches, feature requests, or refactoring demands. 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 future changes, whereas our evaluation measures final generated codebases. Future work should complement static proxies with dynamic maintenance evaluations, such as follow-up issue-patch success, cross-application update consistency, regression frequency, and human judgments of abstraction quality.

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 synthesis for library learning. *Proceedings of the ACM on Programming Languages*, 7(POPL).

DeepSeek-AI. 2026. Deepseek-v4: Towards highly efficient million-token context intelligence.

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.

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.

Ziga Kovacic, Justin T. Chiu, Celine Lee, Wenting Zhao, and Kevin Ellis. 2025. Refactoring Codebases through Library Design. *arXiv preprint*. Version Number: 3.

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 Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Gabriel Orlanski, Devjeet Roy, Alexander Yun, Changho Shin, Alex Gu, Albert Ge, Dyah Adila, Nicholas Roberts, Frederic Sala, and Aws Albarghouthi. 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.

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.

Alex Thillen, Niels Mündler, Veselin Raychev, and Martin Vechev. 2026. CodeTaste: Can LLMs generate human-level code refactorings? *Preprint*, arXiv:2603.04177.

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).

Zhengren Wang, Rui Ling, Chufan Wang, Yongan Yu, Sizhe Wang, Zhiyu Li, Feiyu Xiong, and Wentao Zhang. 2025. 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*.

A Responsible NLP Checklist Details

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

A.1 Use or Creation of Scientific Artifacts

Artifacts we use. Our experiments rely on the following external artifacts: the **WebGen-Bench** benchmark (Lu et al., 2025), the **PaperBench** Code-Dev benchmark (Starace et al., 2025), the **WebVoyager** UI agent used as WebGen-Bench'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-5.4-mini`, `gpt-5.4-nano`, `Claude Opus 4.7`) for code generation and judging. License information is summarized in Table 6.

Artifact	License	Use
WebGen-Bench	Apache 2.0	eval suite
PaperBench	MIT	eval suite
WebVoyager	MIT	UI runner
mini-SWE-agent	MIT	harness
cocoindex-code	Apache 2.0	indexing
Qwen-2.5-7B	Qianwen	MDL ref.

Table 6: External artifacts and their licenses. All are used consistently with their stated research-use intent.

Consistency with intended use. All listed artifacts are explicitly released for 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 generated application codebases, the extracted Super Libraries at each round, the extraction traces, and the evaluation logs — are released under the MIT license for *research use only*. As they are derivatives of research-purpose benchmark inputs, they inherit a research-only condition and are not intended for production deployment or for any downstream task outside academic study of library-extracting coding agents.

A.2 AI Assistants in Research or Writing

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-5.4-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.

As research and engineering aids. During this project we used **ChatGPT (GPT-5 family)** and **Claude (Opus 4.7)** as auxiliary tools in a few bounded roles: (i) sanity-checking the logical flow and consistency of arguments in draft sections, (ii) stress-testing research ideas and surfacing counter-examples or related work we might have missed, and (iii) assisting with implementation-level code generation (e.g., boilerplate analysis scripts, plotting and table-generation 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.

B Supplementary Results

B.1 WebGen-Bench

B.2 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.

B.2.1 Evaluation Suites

Suite	WebGen-Bench tasks
S1 *Content Presentation Brochures & Portfolios*	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 *Data Management CRUD / ERP / CRM*	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 *User Interaction Social & Job Platforms*	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.

B.2.2 Per-Suite Results

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

B.3 PaperBench

B.3.1 Evaluation Suites

Suite	PaperBench tasks
S0	1. `fre`: Functional Reward Encodings. 2. `rice`: RICE. 3. `sapg`: Split and Aggregate Policy Gradients. 4. `ftrl`: RL Fine-tuning as Forgetting Mitigation.
S1	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.
S2	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.

Table 8: **PaperBench evaluation suites.** We construct three disjoint suites of four tasks each. Each suite is used as a sequential task stream for evaluating Super Library Agent methods.

B.3.2 Per-Suite Results

See Table 10 for PaperBench per-suite results.

