DEEPCODE: OPEN AGENTIC CODING

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent advances in Large Language Models (LLMs) have enabled the shift from coding assistants to autonomous software engineers. However, high-fidelity document-to-codebase synthesis—such as reproducing scientific papers—remains challenging due to the fundamental conflict between information overload and the finite context constraints of LLMs. In this work, we introduce DeepCode, a fully autonomous framework that addresses this challenge through principled information-flow management. DeepCode maximizes task-relevant signals under strict context budgets via four orchestrated operations: source compression via blueprint distillation, structured indexing using stateful memory, conditional knowledge injection via retrieval-augmented generation, and closed-loop error correction. Extensive evaluations on PaperBench demonstrate that DeepCode achieves state-of-the-art performance among the evaluated automated agents, outperforms leading commercial agents on a 5-paper subset, and exceeds the reported PhD-level human Best@3 baseline on the PaperBench 3-paper subset under the Replication Score metric. Our source code is available at: https://anonymous.4open.science/r/DeepCode-C464.

1 INTRODUCTION

Large Language Models (LLMs) are changing how software is specified, implemented, and maintained (Jiang et al., 2024; Ge et al., 2025). AI-assisted coding tools such as Cursor and Codex automate routine implementation and offer inline suggestions (Peng et al., 2023; Dong et al., 2025), while recent agentic frameworks aim to plan, implement, and refine entire projects from high-level specifications (Wang et al., 2025a; Tang et al., 2025). This shift toward *agentic software engineering* raises a central question: *can a coding agent translate rich, informal specifications into complete, robust software systems?*

A stringent testbed is *high-fidelity, document-grounded program synthesis*: reproducing a scientific paper as a complete code repository. Papers combine prose, equations, pseudocode, and scattered hyperparameters into a multimodal specification. PaperBench evaluates this task on 20 ICML papers, reporting a 42.4% replication score for o1 with IterativeAgent and 72.4% for human experts on the human-evaluation subset (Starace et al., 2025). PaperCoder uses a planning, analysis, and generation pipeline, achieving 51.14% on PaperBench (Seo et al., 2026). These results motivate four challenges:

(i) Specification Preservation. Papers describe the target system through scattered, multimodal constraints. Preserving a faithful mapping from this fragmented specification to implementation is inherently difficult. **(ii) Global Consistency under Partial Views.** Repositories comprise interdependent modules, but generation proceeds file-by-file under limited context. Maintaining consistency across interfaces, types, and invariants under finite context windows easily leads to broken abstractions. **(iii) Completion of Underspecified Designs.** Papers specify only algorithmic cores, leaving implementation details and experimental frameworks implicit. Inferring these consequential but underspecified choices is non-trivial. **(iv) Executable Faithfulness.** Faithful reproduction requires executable systems, not just plausible code. Long-horizon generation often yields repositories with subtle logic bugs, dependency conflicts, and fragile pipelines that prevent end-to-end execution.

We argue that addressing these challenges requires *principled information-flow management*. We view document-to-repository synthesis as a process of preserving, routing, and exposing task-relevant information under finite context budgets. Naively concatenating the full paper with growing code history can saturate the working context with redundant tokens, making critical algorithmic constraints

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

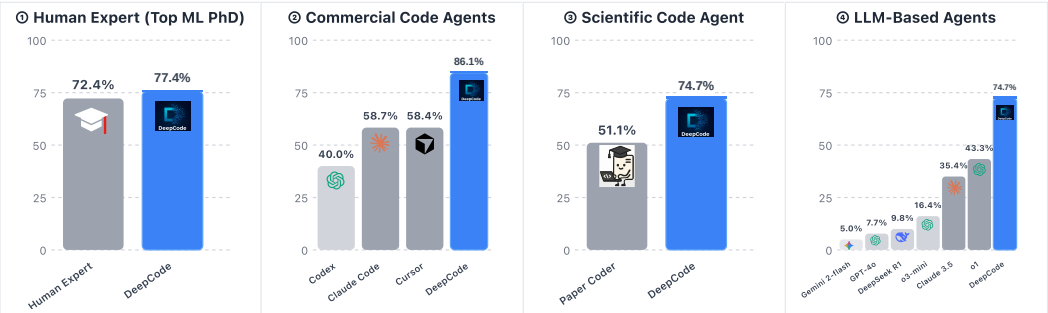


Figure 1: DeepCode achieves superior performance on PaperBench, exceeding the reported PhD-level human Best@3 baseline on the 3-paper subset and outperforming leading commercial agents such as Cursor and Claude Code on the 5-paper subset.

harder to retrieve and maintain. Consequently, reliable repository generation calls for *contextual information maximization*: each step should maintain a compact, structured, and task-relevant context while filtering out irrelevant noise.

We introduce **DeepCode**, an open agentic coding framework organized around four information operations: (1) *source compression* distills multimodal specifications into an implementation blueprint; (2) *structured indexing* summarizes the evolving repository to preserve cross-file consistency; (3) *conditional knowledge injection* retrieves implementation patterns to fill specification gaps; and (4) *error correction* uses execution feedback to guide iterative repair. Our contributions are threefold:

- We characterize document-to-repository synthesis from an information-flow perspective, identifying the key challenge as preserving task-relevant signals from fragmented specifications under finite context budgets. This motivates a design principle for agentic coding: explicitly structure, route, and compress information during long-horizon generation.
- We instantiate this principle in DeepCode, an autonomous framework that coordinates blueprint distillation, stateful memory, conditional knowledge injection, and closed-loop verification. Together, these operations maintain a compact working context and mitigate failures in specification preservation, cross-file consistency, and implicit knowledge completion.
- Extensive evaluations on PaperBench demonstrate that DeepCode achieves state-of-the-art performance among the evaluated automated agents, decisively **outperforming leading commercial agents** (*e.g.*, Cursor, Claude Code) on the evaluated subset and, notably, **surpassing the reported PhD-level human expert baseline** on the PaperBench 3-paper subset under the Replication Score metric. Furthermore, our results validate that hierarchical information orchestration is a more effective scaling path than simply increasing context window size.

2 PRELIMINARIES

The primary objective of this work is to develop a system for *high-fidelity program synthesis*. We formalize this as the task of learning a mapping $\mathcal{F}_{gen} : \mathcal{D} \to \mathcal{P}$, which translates a multimodal specification document $\mathcal{D}$ into a comprehensive, executable code repository $\mathcal{P}$. This formulation encompasses both scientific reproduction (where $\mathcal{D}$ is a research paper) and general software engineering (where $\mathcal{D}$ is a requirements document).

Input Specification. The source $\mathcal{D}$ is modeled as a sequence of multimodal segments $\mathcal{D} = (d_1, \ldots, d_L)$, comprising text, equations, pseudocode, and figures. The length L typically creates a context-saturation challenge for standard models.