C Algorithms

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

Metric	Suite	Zero	Naive		Full
			Implicit	Ward	
Functionality					
Acc. ↑	S1	82.99 ±2.12	83.67 ±0.00	83.67 ±3.68	83.33 ±2.12
	S2	70.28 ±2.55	70.83 ±3.00	70.00 ±4.64	72.22 ±3.37
	S3	75.44 ±0.89	76.36 ±1.57	74.54 ±3.96	76.06 ±2.62
	Avg.	76.24 ±5.80	76.96 ±5.83	76.07 ±7.00	**77.21 ±5.44**
Appear. ↑	S1	3.84 ±0.08	4.00 ±0.00	3.96 ±0.07	3.92 ±0.07
	S2	3.92 ±0.07	3.84 ±0.08	3.96 ±0.07	3.96 ±0.07
	S3	3.75 ±0.00	3.84 ±0.08	3.79 ±0.08	3.71 ±0.08
	Avg.	3.84 ±0.09	3.89 ±0.10	**3.90 ±0.10**	3.86 ±0.13
Maintainability					
LOC ↓	S1	7129 ±235	6809 ±83	6459 ±42	6299 ±245
	S2	9834 ±463	9890 ±741	9668 ±768	9689 ±576
	S3	11215 ±51	10700 ±734	10232 ±665	9669 ±445
	Avg.	9392 ±1819	9133 ±1853	8787 ±1834	**8552 ±1733**
MDL ↓	S1	31474 ±1691	30556 ±136	29927 ±614	30072 ±1193
	S2	36138 ±3020	37607 ±2778	38014 ±2299	36924 ±3397
	S3	37013 ±553	37144 ±125	36082 ±2663	35589 ±1884
	Avg.	34875 ±3118	35102 ±3689	34674 ±4070	**34195 ±3745**
Eros. ↓	S1	0.0113 ±0.0196	0.1476 ±0.0819	0.1406 ±0.0670	0.0102 ±0.0177
	S2	0.0817 ±0.0101	0.1151 ±0.0717	0.0613 ±0.0253	0.0673 ±0.0319
	S3	0.2167 ±0.0463	0.2075 ±0.0312	0.1730 ±0.0358	0.2185 ±0.0335
	Avg.	0.1032 ±0.0939	0.1567 ±0.0697	0.1250 ±0.0639	**0.0986 ±0.0964**
Verb. ↓	S1	0.0670 ±0.0189	0.0148 ±0.0056	0.0277 ±0.0192	0.0361 ±0.0251
	S2	0.2261 ±0.0336	0.1986 ±0.0610	0.1995 ±0.0166	0.1382 ±0.0212
	S3	0.1878 ±0.0234	0.2090 ±0.0335	0.1838 ±0.0142	0.1238 ±0.0127
	Avg.	0.1603 ±0.0754	0.1408 ±0.1008	0.1370 ±0.0836	**0.0994 ±0.0510**
Library Size					
LOC	S1	–	493 ±59	510 ±62	750 ±151
	S2	–	431 ±116	499 ±169	418 ±179
	S3	–	418 ±182	290 ±67	553 ±123
	Avg.	–	447 ±117	433 ±144	573 ±196
Tok.	S1	–	3529 ±399	3666 ±336	5281 ±1103
	S2	–	3123 ±753	3592 ±750	3070 ±1207
	S3	–	2867 ±1161	2012 ±607	4077 ±989
	Avg.	–	3173 ±776	3090 ±957	4143 ±1353
MDL	S1	–	1297 ±289	1445 ±155	1985 ±308
	S2	–	1193 ±299	1332 ±286	1214 ±383
	S3	–	1147 ±318	749 ±141	1693 ±305
	Avg.	–	1212 ±270	1175 ±369	1631 ±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 ± std over three independent trials per suite. Avg. is pooled across all 9 trials (3 suites × 3 trials).

Metric	Suite	Zero	Naive		Full
			Implicit	Ward	
Functionality					
Score ↑	S0	0.3409 ±0.0918	0.3393 ±0.0822	0.3918 ±0.0594	0.3690 ±0.0456
	S1	0.4394 ±0.0127	0.4885 ±0.0382	0.4557 ±0.0269	0.4584 ±0.0104
	S2	0.4461 ±0.0193	0.4006 ±0.0154	0.3958 ±0.0799	0.4160 ±0.0597
	Avg.	0.4088 ±0.0696	0.4095 ±0.0796	0.4144 ±0.0602	**0.4144 ±0.0542**
Maintainability					
LOC ↓	S0	7128 ±891	7155 ±582	7156 ±602	7080 ±526
	S1	6239 ±1006	6288 ±754	6381 ±820	6387 ±1109
	S2	6922 ±86	6309 ±518	5992 ±337	6198 ±381
	Avg.	6763 ±785	6584 ±691	**6510 ±742**	6555 ±758
Tok. ↓	S0	69723 ±6679	70292 ±4967	70942 ±5114	67897 ±5531
	S1	64270 ±10973	65198 ±6768	64823 ±9037	63029 ±12116
	S2	73351 ±2528	64868 ±6821	59797 ±4782	62997 ±5168
	Avg.	69115 ±7650	66786 ±6015	65187 ±7486	**64641 ±7549**
MDL ↓	S0	28483 ±2831	29080 ±2283	28671 ±2073	29006 ±3820
	S1	29510 ±4585	29216 ±1649	30483 ±4302	30444 ±7164
	S2	36079 ±3979	32919 ±2949	29731 ±2625	32518 ±624
	Avg.	31357 ±4894	30405 ±2778	**29628 ±2836**	30656 ±4349
Eros. ↓	S0	0.2070 ±0.0202	0.2536 ±0.0835	0.2585 ±0.0159	0.1552 ±0.0244
	S1	0.2342 ±0.0777	0.2394 ±0.0460	0.2281 ±0.0769	0.2678 ±0.0348
	S2	0.2560 ±0.0660	0.2656 ±0.0509	0.2709 ±0.0995	0.2616 ±0.0347
	Avg.	0.2324 ±0.0562	0.2529 ±0.0552	0.2525 ±0.0662	**0.2282 ±0.0613**
Verb. ↓	S0	0.8889 ±0.0042	0.8707 ±0.0216	0.8860 ±0.0135	0.8552 ±0.0064
	S1	0.8482 ±0.0231	0.8413 ±0.0197	0.8582 ±0.0102	0.8356 ±0.0154
	S2	0.8455 ±0.0102	0.8394 ±0.0217	0.8404 ±0.0278	0.8393 ±0.0221
	Avg.	0.8609 ±0.0246	0.8504 ±0.0237	0.8615 ±0.0257	**0.8434 ±0.0165**
Library Size					
LOC	S0	–	216 ±91	308 ±83	541 ±73
	S1	–	156 ±87	355 ±109	277 ±150
	S2	–	220 ±49	231 ±167	217 ±128
	Avg.	–	197 ±74	298 ±121	345 ±182
Tok.	S0	–	1960 ±821	2914 ±712	4872 ±754
	S1	–	1414 ±629	3132 ±963	2733 ±1786
	S2	–	2104 ±587	2089 ±1418	1879 ±1135
	Avg.	–	1826 ±673	2712 ±1043	3161 ±1745
MDL	S0	–	650 ±190	901 ±125	1583 ±263
	S1	–	511 ±218	1212 ±381	1043 ±441
	S2	–	854 ±131	825 ±561	786 ±379
	Avg.	–	672 ±218	979 ±388	1138 ±475

Table 10: **Per-suite results on PaperBench.** We report functionality and maintainability metrics across three disjoint suites of four tasks each, as mean ± std over three independent trials per suite. Avg. is pooled across all 9 trials (3 suites × 3 trials). Library Size reports the size of the final Super Library only.