Target Repository. We define the output repository as a structural tuple $\mathcal{P} = (\mathcal{H}, \mathcal{C})$:

- $\mathcal{H}$ represents the directory hierarchy, acting as the project's skeleton by defining the logical organization and file paths.

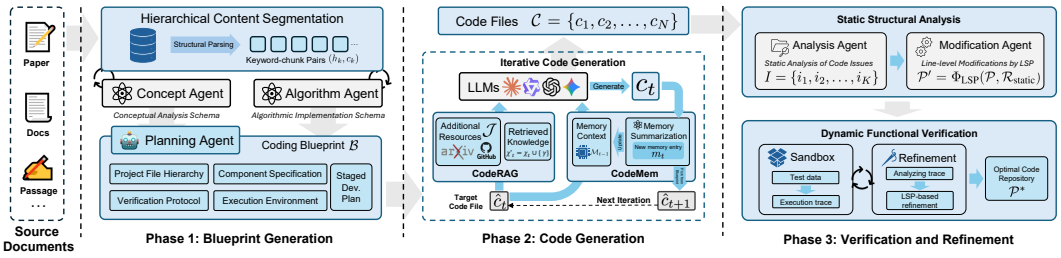


Figure 2: The overall framework of DeepCode.

- $\mathcal{C} = \{c_1, \ldots, c_N\}$ denotes the set of file contents, encompassing both executable source logic (*e.g.* `.py`, `.js`) and environment configurations (*e.g.* `requirements.txt`, `README.md`).

3 The DeepCode Framework

We introduce DeepCode, a framework for repository-level synthesis through principled information-flow management. To maintain a compact and task-relevant working context under finite context windows, DeepCode orchestrates three phases: (1) **Blueprint Generation** distills the complex document $\mathcal{D}$ into a structured blueprint, filtering redundant exposition while preserving key algorithmic specifications; (2) **Code Generation** maintains cross-file consistency through stateful *Code Memory* and bridges implicit implementation gaps via *CodeRAG*; and (3) **Automated Verification** incorporates execution feedback into an iterative repair loop, correcting implementation errors and improving functional correctness. The overall framework of DeepCode is illustrated in Fig 2.

3.1 Phase 1: Blueprint Generation

The primary goal of Phase 1 is source compression: distilling the unstructured, lengthy content of a source document $\mathcal{D}$ into a structured, machine-readable implementation blueprint. This process mitigates information overload by converting raw inputs into a compact blueprint that highlights implementation-relevant details.

3.1.1 Hierarchical Content Segmentation

To achieve this without context saturation, we introduce a **hierarchical content index** as a preprocessing step. Instead of processing the entire document simultaneously, we parse $\mathcal{D}$ based on explicit structural delimiters (e.g., section titles) to create distinct content chunks accessible via semantic keywords. This indexing mechanism transforms long-context comprehension into a series of manageable, on-demand retrievals, allowing subsequent agents to fetch only task-relevant segments. Detailed formal definitions of the parsing logic and chunking structure are provided in A.2.1.

3.1.2 Multi-Agent Specification Analysis

Following segmentation, we employ a dual-track multi-agent system to decompose the comprehension task, ensuring both architectural coherence and mathematical precision without overload.

Specifically, two specialized agents process the indexed document in parallel. First, the **Concept Agent** constructs a high-level *Conceptual Analysis Schema*, mapping the paper's core contributions, system architecture, and reproduction goals using broad semantic queries (*e.g.* methodology, architecture). Simultaneously, the **Algorithm Agent** executes a fine-grained search using specific technical keywords (*e.g.* hyperparameter, equation) to populate an *Algorithmic Implementation Schema*. This schema captures granular details, including verbatim pseudocode, mathematical formulations, and network specifications. This separation of concerns allows the system to parse abstract design and concrete implementation details independently.

3.1.3 IMPLEMENTATION BLUEPRINT SYNTHESIS

The **Code Planning Agent** unifies the outputs from the Concept and Algorithm agents into a single, holistic implementation blueprint ($\mathcal{B}$). This agent acts as a structural architect, grounding abstract components from the conceptual schema in the precise technical specifications from the algorithmic schema. It resolves ambiguities via targeted re-querying of the document index.

The resulting Blueprint $\mathcal{B}$ serves as the definitive "source of truth" for generation, replacing the raw document. It is organized into five canonical sections: (1) *Project File Hierarchy*, dictating the logical repository structure; (2) *Component Specifications*, mapping code modules to specific algorithms; (3) *Verification Protocols*, defining success metrics; (4) *Execution Environment*, specifying dependencies; and (5) *Staged Development Plan*, outlining the implementation order. This structured representation maximizes signal density, effectively bypassing the long-context bottleneck.

3.2 PHASE 2: CODE GENERATION

Following the structured blueprint, Phase 2 synthesizes the repository. The key challenge is maintaining cross-file consistency under finite context budgets, since naively concatenating generated code history introduces substantial redundant context and obscures relevant dependencies. To address this, we propose a dual-mechanism strategy for efficient information routing: (1) **CodeMem**, a stateful memory module that preserves internal structure via semantic compression and structured indexing, and (2) **CodeRAG**, which performs conditional knowledge injection by grounding implementations in external patterns to bridge implicit knowledge gaps.

3.2.1 STATEFUL GENERATION WITH CODEMEM

The Code Memory mechanism decouples repository growth from context window limitations by maintaining a compressed, structured memory bank $\mathcal{M}$ that evolves dynamically. Instead of passing full historical source code to the generative agent, the system incrementally builds and queries $\mathcal{M}$ to keep cross-file consistency.

Iterative Generation Process. Let $\mathcal{C} = \{c_1, \ldots, c_N\}$ denote the set of generated code files, each corresponding to a blank target file $\hat{c}_t$. The generation proceeds as a sequential state transition. At step t, the agent constructs a local context $\mathcal{X}_t$ to generate c_t from $\hat{c}_t$, using the global blueprint $\mathcal{B}$ and a selectively retrieved subset of memory entries $\mathcal{M}_{t-1}$ from previously implemented files:

$$\mathcal{X}_t = (\mathcal{B}, \text{SelectMemory}(\mathcal{M}_{t-1}, \mathcal{B}, \hat{c}_t)) \tag{1}$$

Here, SelectMemory represents an LLM-driven retrieval process where the agent examines the blueprint and the target file $\hat{c}_t$ to identify relevant dependencies from $\mathcal{M}_{t-1}$, retrieving only pertinent implementation summaries. The Coding Agent $\mathcal{L}$ then synthesizes the code: $c_t = \mathcal{L}(\mathcal{X}_t)$.