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)

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

$L_0 \leftarrow \emptyset, \quad \mathcal{N}_0 \leftarrow \emptyset, \quad T_0 \leftarrow \emptyset$
for $r = 1$ **to** N/m **do**
  $\mathcal{B}_r \leftarrow \{x_{(r-1)m+1}, \ldots, x_{rm}\}$
  Stage 1. Coding agent — Generate codebases
    in parallel for all i **with** $x_i \in \mathcal{B}_r$:
      $c_i \leftarrow \text{CODING}(x_i \mid L_{r-1},\ \mathcal{N}_{r-1})$
    end parallel
    $\mathcal{N}_r \leftarrow \text{INDEXBLOCKS}\big(\mathcal{N}_{r-1},\ \{c_i\}_{x_i \in \mathcal{B}_r}\big)$
  Stage 2a. Two-level library extraction (consolidation then update)
    in parallel for all i **with** $x_i \in \mathcal{B}_r$:
      $c_i \leftarrow \text{CONSOLIDATE}(c_i,\ \mathcal{N}_r)$ ▷ intra-codebase pre-extraction consolidation
    end parallel
    $(L_r,\ T_r) \leftarrow \text{LIBRARYUPDATE}\big(\{c_t\}_{t=1}^{rm},\ L_{r-1},\ T_{r-1},\ \mathcal{N}_r\big)$
  Stage 2b. Dependency migration with extraction trace
    in parallel for all $j \in \{1, \ldots, rm\}$:
      $c_j \leftarrow \text{MIGRATE}(c_j,\ L_r,\ T_r)$ ▷ T_r pre-localizes migration sites in c_j
    end parallel
end for
return $\{c_j\}_{j=1}^{N},\ L_{N/m}$

D Prompts Used in SLA

This appendix lists the exact prompts used by both SLA variants. To keep the appendix compact while remaining reproducible, we adopt the following display policy.

Display policy. The pipeline core of SLA-full is benchmark-agnostic: the *Global Extract*, *Local Extract*, and *Apply* system prompts are equal across WebGen-Bench and PaperBench. We therefore reproduce those three system prompts *once* (§D.3.3, §D.3.4, §D.3.5). For the parts that are domain-specific — the *Naive single-shot library agent* and the *Coding agent* — we show the WebGen-Bench prompt verbatim and list the PaperBench delta as a short bullet list.

Cross-references to companion appendices. Section E reproduces the auxiliary *candidate-selection* LLM judge prompts used by Global Extract and Apply to assemble the `[Extract candidates]` / `[Apply candidates]` blocks referenced below.

D.1 Overview: SLA-naive vs. SLA-full

Stage	SLA-naive	SLA-full
Coding	✓	✓
Local Extract (consolidate)	—	✓
Global Extract	Library Agent	✓
Apply (migration)		✓

Table 11: Stage breakdown. SLA-full runs four separate sub-agents per round (§D.3): Coding, Local Extract (intra-codebase consolidation), Global Extract, and Apply (library-aware migration). SLA-naive omits Local Extract and collapses Global Extract + Apply into one single-shot Library Agent (§D.2).

D.2 SLA-naive — Single-Shot Library Agent

SLA-naive uses one sub-agent per round that walks every cumulative app, builds the shared library, and rewrites the apps in place during the same run. There is no candidate list (optional) and no `extract_map.md`.

System Prompt (WebGen-Bench).

System Prompt — WebGen Library Agent

```
You are a library manager. In one run you
build a shared library from N codebases
AND refactor those codebases in place to
import from it.

[Dual goals]
1. EXTRACT: identify cross-app primitives
(>=2 apps, behavior comparable)
   and write them into library.
2. APPLY: rewrite or refactor each codebase
to reduce its code size by utilizing library,
   delete the obsolete codes and local files.

Extraction without application is incomplete.
 Both are mandatory.

[Discovery]
Promote only patterns shared by **2+ tasks**
with comparable behavior
(API shape, side effects, observable outputs).


[Reject]
- Single-task helpers (used in only one task).

- Trivial <10 LOC patterns.
- Task-specific domain logic.
- Surface-similar code with diverging behavior
.

[Per-symbol procedure]
For each library symbol you adopt:
  1. Locate every local import / inline copy
of the baseline pattern in
     this task (e.g., `grep -rn`).
  2. Rewrite the call sites to import the
symbol from the library and
     use its API.
  3. Map argument / prop names where the
local API diverged.
  4. Delete the now-obsolete local definition
 file and any colocated
     resources (styles, fixtures, etc.) that
exist only to support it.
     No wrappers, no re-exports.

[Dead-code sweep] (after all symbol swaps)
Migration commonly leaves orphans behind.
Find and remove them:
- Definition files whose exported symbol was
just swapped --- delete.
- Import statements pointing at those deleted
 files --- remove.
- Helper modules referenced only by deleted
files --- delete (cascade).
- Colocated assets (styles, fixtures, JSON,
etc.) whose only consumer
  was a deleted module --- delete.
A final pass (`grep -rn`) over each swapped
symbol name AND each deleted
file path confirms zero stale references.

[Cheating prohibition]
No `WebFetch` / `WebSearch`. Do not read
sibling tasks outside `[Source
apps]`.

[Workflow]
1. Investigate every app under `[Source apps
]` --- if `[Library candidates]` is
   provided, use those clusters as the
starting point; otherwise explore freely.
2. Briefly plan (5-8 lines): which primitives,
 from which apps, at what
```

```
   abstraction, which local files will be
deleted.
3. Write the library at `<library_dir>`.
4. For each app: rewrite imports to relative
paths into
   `<library_dir>/src/` -> delete obsolete
files -> `grep -rn` to confirm
   no stale local imports remain.
```

User Prompt Template (WebGen-Bench).