Semantic Compression and Memory Update. Crucially, once c_t is generated, it is not stored in the active context. Instead, a specialized Summarization Agent $\mathcal{S}$ distills the raw code into a compact memory entry m_t, updating the bank: $\mathcal{M}_t = \mathcal{M}_{t-1} \cup \{\mathcal{S}(c_t)\}$. Each memory entry m_t is a structured object designed to maximize information density for downstream use. It discards implementation internals and retains only the architectural signature: **(i) Core Purpose ($\mathcal{P}_t$):** A concise natural language abstract of the module's responsibility. **(ii) Public Interface ($\mathcal{I}_t$):** A formal definition of all exported classes, functions, and method signatures (with type hints), enabling other modules to invoke c_t correctly without accessing its code. **(iii) Dependency Edges ($\mathcal{E}_t$):** A topological map documenting explicit *afferent couplings* (imported dependencies) and predicted *efferent couplings* (downstream consumers), enabling dependency-aware code generation.

In parallel, the next implementation target $\hat{c}_{t+1}$ is proposed based on the blueprint, dependency graph and current state. This information is separated from m_t and passed directly to $\mathcal{L}$ as input. By replacing raw code with these architectural signatures, CodeMem significantly reduces context overhead, enabling the synthesis of large-scale repositories while maintaining global consistency.

3.2.2 KNOWLEDGE GROUNDING WITH CODERAG

While CodeMem ensures internal structural consistency, LLMs remain prone to hallucination when facing implicit domain logic or complex library usages. To bridge this gap, we introduce **CodeRAG**, a

retrieval-augmented generation framework that grounds the synthesis process in high-quality external knowledge. Unlike standard RAG which retrieves based on query similarity, CodeRAG proactively aligns external repositories with the internal implementation blueprint.

Blueprint-Aligned Indexing. We treat reference repositories $\mathcal{R}$ (sourced from paper citations or web search) not as raw text, but as a source of structural patterns. The indexing process, modeled as $\mathcal{I}_{\text{index}} : \mathcal{R} \times \mathcal{B} \to \mathcal{J}$, transforms repositories into a structured knowledge base $\mathcal{J}$ explicitly mapped to our target blueprint $\mathcal{B}$, which involves:

(i)Relevance Filtering & Abstraction: We first filter $\mathcal{R}$ to retain only files relevant to $\mathcal{B}$'s architecture. An analysis agent then abstracts each file c'_s into a functional summary, discarding implementation noise while preserving algorithmic logic.

(ii)Structural Mapping: We establish explicit relational links between external source files and target modules through *Knowledge Tuples*, defined as $(c'_s, \hat{c}_t, \tau, \gamma)$. Here, c'_s is the external source, $\hat{c}_t$ is the target file in our blueprint, τ denotes the relationship type (*e.g.* Algorithmic Reference vs. Utility Pattern), and γ represents actionable context—extracted snippets and usage patterns tailored to guide the implementation of $\hat{c}_t$.

Adaptive Retrieval. To prevent context pollution from unnecessary retrievals, we employ an adaptive gating mechanism during generation. At step t, a decision function δ evaluates whether external grounding is needed. Specifically, the LLM assesses the complexity of the target file $\hat{c}_t$ against the blueprint's detail level and determines retrieval necessity via self-evaluation:

$$r_t = \delta(\mathcal{X}_t, \hat{c}_t) \in \{0, 1\} \tag{2}$$

If external grounding is deemed necessary ($r_t = 1$), the system queries index $\mathcal{J}$ for top-k tuples linked to $\hat{c}_t$. The actionable context γ from the linked tuple is then injected into the generation prompt, forming an augmented context $\mathcal{X}'_t = \mathcal{X}_t \cup \{\gamma\}$. This mechanism effectively injects proven implementation patterns "just-in-time", significantly reducing the likelihood of logical drift.

3.3 Phase 3: Verification and Refinement

This final phase implements a closed-loop error correction mechanism to address transmission errors—such as logic bugs or broken dependencies—that inevitably arise during code generation. We achieve this through two complementary feedback channels: static structural analysis and dynamic functional verification.

3.3.1 Static Structural Analysis

Prior to execution, we validate the code's structural correctness. A **Static Analysis Agent** examines the generated repository $\mathcal{P}$ against the implementation blueprint $\mathcal{B}$, scanning for two types of defects: (1) *Structural Discrepancies*, such as missing modules or empty files defined in the file hierarchy; and (2) *Quality Deficiencies*, including syntax errors or maintainability issues identified via static linting.

Rather than regenerating entire files—which consumes significant context window—we introduce a **Modification Agent** equipped with a Language Server Protocol (LSP) interface. This agent translates identified issues into precise, line-level patch instructions. We formalize the refinement as $\mathcal{P}' = \Phi_{\text{LSP}}(\mathcal{P}, \mathcal{R}_{\text{static}})$, where Φ_{LSP} applies targeted edits based on the analysis report $\mathcal{R}_{\text{static}}$, yielding a structurally valid repository $\mathcal{P}'$.

3.3.2 Dynamic Functional Verification

To ensure functional correctness, we deploy the repository $\mathcal{P}'$ into an isolated sandbox environment. The **Sandbox Agent** first provisions the runtime environment according to the blueprint's dependency specifications, then initiates iterative testing.

The agent autonomously generates test cases for the repository's entry points and captures the execution trace $\mathcal{T}_j$ at iteration j. Using the execution output (including standard error and stack traces) as corrective feedback, the agent diagnoses the root causes of any failures. When errors are detected ($\mathcal{T}_j^{\text{error}} \neq \emptyset$), the agent invokes the LSP-based modifier to produce the next repository state:

$$\mathcal{P}'_{j+1} = \Phi_{\text{LSP}}(\mathcal{P}'_j, \mathcal{T}_j^{\text{error}}) \tag{3}$$

This cycle continues until the code executes successfully or a maximum iteration limit J is reached, producing the final verified repository $\mathcal{P}^*$. The process ensures that the output is not merely plausible text, but a compiled, executable software artifact.

4 EXPERIMENTS

In this section, we evaluate the effectiveness of the proposed DeepCode framework by addressing the following 3 research questions: **RQ1:** How does DeepCode perform compared to existing agent frameworks? **RQ2:** How does the choice of different LLMs affect the performance of DeepCode? **RQ3:** What is the contribution of each module within the DeepCode architecture?

4.1 EXPERIMENTAL SETUP

Benchmark. We evaluate DeepCode on PaperBench Code-Dev (Starace et al., 2025), a challenging reproduction benchmark, where a model must recreate a functional ML research codebase *from scratch* using only the paper as reference, and produce an executable `reproduce.sh`. Submissions are graded by an author-approved, fine-grained rubric via the SimpleJudge pipeline (Starace et al., 2025).

Baselines. We compare against four baseline categories reported in prior work (Starace et al., 2025; Seo et al., 2026): (i) LLM-based agents (e.g., GPT-4o, o1, o3-mini, DeepSeek-R1, Claude 3.5 Sonnet, Gemini 2.0 Flash) under standard scaffolds; (ii) scientific code agents (PaperCoder/Paper2Code (Seo et al., 2026) and AutoP2C (Lin et al., 2026) on selected subsets); (iii) commercial code agents (Cursor (Anysphere, 2025), Claude Code (Anthropic, 2025), Codex (OpenAI, 2025)); and (iv) human experts (Best@3 on a 3-paper subset) (Starace et al., 2025). Further benchmark and baseline details are provided in Appendix A.3.