User Prompt — WebGen Library Agent

```
Build a shared library from {N} WebGen-Bench
apps and refactor them
to use it --- in a single run.

[Library output]
Write at `{library_dir}`. Apps import via a
relative path from each
source file to `{library_dir}/src/index.js` (
barrel) or to a subpath
file under `{library_dir}/src/`. The library
is not an npm package ---
do not declare it in `package.json` and do
not add a `resolve.alias`
entry for it in `vite.config.js`.

[Source apps] (READ and EDIT --- in-place
refactor)
- `{task_id_1}` -> `{path_1}`
- `{task_id_2}` -> `{path_2}`
- ...

[App rules]
{APP_RULES_WEBGEN}     # see APP_RULES
listing below

[Existing library]
{existing_lib_summary}

{LIB_RULES_WEBGEN}     # see LIB_RULES
listing below

[Task descriptions]
### {task_id_1}
[Task]
{instruction_1}

### {task_id_2}
[Task]
{instruction_2}

...

[Library candidates]   # optional block ---
entirely omitted when no candidates
configured
{library_candidates}   # Ward-clustering or NL
-index, depending on configuration
```

D.2.1 PaperBench Delta

The PaperBench naive library agent uses a system prompt and user prompt template that are structurally identical to the WebGen-Bench versions. Concrete differences:

- **System prompt**: workflow step 2 substitutes "from which apps" → "from which papers". Workflow step 4 substitutes "rewrite imports to relative paths into `<library_dir>/src/`" → "rewrite imports to `from lib.<subpkg> import <Symbol>`". All other clauses ([Dual goals], [Discovery], [Reject], [Per-symbol procedure], [Dead-code sweep], [Cheating prohibition]) are byte-equal.
- **User prompt template**: "WebGen-Bench apps" → "paperbench submissions"; `[Source apps]` → `[Source submissions]`; `[Task descriptions]` → `[Paper descriptions]` where each block lists the paper's source materials (`paper.md`, `addendum.md`, `blacklist.txt`, `assets/`) rather than a self-contained instruction string.
- **Library output paragraph**: replaced with a Python-package pointer ("Submissions import as `from lib.<subpkg> import <Symbol>` — `PYTHONPATH` is configured by the runner"). The WebGen-specific Vite / npm caveat is dropped.
- **`APP_RULES` / `LIB_RULES`**: replaced with the PaperBench equivalents (Python 3.11+ package tree, no GPU assumption, plain Python `lib/<subpkg>/<module>.py` layout instead of `src/components/<Name>.jsx`).

D.3 SLA-full — Four-Stage Pipeline

Each round of SLA-full runs four sub-agents in sequence: Coding (§D.3.1), Local Extract (§D.3.4), Global Extract (§D.3.3), and Apply (§D.3.5). Local Extract, Global Extract, and Apply share their *system* prompts byte-equal across WebGen-Bench and PaperBench; only the user-prompt scaffolding (workspace block, app rules) differs.

D.3.1 Coding Agent (WebGen-Bench)

System Prompt.

System Prompt — WebGen Coding Agent

```
You are a helpful coding agent generating a
code application from a user instruction.

[Scope discipline]
```

```
Implement exactly what the [Task] instruction
 asks for --- nothing more.
Do not add more complex codes (pages, routes,
 components, or features) the user did not
request, even when they would be "nice to
have" or would let you
showcase available library symbols.

[Cheating prohibition]
- Do NOT read any other task's submission
directory (sibling tasks in the
  same `tasks/` parent). You may only read
inside your own workspace.
- Do NOT browse the web (`WebFetch`, `
WebSearch`) for the task. You may rely
  on prior knowledge.
```

User Prompt Template.

User Prompt — WebGen Coding Agent

```
You are generating a React + Vite web
application.

[Task]
{instruction}

{APP_RULES_WEBGEN}     # see APP_RULES
listing below

[Workspace]
- Your submission target is `{workspace_dir}`.
 Every file you write MUST live
  under that directory. Files outside it are
NOT preserved when this task ends.

{library_block}        # empty in baseline;
LIBRARY_BLOCK in sla_naive / sla_ours
```

D.3.2 Coding Agent — PaperBench Delta

For PaperBench we use the upstream PaperBench reference solver's coding prompts *near-verbatim* so behavior stays directly comparable to the original PaperBench evaluation (specifically the non-iterative `code_only` branch of the upstream system / user prompt). The only changes are:

- **Path substitutions**: the hardcoded upstream container paths are templated as `{paper_dir}`, `{workspace_dir}`, `{agent_env_path}`.
- **Additional-notes trim**: dropped the upstream "root access" bullet, hardcoded "no GPU" instead of probing, and surfaced the time limit in hours. Added two safety bullets ("Per-bash timeout (10 minutes)" and "Grading is code-only") that document runtime invariants not present in the upstream template.
- **Library slot**: an optional `{library_block}` slot is appended to the `RESOURCES` section — non-empty in SLA-naive and SLA-full rounds where a pre-built shared library is mounted, empty in the baseline.
- **`APP_RULES` block**: the PaperBench `APP_RULES` (Python 3.11+, no monorepo, no vendoring of large dependencies) is injected between `RESOURCES` and `SUBMISSION`. The same block is also injected by the Apply / Local Extract agents so the coding submission obeys the same rules its downstream refactor phases will enforce.

D.3.3 Global Extract Agent — Cross-Bench (byte-equal)

The *Global Extract* system prompt and its companion `extract_map.md` writing instruction are byte-equal across WebGen-Bench and PaperBench. The bench-specific user-prompt scaffolding substitutes `LIB_RULES` and the source-app listing.

System Prompt.

System Prompt — Global Extract (Cross-Bench)

```
You are a library extractor.
Given N task submissions and a (possibly
seeded) library, identify
**cross-task** code blocks and add/update
them in the library. Task
sources are READ-ONLY in this phase --- do
not edit any task source.

[Discovery]
Promote only patterns shared by **2+ tasks**
with comparable behavior
(API shape, side effects, observable outputs).


[Reject]
- Single-task helpers (used in only one task).

- Trivial <10 LOC patterns.
- Task-specific domain logic.
- Surface-similar code with diverging behavior
.
```

Second-Turn Instruction (`extract_map.md` writing protocol). Delivered as a 2nd resume turn so the 1st-turn prompt stays focused on extraction.

Second-Turn Instruction — extract_map.md (Cross-Bench)

```
Update `{library_dir}/extract_map.md`
cumulatively.

If the file already exists (from a prior
round), `cat` it FIRST and keep
all prior sections verbatim. Then add or
revise sections for symbols
touched in THIS run, separating each with
`---`. Do not silently drop
prior sections.

Each section format:

## library symbol: `<exported name>`

**Module**: `<path/from/library/root>`

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

| Task | File   | Symbol |
|------|--------|--------|
| <id> | <path> | <name> |
| <id> | <path> | <name> |
(File = path under submission/; Symbol =
original name)

**Why generalized**
2-3 sentences. The common pattern, and what
each source shared.

**Apply guidance**
1-2 sentences. Which baseline pattern Apply
replaces, and how (import
path, API mapping).

---

**STRICT rules** (Apply trusts this verbatim):

1. Verify every row. No vague placeholders
(`(implicit)`, `(inline)`, etc.).
2. Use literal task_ids.
3. Drop a symbol entirely if fewer than 2
verified rows remain.
4. Cumulative: include sections for symbols
added in earlier rounds when
   their entries change (otherwise the
previous round's section stays).

If you added zero new cross-task-anchored
symbols, write
`(no new cross-task symbols this round)` at
the top and keep the previous
round's sections intact.
```

User Prompt Template (WebGen-Bench).

User Prompt — Global Extract (WebGen)

```
Extract a cross-task shared library from {N}
tasks.

[Source tasks] (READ-ONLY)
- `{task_id_1}` -> `{path_1}`
- `{task_id_2}` -> `{path_2}`
- ...

[Task descriptions]
### {task_id_1}
[Task]
{instruction_1}

...

[Library output]
Target directory: {library_dir}

[Existing library]
{existing_lib_summary}

{LIB_RULES_WEBGEN}     # see LIB_RULES
listing below

[Extract candidates]
{nl_extract_candidates}    # see Appendix C.1
```

PaperBench delta: `[Source tasks]` → `[Source submissions]`, `[Task descriptions]` → `[Paper descriptions]`; `LIB_RULES_WEBGEN` → `LIB_RULES_PAPERBENCH` (Python package layout).

D.3.4 Local Extract Agent — Cross-Bench (byte-equal)

The *Local Extract* system prompt is byte-equal across WebGen-Bench and PaperBench. User-prompt scaffolding differs only in `APP_RULES` and the workspace path.

System Prompt.

System Prompt — Local Extract (Cross-Bench)

```
You factor out intra-codebase duplication
inside one task to make the code minimal.
Goal: move helpers / utilities / reusable
units that the app uses in 2+ places into the
 app's existing source layout, and rewrite
call sites to import them.

[Discovery --- start from candidates, verify
with bash]
- `[Local extract candidates]` in the user
prompt names patterns that recur
  inside this single app. They are starting
points, not authority.
- Verify each one with `grep -rn` / `cat`.
Keep only patterns with 2+ real
  call sites inside this app.
- A pre-existing global library may be listed
 in `[Existing global library]`.
  If a candidate is already covered there, do
 NOT recreate it locally ---
  rewrite the call sites to import from the
global library.
```

```
[Skip rules]
- One-off helpers (single call site) stay
where they are.
- Trivial <10 LOC patterns stay inline.
- Apparent duplicates whose behavior actually
 diverges --- keep separate.

[Placement --- let the existing layout decide
]
- Read the current codebase tree first.
Notice the conventions the app
  already uses (where components live, whether
 there's a hooks file,
  how utilities are organized, etc.).
- Add each extracted helper where it most
naturally fits given that
  layout. If a thematically related file
already exists, append to it
  rather than create a new one.
- Trust the existing structure first. If a
helper truly has no
  natural home, create whatever fits at the
level of its callers.
- The name `lib/` is reserved for the cross-
task shared library
  (mounted separately and referenced by
global imports). Do NOT
  create a `src/lib/` (or `src/local_lib/`)
for app-local helpers.

[Verify]
After editing, `grep -rn` to confirm no stale
 local definitions of moved
symbols remain. The task's build command (see
 [Task] / [Implementation
rules]) should still succeed.
```

User Prompt Template (WebGen-Bench).