Experimental Protocol. We evaluate DeepCode under the following execution constraints and PaperBench grading protocol.

(i) Implementation Environment. All experiments are conducted within an Ubuntu 22.04 LTS-based sandboxed environment. This infrastructure is provisioned with a standard Python development stack and essential dependencies. DeepCode is configured to operate within this isolated space, retaining privileges for file system manipulation, shell command execution, and internet access, thereby simulating a standard software research and development workflow.

(ii) Task Execution. DeepCode accepts the target paper in both PDF and Markdown formats, along with any supplementary addenda, as primary inputs. To ensure that generated solutions stem from algorithmic reasoning rather than retrieval, a source-code blacklist is enforced during execution. This protocol precludes access to the authors' original repositories and known third-party implementations during web browsing. With input parameters defined and the search space constrained, DeepCode initiates its autonomous workflow for code generation and debugging.

(iii) Grading Methodology. Assessment of the generated code follows the PaperBench Code-Dev protocol, which focuses on structural and functional correctness and does not include post-submission reproduction. Grading is carried out by SimpleJudge, an automated system based on OpenAI's o3-mini, which performs static analysis of the submitted repository against a set of fine-grained, hierarchical criteria co-developed with the authors of the source paper. The judging logic is restricted to the Code Development leaf nodes of this rubric and examines core aspects of software quality, including static correctness (syntax validity and compliance with language standards), dependency validity (completeness and correctness of dependency specifications such as `requirements.txt`), project structure (coherent and consistent organization of files and directories), and algorithmic fidelity (faithful implementation of the algorithms and interfaces described in the original paper). This procedure is designed to align the evaluation with the central technical contributions of the work.

(iv) Evaluation Metrics and Protocol. Our primary evaluation metric is the Replication Score, which quantifies the proficiency of DeepCode in translating theoretical concepts into a functional codebase. The score for a single replication trial is derived from the hierarchical rubric through a bottom-up aggregation process. **(i) Leaf node scoring:** SimpleJudge first evaluates each leaf-node criterion on a binary basis, assigning a score of 1 for pass and 0 for fail. **(ii) Score aggregation:** The score for any parent node is then computed as the weighted average of the scores of its immediate

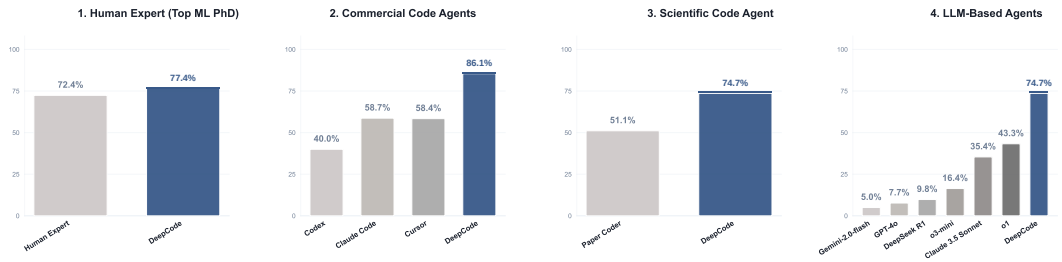


Figure 3: Comparison of DeepCode with four baseline categories: (i) human experts, (ii) state-of-the-art commercial code agents, (iii) scientific code agents, and (iv) LLM-based agents.

Table 1: Mean reproduction scores on a 5-paper subset. Each task cell is a per-paper mean, and Avg. is the equal-weight mean across papers. AutoP2C's FRE score is reused from the 3-paper evaluation; rows are not paired run by run.

Model	fre	rice	bam	pinn	mech-u	Avg.
Codex (GPT 5 Codex-high)	0.4095	0.3645	0.1937	0.5382	0.4926	0.3997
Claude Code (Claude Sonnet 4.5-think)	0.6286	0.3787	0.3829	0.7233	0.8222	0.5871
Cursor (Claude Sonnet 4.5-think)	0.6344	0.4186	0.3779	0.7748	0.7148	0.5841
AutoP2C (Claude Sonnet 4.5)	0.7667	0.6954	0.7242	0.7456	0.8013	0.7466
DeepCode (Claude Sonnet 4.5-think)	**0.8435**	**0.7380**	**0.8530**	**0.9474**	**0.925**	**0.8614**

children. The weights, predetermined during the rubric design, reflect the relative importance of each sub-task. **(iii) Final score derivation:** This recursive aggregation continues up the hierarchy until a single score is obtained for the root node, which serves as the Replication Score for that trial.

To account for the stochasticity inherent in code generation, we adopt a strict evaluation protocol. For each target paper, three independent replication trials are performed, and each resulting repository is scored separately by SimpleJudge using the procedure described above. The final Replication Score is the average of the three scores, mitigating outliers and providing a more stable and reliable measure of the model's typical performance. For benchmark and subset summaries, we report the equal-weight mean and sample standard deviation across the corresponding per-paper Replication Scores. Additional benchmark and baseline details are in Appendix A.3.

4.2 Main Results

Figure 3 summarizes the comparison with the four baseline categories.

- **Comparison against LLM Agents.** With BasicAgent, Claude-3.5-Sonnet achieves the highest score (35.4 ± 0.8), while other models range from 5.0 to 19.5. With IterativeAgent, o1 reaches 43.3 ± 1.1. DeepCode achieves 74.7 ± 11.1, approximately 72% higher in mean score than the strongest LLM-agent baseline under the PaperBench protocol.
- **Comparison against Scientific Code Agents.** PaperCoder achieves 51.1 ± 1.4, outperforming the general-purpose LLM-agent baselines. DeepCode reaches 74.7 on the full benchmark, a 23.6-point improvement over PaperCoder. On the 5-paper subset in Table 1, AutoP2C with Claude Sonnet 4.5 scores 0.7466 versus 0.8614 for DeepCode. The AutoP2C FRE value is reused from the 3-paper evaluation reported in Appendix 3; this subset comparison does not establish a full-benchmark ranking.
- **Comparison against Commercial Code Agents.** On the 5-paper subset in Table 1, DeepCode achieves 0.8614, compared with Codex (0.3997), Cursor (0.5841), and Claude Code (0.5871). DeepCode, Cursor, and Claude Code use the same base model, indicating that the observed advantage extends beyond backbone choice to the agent framework.
- **Comparison against Human Experts.** On the 3-paper subset, DeepCode achieves an average score of 77.4 ± 6.3, exceeding the reported human Best@3 score of 72.4. This comparison concerns