User Prompt — Local Extract (WebGen)

```
You are deduplicating intra-app code in
WebGen-Bench task `{task_id}`.

[Task]
{instruction}

{APP_RULES_WEBGEN}     # see APP_RULES
listing below

[Workspace] (READ and EDIT in place)
{workspace_dir}

[Output --- adapt to the existing source
layout]
Inspect codebase first and integrate each
extracted helper into the
app's existing structure. Guidance:

  - Reuse the layout the app already has
where it makes sense.
    If a thematically related file already
exists, APPEND to it
    rather than create a new file.
  - If the existing layout doesn't have a
natural home for a given
    helper, create whatever directory or file
 feels appropriate at
    the level of its callers.
  - `lib/` is reserved for the cross-task
shared library. Do NOT
    create `src/lib/` (or `src/local_lib/`)
for app-local helpers.
  - Rewrite call sites with relative imports
against whatever
    paths you actually chose. No barrel files
 unless the app
    already uses one.

[Existing global library] (read-only --- do
not recreate these locally)
{existing_global_lib_summary}

[Local extract candidates]
{nl_local_extract_candidates}
```

PaperBench delta: APP_RULES_WEBGEN → APP_RULES_PAPERBENCH; the [Output] block remains byte-equal (the "existing-layout-first" philosophy applies to Python package trees identically).

D.3.5 Apply Agent — Cross-Bench (byte-equal)

The *Apply* system prompt and user-prompt template are byte-equal across WebGen-Bench and PaperBench. Bench-specific scaffolding fills the {app_rules}, {library_block}, and {workspace_block} slots.

System Prompt.

System Prompt — Apply (Cross-Bench)

```
You refactor a task's code to use a pre-built
 shared library.
Your goal is to minimize the task's total
amount of code by replacing
local implementations with library
equivalents. An import without the
corresponding local-code removal is
incomplete work.

[Apply candidates]
- `extract_map.md`: Each section names a
symbol with its "Apply guidance"
  line --- those are the swap targets to
action first.
- `[Apply candidates]`: NL-index suggestions.
 Verify by reading both lib
  source and the task file.

[Per-symbol procedure]
For each library symbol you adopt:
  1. Locate every local import / inline copy
of the baseline pattern in
     this task.
```

```
  2. Rewrite the call sites to import the
symbol from the library and
     use its API.
  3. Map argument / prop names where the
local API diverged.
  4. Delete the now-obsolete local definition
 file and any colocated
     resources (styles, fixtures, etc.) that
exist only to support it.
     No wrappers, no re-exports.

[Dead-code sweep] (after all symbol swaps)
Migration commonly leaves orphans behind.
Find and remove them:
- Definition files whose exported symbol was
just swapped --- delete.
- Import statements pointing at those deleted
 files --- remove.
- Helper modules referenced only by deleted
files --- delete (cascade).
- Colocated assets (styles, fixtures, JSON,
etc.) whose only consumer
  was a deleted module --- delete.
A final pass (`grep -rn`) over each swapped
symbol name AND each deleted
file path confirms zero stale references.

[Cheating prohibition]
No `WebFetch` / `WebSearch`. Do not read
sibling tasks outside `[Source
apps]`.

[Verify]
- No stale local imports of replaced symbols
remain (`grep -rn`).
```

User Prompt Template.

User Prompt — Apply (Cross-Bench)

```
You are refactoring task `{task_id}` to use
library.

[Task]
{instruction}

{app_rules}            # WebGen or PaperBench
 APP_RULES

[Workspace] (READ and EDIT in place)
{workspace_dir}

[Library] (read-only)
{library_dir}
{library_block}        # WebGen or PaperBench
 LIBRARY_BLOCK

[Extract map]
{extract_map_block}    # full text of <
library_dir>/extract_map.md

[Apply candidates]
{nl_apply_candidates}  # see Appendix C.2

[Workspace]
- Your submission target is `{workspace_dir}`.
 Every file you write MUST live
  under that directory. Files outside it are
NOT preserved when this task ends.
```

D.4 Bench-Level Rule Blocks

The agents above splice in APP_RULES, LIB_RULES, and LIBRARY_BLOCK from the bench-level configuration. We reproduce the WebGen-Bench versions once below; PaperBench versions differ only as noted in the deltas above (Python package layout, lib/<subpkg> import path, no Vite/CSS scaffolding).

WebGen APP_RULES.

APP_RULES — WebGen-Bench

```
[Stack]
- React 18 + Vite 4 + plain CSS. No Tailwind,
 no UI kit.
- All state lives in-memory or in `
localStorage`. No backend, no `fetch()`
  unless the instruction explicitly requires
it.

[Error UI]
- Do NOT use `alert()` for validation or auth
 errors. Render messages
  as in-page elements. The browser modal
dialog blocks the UI-test
  runner (Selenium aborts on unexpected alert)
.

[Auth seed]
If the app has login/registration, seed at
least one demo account that
accepts username `admin`, email `
admin@example.com`, and password
`admin123456`. Any of the three identifier
strings (username OR email)
must work depending on which the login form
requests. These are the
exact credentials the UI-test runner enters.

[Build]
Before ending the turn, the production build
at `<workspace_dir>` MUST
exit 0. Actually run it --- reading files and
 declaring "looks correct"
does NOT satisfy this. Fix any error and re-
run before ending.

If you swapped symbols or removed files
during this turn, also clean up
the resulting stale references and dead code
(orphaned imports,
abandoned helper files, unused colocated
assets) before the final
build. Skip this clean-up step when nothing
was swapped or removed.

[Required files at submission root]
```

```
  package.json   (name, type=module, scripts:
 dev/build/preview,
                  deps: react ^18 + react-dom
 ^18,
                  devDeps: @vitejs/plugin-
react ^4 + vite ^4)
  vite.config.js (defineConfig + react plugin;
 template below)
  index.html     (single <div id="root"></div
>)
  src/main.jsx   (ReactDOM.createRoot -> <App
 />)
  src/App.jsx    (top-level component)

[Styling discipline]
- Each app-owned component co-locates its `.
css` next to the `.jsx`
  file and imports it from the top of the JSX
 (`import './Foo.css'`).
- Do NOT define styles for class names that
belong to a library
  component's internals. Library-internal
class names belong in the
  library's own CSS file.
- Override library styling only through
documented props
  (`style` / `className` / `variant`).

[Library import discipline]
- Apps import library symbols via a RELATIVE
PATH to the library's
  `src/index.js` barrel (named imports only).
 Never via an alias, a
  package specifier, or a deep path that
bypasses the barrel.

[vite.config.js template] --- rewrite the
existing file if it diverges
(no `resolve.alias`; the shared library is
imported via relative paths,
not aliases or npm workspaces)

```
import { defineConfig } from 'vite';
import react from '@vitejs/plugin-react';

export default defineConfig({
  plugins: [react()],
});
```
```

WebGen LIB_RULES.

LIB_RULES — WebGen-Bench

```
[Directory layout] --- plain JS/JSX directory
 at `<library_dir>`:
  src/index.js                      barrel re-
export of every public symbol
  src/components/<Name>.jsx         + co-
located <Name>.css
  src/hooks/<useX>.js
  src/utils/<x>.js
  src/services/<x>.js
JSX in `.jsx` only. Plain CSS only --- no
Tailwind / Bootstrap wrappers.
The library is NOT an npm package --- no `
package.json`, no install step.

[Module conventions]
- Components: `export default function Foo
(...)`.
- Hooks / utils / services: named exports.

[Barrel exports --- `src/index.js`]
Re-export every public symbol as a NAMED
export. The barrel is the
ONLY supported entry point --- apps must not
import library internals
via subpaths. Example:

```
export { default as LoginForm }     from './
components/LoginForm.jsx';
export { default as SearchBar }     from './
components/SearchBar.jsx';
export { useFormValidation }        from './
hooks/useFormValidation.js';
export { loadStorage, saveStorage } from './
utils/localStorage.js';
```

[Self-contained styling --- MANDATORY]
Every component MUST own its styles.
Concretely:
- For each `components/<Name>.jsx`, ship a
sibling `components/<Name>.css`
  containing every class name the component
renders (including `:hover`,
  `:focus`, state classes, and `@media`).
- The component MUST `import './<Name>.css'`
at the top of the JSX
  file. Do not rely on the importing app to
define any of the
  component's class names.

[Style override interface]
Components SHOULD expose at minimum:
- `className?: string` --- appended to the
root element so the caller can
  scope overrides.
- `style?: React.CSSProperties` --- merged
into the root element's inline
  style.
For multi-part components, prefer a small `
variant` enum prop over
leaking internals.

[Reuse discipline]
Do not duplicate symbols already present in
the seeded library. Before
adding a new symbol, scan the existing barrel
 for an equivalent --- extend
or parameterize the existing one if possible.
```

WebGen LIBRARY_BLOCK.

LIBRARY_BLOCK — WebGen-Bench

```
[LIBRARY]
A useful library is available at `{
library_dir}`.
```

```
This library consists of several useful
modules that recur frequently when writing
app code.
Use this library to keep your app code
minimal.

**Library usage rules:**
  (1) Use library if possible: Do not re-
implement code that already exists there.
  (2) Library Investigation: Before using a
library symbol, read its
      source under `{library_dir}` to learn
its parameters and behavior.
  (3) Use the library solely for code reuse:
Do not imitate its
      internal patterns. Your code structure
should follow the task's
      requirements, not the library.
  (4) If a building block is not in the
library, implement it yourself.
  (5) Do not modify the library (read-only).
  (6) Import via relative path from the
importing file to
      `{library_dir}/src/index.js` (barrel)
or to a subpath file. The
      library is not an npm package --- do
not add it to `package.json`,
      and do not add a `resolve.alias` entry
for it in `vite.config.js`.
```

E Prompts Used for Candidate Selection (SLA-Ours)

E.1 Library Extraction Candidates Selection

E.1.1 System Prompt

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
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.
```

E.1.2 User Prompt

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.
```

E.1.3 Assistant Response

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/JobsScreen/
JobsScreen.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 apps 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)
```

E.2 Library Application Candidates Selection

E.2.1 System Prompt

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'.
```

E.2.2 Assistant Response

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 hooks 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 hooks derived
categories + filtered list).

(skip)
```

F Extraction Trace

After each global extraction phase, agent writes an markdown format document that records the provenance of every new or updated library symbol: which task files contributed the original implementation, why the pattern was deemed generalizable, and how downstream apps should migrate to it. The file is cumulative with later rounds appending newly extracted symbols and revising existing entries whose lib-side implementation changed.

The entry below is taken verbatim from the round-1 extract trace produced by the Full variant on Suite 1 (trial 1), in which the agent generalized two independent `localStorage` usage patterns from tasks 000019 and 000075 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.
```

Each extract_map entry follows the same four-part schema: **library symbol** (the new export name and its module path), **Sources** (a table keyed by task ID, file, and the original inline location), **Why generalized** (the agent's justification for cross-app consolidation), and **Apply guidance** (concrete migration instructions for the dependency- migration phase).