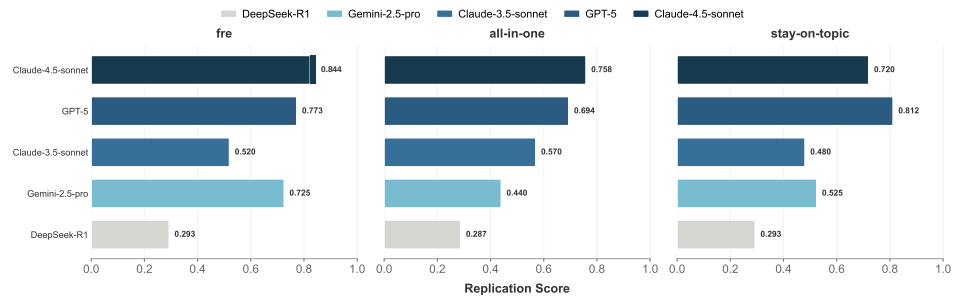
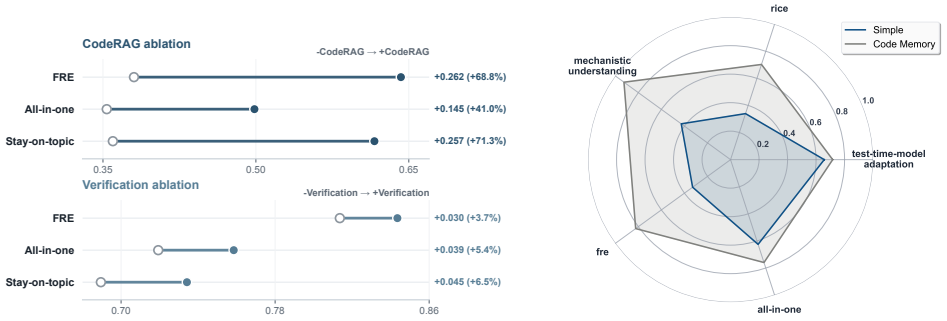


Figure 4: DeepCode reproduction results on the 3-paper subset across LLM backbones.



(a) Ablation of CodeRAG and Verification

(b) Ablation of CodeMem

Figure 5: Ablation studies of key components in DeepCode on PaperBench.

the Replication Score on this subset and uses different aggregation protocols: the agent's average performance and the best of three human attempts.

4.3 Analysis on Different LLMs

We evaluate DeepCode with five LLM backbones (Claude-4.5-Sonnet, GPT-5, Claude-3.5-Sonnet, Gemini-2.5-Pro, DeepSeek-R1) on three PaperBench tasks (fre, all-in-one, stay-on-topic). The tasks vary in specification complexity: fre and all-in-one contain long, interdependent setups with overlapping constraints, while stay-on-topic provides more structured descriptions. Agent architecture and tooling remain constant to isolate pure model capability effects.

As shown in Figure 4, reproduction scores exhibit consistent stratification across all three tasks. Claude-4.5-Sonnet achieves the best or near-best performance (0.72-0.84), demonstrating particular strength on fre and all-in-one where it more reliably reconstructs implementation details and multi-stage pipelines implied by complex, underspecified descriptions. GPT-5 tracks Claude-4.5-Sonnet closely on most metrics (0.69-0.81) and shows marginal advantages on stay-on-topic (0.81 vs. 0.72), suggesting additional robustness in maintaining alignment with fixed experimental framings, though this does not overturn Claude-4.5-Sonnet's overall dominance. Mid-tier models occupy an intermediate performance range: Claude-3.5-Sonnet (0.48-0.57) and Gemini-2.5-Pro (0.44-0.73) successfully recover main experimental skeletons but leave notable gaps in finer-grained procedural steps. DeepSeek-R1 consistently underperforms ($\approx$0.29), reproducing only fragments of target workflows across all tasks. This stable ranking pattern across heterogeneous specifications indicates that under fixed agent architecture, the underlying language model becomes the primary factor determining the ceiling and reliability of automatic paper-level reproduction.

4.4 Ablation Studies

In this section, we conduct ablation studies on three core components of DeepCode: CodeRAG, CodeMem, and Automated Verification. Specifically, we evaluate CodeRAG and Automated Verifica-

tion on a 3-paper subset (all-in-one, fre, stay-on-topic), while CodeMem is assessed on 5 randomly selected tasks (test-time-model-adaptation, rice, mechanistic-understanding, fre, all-in-one). Our key findings are summarized as follows.

(i) Impact of CodeRAG. To decouple the impact of CodeRAG, we conducted an ablation study using Gemini-2.5-Flash. As visualized in Figure 5a, the integration of CodeRAG delivers a performance leap (up to 70% relative gain), effectively breaking the base model's performance ceiling (0.35–0.38). Notably, we observed negligible gains when applying CodeRAG to frontier models like Claude 4.5 Sonnet. This contrast yields a critical insight: while reasoning giants likely encode sufficient implementation patterns within their parameters, cost-efficient models like Flash suffer from inherent *knowledge gaps*. Consequently, CodeRAG proves indispensable for these architectures, acting as a vital bridge to fill implicit domain voids with standard practices—confirming that external knowledge injection is essential for democratizing high-fidelity replication on lightweight models.

(ii) Impact of CodeMem. We ablate CodeMem's contribution on five PaperBench tasks using Claude-4.5-Sonnet, comparing DeepCode's structured memory against a "Simple" baseline that naively evicts historical messages via sliding windows when approaching context limits.

Results demonstrate that unstructured eviction causes context saturation with signal loss: the Simple protocol achieves only 0.33-0.43 in rice, fre, and mechanistic-understanding tasks due to dependency truncation, where foundational class definitions are discarded before dependent code generation. CodeMem's indexing maintains task-relevant signal density, restoring scores to 0.70-0.92 by preserving critical dependencies without exhausting context budgets. Even in scenarios with baseline performance (test-time-model-adaptation: $0.62 \rightarrow 0.72$; all-in-one: $0.66 \rightarrow 0.76$), Structured memory delivers consistent gains, confirming our core thesis: effective agentic coding requires explicit information flow management to maximize signal-to-noise ratio under context constraints.

(iii) Impact of Automated Verification. Across 3 test papers, Automated Verification yields consistent gains of 3.7–6.5%, elevating scores from 0.69–0.81 to 0.73–0.84. The layer corrects three types of errors: typos in variable names, missing dependencies, and wrong command-line arguments. These errors prevent otherwise sound implementations from executing reliably. The modest improvement reflects a fact: the earlier phases have achieved technical correctness. Verification is a final pass to ensure reliable execution. It eliminates small but consequential deviations that cause borderline implementations to fail, transforming them into faithful replications.