G Library Export Listings for Abstraction Quality

We report the complete Super Library export inventory underlying the abstraction-quality analysis in Figure 4. Each table corresponds to one WebGen-Bench 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.

Method	Trial	#	Exported symbols with reuse rate
NAIVE	T1	6	Footer (100%), Header (88%), ContactForm (62%), PillNav (50%), useLocalStorage (50%), SkillBar (25%).
NAIVE	T2	5	Footer (100%), Header (88%), ContactForm (62%), useLocalStorage (62%), SkillBar (25%).
NAIVE	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 12: 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	T1	10	Button (100%), FormField (88%), Card (75%), useLocalStorage (75%), ErrorMsg (62%), NavBar (50%), StatusPill (50%), SearchInput (38%), loadStorage (0%), saveStorage (0%).
NAIVE	T2	10	useLocalStorage (100%), Button (88%), Card (88%), FormField (75%), KpiCard (75%), Footer (50%), NavBar (50%), SearchBar (38%), PageHeader (12%), DataTable (0%).
NAIVE	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 13: 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.

Method	Trial	#	Exported symbols with reuse rate
NAIVE	T1	8	Alert (100%), loadStorage (100%), saveStorage (100%), Footer (62%), FileInput (38%), SearchBar (38%), useFileUpload (25%), useLocalStorage (0%).
NAIVE	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	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 (50%), useImageUpload (38%).

Table 14: 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.

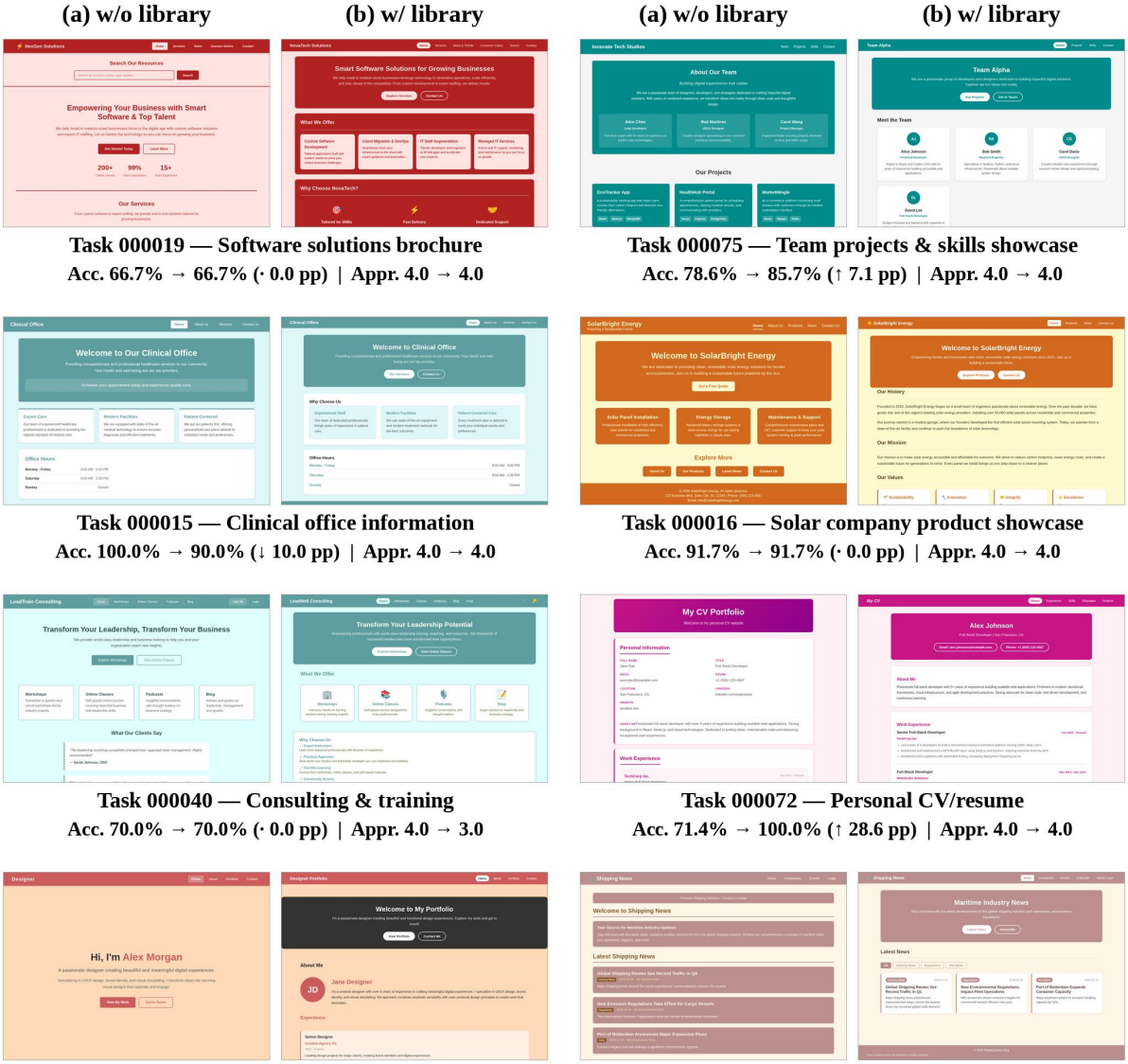


Figure 5: **All eight Content-Presentation tasks under trial 1.** Each task is shown as a (without library, with library) pair; the eight tasks are tiled as four rows of two tasks. UI-test accuracy and appearance grade are reported below each pair.