5 Conclusion

We introduced DeepCode, an autonomous framework for document-to-repository synthesis based on principled information-flow management. By organizing task-relevant information through blueprint distillation, stateful memory, conditional knowledge injection, and closed-loop verification, DeepCode mitigates the tension between complex paper specifications and finite context budgets. Empirical evaluations on PaperBench show that DeepCode achieves state-of-the-art performance among the evaluated automated agents, outperforms leading commercial agents on a 5-paper subset, and exceeds the reported PhD-level human Best@3 baseline on the 3-paper subset under the Replication Score metric. Our findings suggest that hierarchical information orchestration, rather than indiscriminate context scaling, is a promising direction for robust autonomous scientific reproduction.

AI use statement

Generative AI tools were used only to refine the language, clarity, and presentation of this manuscript. The authors take full responsibility for the final content of the paper.

Ethics statement

DeepCode is intended to support scientific reproducibility and the translation of technical specifications into executable repositories. Its potential benefits and risks are discussed in Appendix A.9. Generated code may contain insecure, brittle, or misleading implementations and should undergo human review and independent validation before use. Our evaluation uses the existing PaperBench benchmark and its previously reported human baseline.

REPRODUCIBILITY STATEMENT

Section 3 describes the DeepCode framework, and Section 4.1 specifies the execution environment, task constraints, grading procedure, and score aggregation. Appendices A.2 and A.3 provide further implementation and baseline details, while Appendix A.4 reports per-paper results across independent runs. Source code is provided through the anonymous repository at https://anonymous.4open.science/r/DeepCode-C464.

REFERENCES

Anthropic. Claude code: Agentic coding tool for your terminal. https://docs.claude.com/en/docs/claude-code/overview, 2025.

Anysphere. Cursor: The best way to code with ai. https://cursor.com, 2025.

ByteDance. Trae: The real ai engineer. https://www.trae.ai, 2025.

Yihong Dong, Xue Jiang, Jiaru Qian, Tian Wang, Kechi Zhang, Zhi Jin, and Ge Li. A survey on code generation with llm-based agents, 2025. URL https://arxiv.org/abs/2508.00083.

Yuyao Ge, Lingrui Mei, Zenghao Duan, Tianhao Li, Yujia Zheng, Yiwei Wang, Lexin Wang, Jiayu Yao, Tianyu Liu, Yujun Cai, Baolong Bi, Fangda Guo, Jiafeng Guo, Shenghua Liu, and Xueqi Cheng. A survey of vibe coding with large language models, 2025. URL https://arxiv.org/abs/2510.12399.

GitHub and OpenAI. Github copilot: Your ai pair programmer. https://github.com/features/copilot, 2025.

Google. Gemini cli: An open-source ai agent that brings the power of gemini directly into your terminal. https://github.com/google-gemini/gemini-cli, 2025.

Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework, 2024. URL https://arxiv.org/abs/2308.00352.

Dong Huang, Jie M. Zhang, Michael Luck, Qingwen Bu, Yuhao Qing, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation, 2024. URL https://arxiv.org/abs/2312.13010.

Md. Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. MapCoder: Multi-agent code generation for competitive problem solving. In *ACL*, pp. 4912–4944, 2024.

Peter Jansen, Oyvind Tafjord, Marissa Radensky, Pao Siangliulue, Tom Hope, Bhavana Dalvi Mishra, Bodhisattwa Prasad Majumder, Daniel S. Weld, and Peter Clark. Codescientist: End-to-end semi-automated scientific discovery with code-based experimentation, 2025. URL https://arxiv.org/abs/2503.22708.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

Zijie Lin, Yiqing Shen, Qilin Cai, He Sun, Jinrui Zhou, and Mingjun Xiao. AutoP2C: An LLM-based agent framework for code repository generation from multimodal content in machine learning papers. In *ICASSP 2026 - 2026 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 12487–12491. IEEE, 2026. doi: 10.1109/ICASSP55912.2026.11462205. URL https://doi.org/10.1109/ICASSP55912.2026.11462205.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist: Towards fully automated open-ended scientific discovery, 2024. URL https://arxiv.org/abs/2408.06292.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

OpenAI. Codex cli: Pair with codex in your terminal. https://developers.openai.com/codex/cli, 2025.

Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590*, 2023.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. Chatdev: Communicative agents for software development, 2024. URL https://arxiv.org/abs/2307.07924.

Zeeshan Rasheed, Malik Abdul Sami, Kai-Kristian Kemell, Muhammad Waseem, Mika Saari, Kari Systä, and Pekka Abrahamsson. Codepori: Large-scale system for autonomous software development using multi-agent technology, 2024. URL https://arxiv.org/abs/2402.01411.

Saoud Rizwan and others. Cline: Autonomous coding agent for vs code. https://github.com/cline/cline, 2024.

Minju Seo, Jinheon Baek, Seongyun Lee, and Sung Ju Hwang. Paper2Code: Automating code generation from scientific papers in machine learning. In C. Vondrick, B. Hariharan, C. Raffel, L. Pinto, D. Yang, and A. Faust (eds.), *International Conference on Learning Representations*, volume 2026, pp. 38867–38932, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/41b6674c28a9b93ec8d22a53ca25bc3b-Paper-Conference.pdf.

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. Paperbench: Evaluating ai's ability to replicate ai research, 2025. URL https://arxiv.org/abs/2504.01848.

Jiabin Tang, Lianghao Xia, Zhonghang Li, and Chao Huang. AI-Researcher: Autonomous Scientific Innovation. In *NeurIPS*, 2025.

Huanting Wang, Jingzhi Gong, Huawei Zhang, Jie Xu, and Zheng Wang. Ai agentic programming: A survey of techniques, challenges, and opportunities. *arXiv preprint arXiv:2508.11126*, 2025a.

Renxi Wang, Xudong Han, Lei Ji, Shu Wang, Timothy Baldwin, and Haonan Li. Toolgen: Unified tool retrieval and calling via generation. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=XLMAMmowdY.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: agent-computer interfaces enable automated software engineering. In *NeurIPS*, pp. 50528–50652, 2025.

Daoguang Zan, Ailun Yu, Wei Liu, Dong Chen, Bo Shen, Yafen Yao, Wei Li, Xiaolin Chen, Yongshun Gong, Bei Guan, et al. Codes: Natural language to code repository via multi-layer sketch. *ACM Transactions on Software Engineering and Methodology*, 2024.

Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. CodeAgent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *ACL*, pp. 13643–13658, 2024.

A APPENDIX

Organization of the Appendix. This appendix complements the main text by providing additional context, technical details, and extended evidence to support reproducibility. Sec. A.1 surveys related work on *general coding agents* and *scientific coding agents*, positioning DeepCode and DeepCode within the broader landscape. Sec. A.2 details the DeepCode framework, including hierarchical content segmentation for long-document indexing, multi-agent specification analysis (Concept/Algorithm Agents), and implementation blueprint synthesis as an intermediate representation for code generation and verification. Sec. A.3 provides additional benchmark and baseline details for PaperBench Code-Dev, complementing the execution and grading protocol described in Sec. 4. Sec. A.4 reports complete quantitative results, including benchmark comparisons and reproducibility analyses across papers and operational scenarios. Sec. A.5 presents DeepCode application cases with representative visualizations, illustrating end-to-end behavior across backend systems, web user interfaces, and the Paper2Code workflow. Sec. A.8 discusses open challenges and future directions for agentic software engineering, with emphasis on capability–efficiency trade-offs, lifelong (non-episodic) agent evolution, and dynamic planning under evolving constraints. Finally, Sec. A.9 summarizes limitations of the current workflow, including its plan-then-code structure and limited systematic replanning under evolving constraints.

A.1 RELATED WORK

A.1.1 GENERAL CODING AGENTS

The field of software engineering is being rapidly transformed by agentic systems that have evolved from passive code assistants into autonomous entities capable of planning, executing multi-step tasks, and self-correction (Dong et al., 2025; Ge et al., 2025). Research has explored several key architectures for these agents. One prominent trend involves multi-agent frameworks that emulate human development teams. This includes systems like ChatDev (Qian et al., 2024), MetaGPT (Hong et al., 2024), and CodePoRi (Rasheed et al., 2024), which simulate entire software company organizational structures to manage development tasks from scratch. For repo-level code generation, CodeS (Zan et al., 2024) proposed to decompose repository generation into specialized agents for structure planning and content filling. AgentCoder (Huang et al., 2024) employs atest-driven refinement loop involving programmer, test designer, and test executor agents, while MapCoder (Islam et al., 2024) mirrors human program synthesis with four agents handling example retrieval, planning, generation, and debugging. A second major trend focuses on enhancing agents with specialized tools and interfaces. For instance, CodeAgent (Zhang et al., 2024) integrates five domain-specific tools to support repository-level analysis, while SWE-agent (Yang et al., 2025) introduces a high-level Agent-Computer Interface (ACI) to enable robust agent interaction with file systems and development environments. In addition, ToolGen (Wang et al., 2025b) proposes representing each tool as a unique token and directly integrating tool-specific knowledge into the parameters of the LLM, thereby enabling a paradigm shift toward seamless unification of tool invocation and natural language generation.

Recent advancements in academic research are increasingly being translated into practical, productized tools. Commercial code agents emerging from this trend can be broadly categorized into two distinct paradigms: (1) AI-native integrated development environments (IDEs) such as Cursor (Anysphere, 2025) and Trae (ByteDance, 2025) that embed AI capabilities directly into the editor interface, and (2) terminal-based or extension-based agents including Claude Code (Anthropic, 2025), Gemini CLI (Google, 2025), Github Copilot (GitHub and OpenAI, 2025), and Cline (Saoud Rizwan and others, 2024) that operate through command-line interfaces or editor extensions. These coding agents leverage a holistic understanding of the codebase to perform complex tasks such as multi-file refactoring and autonomous edits. They support flexible, composable workflows and integrate seamlessly into diverse development pipelines. Commercial deployments indicate significant improvements in both function implementation and overall programming productivity. Despite their effectiveness, these agents suffer from context window limitations that impair their ability to process lengthy technical documents such as academic papers, and struggle to maintain coherence and correctness when synthesizing repository-level codebases.

A.1.2 SCIENTIFIC CODING AGENTS

In contrast to general-purpose coding agents, this class of agents targets more complex code generation scenarios, including the implementation and reproduction of entire codebases from high-level ideas and academic papers. For example, Paper2Code (Seo et al., 2026) addresses the research reproducibility crisis by transforming machine learning papers into executable repositories. Its code generation framework follows a structured three-stage process that includes system architecture design, implementation detail extraction, and modular code generation. AutoP2C (Lin et al., 2026) also targets complete repositories, combining multimodal paper parsing, repository-blueprint construction, hierarchical task decomposition, and iterative debugging. CodeScientist (Jansen et al., 2025) generates experimental code from literature, employing an iterative generate-execute-reflect cycle to write, run, and debug Python experiments. In addition, AlphaEvolve (Novikov et al., 2025) utilize code generation for algorithmic discovery, using an LLM as an evolutionary mutator to propose variations to entire codebases, which are then rigorously evaluated. Besides, the automation code in AI Scientist (Lu et al., 2024) and AI-Researcher (Tang et al., 2025) enables agents to iteratively plan and execute experiments, handle errors, and refine future runs based on results. AI Scientist focuses on experimental automation, maintaining execution history and generating plots and notes to support scientific write-ups. AI-Researcher extends this with a multi-stage refinement framework, where a code agent implements modular solutions and an advisor agent provides structured feedback for iterative validation, revision, and scaling. These agents have advanced the pace of scientific research, yet achieving higher generation efficiency without compromising code quality remains an open challenge.

A.2 DETAILS OF DEEPCODE FRAMEWORK

A.2.1 HIERARCHICAL CONTENT SEGMENTATION

The process of hierarchical content indexing is:

1. **Structural Parsing:** The source document $\mathcal{D}$ is parsed to identify its hierarchical structure based on explicit delimiters like section and subsection titles (*e.g.* "3. Methodology", "3.1. Model Architecture"). This divides the document into a set of content chunks $S = \{s_1, s_2, \ldots, s_K\}$.
2. **Keyword-Chunk Association:** Each chunk s_k is stored as a key-value pair (h_k, c_k), where the heading h_k serves as a natural, high-level semantic keyword, and c_k is the corresponding raw text content of that section.

This indexed structure effectively transforms the problem from one of long-context comprehension to a series of more manageable, on-demand retrievals. An agent no longer needs to process the entire document at once. Instead, it can query the index using semantic keywords to fetch only the most relevant context for its current task. This structured representation serves as the foundational input for the specialized agents that perform the detailed analysis in the subsequent steps.

A.2.2 MULTI-AGENT SPECIFICATION ANALYSIS

The design details of the Concept Agent and the Algorithm Agent are as follows:

Concept Agent. The Concept Agent is tasked with building a holistic, high-level understanding of the document. Its primary objective is to map the paper's entire conceptual structure, identify its core scientific contributions, and outline the necessary components for a successful experimental reproduction. Operating on the indexed document, the agent is instructed to use a segmented reading strategy, querying the index with semantically broad keywords (*e.g.* "introduction", "method"). This allows it to assemble a comprehensive overview by strategically fetching relevant sections. The output of this agent is a structured *Conceptual Analysis Schema*. This schema comprises a detailed paper structure map, a method decomposition map outlining the system's core functional components, an implementation map aligning claims with code requirements, and a reproduction roadmap specifying the criteria for success. Collectively, these elements translate the paper's narrative into a structured project plan.

Algorithm Agent. Complementing the conceptual overview, the Algorithm Agent is responsible for the meticulous extraction of every low-level technical detail required for an exact implementation. It's designed to perform an exhaustive search for all algorithms, mathematical formulations, model architectures, training procedures, and hyperparameters. Moreover, it can leverage online search capabilities to retrieve relevant algorithm implementations from the web as references. Like the Concept Agent, it leverages the segmented reading strategy but uses a distinct set of highly specific keywords (*e.g.* "algorithm", "hyperparameter") to perform targeted queries on the most technically dense sections of the document. The agent's output is a granular *Algorithmic Implementation Schema*. This schema captures verbatim pseudocode from algorithm boxes, exact mathematical equations and their variables, detailed layer-by-layer network architectures, and a comprehensive list of all hyperparameters with references to their locations in the paper. This schema serves as a precise, unambiguous technical specification that eliminates ambiguity during code generation.

A.2.3 Implementation Blueprint Synthesis

The final **Implementation Blueprint** $\mathcal{B}$ is a structured intermediate representation designed to be a unambiguous specification for code generation. This blueprint is organized into the following sections:

- **Project File Hierarchy:** A prioritized project file structure that dictates the logical organization of the codebase and the implementation order of its modules.
- **Component Specification:** A granular specification for every module, class, and function, explicitly mapping each to its corresponding algorithmic pseudocode and mathematical formulation.
- **Verification Protocol:** A formal plan for validating the final implementation. It defines the experimental setup, specifies the target metrics from the source document, and establishes the success criteria for reproduction.
- **Execution Environment:** A complete specification of all software dependencies, library versions, and requisite hardware configurations needed to compile and run the code.
- **Staged Development Plan:** A phased implementation roadmap that defines the build order of components and integrates staged verification checks to ensure modular correctness.

A.3 Benchmark and Baseline Details

Datasets. To evaluate DeepCode's capabilities in code comprehension and generation, particularly for automated vulnerability detection, we employ **PaperBench Code-Dev**, an innovative benchmark created by OpenAI (Starace et al., 2025). PaperBench Code-Dev assesses AI models' ability to independently reproduce leading ML research from major conferences like ICML 2024, focusing on 20 significant papers. Models are required to generate all necessary code from scratch, using only the research papers as references, without accessing existing codebases from the original authors. These tasks are performed in a virtual machine environment, with the goal of building a functional codebase, replicating experiments, and creating a `reproduce.sh` script for execution. Each paper is accompanied by a detailed evaluation rubric approved by the authors, which breaks down the reproduction task into 8,316 specific, gradable components, meticulously assessed using a hierarchical weighting scheme and SimpleJudge, a sophisticated automated judge powered by OpenAI's o3-mini model. This benchmark is rigorously crafted to challenge AI with tasks requiring advanced natural language understanding, algorithmic reasoning, and the ability to generate reliable code from abstract descriptions, all of which are crucial skills for automating vulnerability detection effectively.

Baselines. In order to evaluate the effectiveness of the proposed framework, we include a range of baseline methods for comparison. These baselines fall into four distinct categories:

(1) LLM Agents. We compare against results reported in Starace et al. (2025) for several state-of-the-art language models using two agent scaffolding approaches: (1) *BasicAgent*, a simple tool-use loop based on Inspect AI's basic agent that allows models to terminate early, and (2) *IterativeAgent*, which forces models to use their full allocated time and employs prompts designed to encourage incremental, piecemeal progress. All agents run in Ubuntu 24.04 Docker containers with access to a single A10 GPU, the internet, and standard development tools including bash, Python, web browsing, and file reading capabilities (Starace et al., 2025). The baseline models include GPT-4o, o1, o3-mini,

DeepSeek-R1, Claude 3.5 Sonnet, and Gemini 2.0 Flash, with most experiments using a 12-hour time limit (extended to 36 hours for select o1 runs).

(2) Scientific Code Agents. *PaperCoder* (Seo et al., 2026). PaperCoder (also referred to as Paper2Code) is a multi-agent LLM framework that transforms machine learning papers into executable code repositories via a three-stage pipeline: planning, which constructs implementation roadmaps, system architecture diagrams, and file dependencies; analysis, which extracts file-level implementation details; and generation, which produces modular code in dependency order. We additionally compare with *AutoP2C* (Lin et al., 2026) on two subsets: the matched FRE, ALL-IN-ONE, and STAY-ON-TOPIC comparison, using Claude Sonnet 4.5 for both systems under the same generation and PaperBench Code-Dev evaluation conditions; and the FRE, RICE, BAM, PINN, and MECH-U comparison in Table 1. The latter reuses AutoP2C's FRE score from the three-paper comparison and is not a paired-run evaluation. These subset comparisons are distinct from the full-benchmark results reported for PaperCoder.

(3) Commercial Code Agents. We compare against three state-of-the-art commercial code agents that provide AI-powered development assistance through different interfaces and capabilities:

- *Cursor* (Version 1.7.52) is an AI-assisted integrated development environment built as a fork of Visual Studio Code with additional AI features. Cursor allows developers to choose between cutting-edge LLMs and provides codebase embedding models that give agents deep understanding and recall (Anysphere, 2025). In our experiments, Cursor uses Claude Sonnet 4.5-thinking as the underlying model.
- *Claude Code* (Version 2.0.22) is Anthropic's agentic coding tool that lives in the terminal and helps developers turn ideas into code. Claude Code maintains awareness of the entire project structure, can find up-to-date information from the web, and with MCP can pull from external data sources like Google Drive, Figma, and Slack. It can directly edit files, run commands, create commits, and use MCP to read design docs or update tickets (Anthropic, 2025). Our evaluation uses Claude Sonnet 4.5-thinking.
- *Codex* (Version codex-cli 0.47.0) is OpenAI's coding agent that runs locally from the terminal and can read, modify, and run code on the user's machine. Codex is optimized for use with GPT-5-Codex for agentic coding, with configurable reasoning levels from medium to high for complex tasks. In auto approval mode, Codex can read files, make edits, and run commands in the working directory automatically (OpenAI, 2025). We configure Codex with GPT-5 Codex-high.

(4) Human Experts. The human baseline (Starace et al., 2025) consists of 8 ML PhD students and graduates from top institutions (*e.g.* Berkeley, Cambridge, Carnegie Mellon) who worked part-time over a four-week window on a 3-paper subset (all-in-one, fre, stay-on-topic). Participants had similar computational resources (A10 GPU) and could use AI coding assistants like ChatGPT and GitHub Copilot. The best-of-3 human attempts (Best@3) represent expert-level performance on this subset.

A.4 Full Results

This appendix reports quantitative results that complement the main text and provide a more systematic evaluation of DeepCode's overall capability and stability on research code reproduction tasks. Table 2 compares general-purpose code execution agents (including BasicAgent and IterativeAgent configurations), PaperCoder, and human experts; subset-specific rows are separated from full-benchmark results. The DeepCode aggregates are the equal-weight mean and sample standard deviation across the corresponding per-paper run means in Table 4, rounded for display. DeepCode achieves an average reproduction score of 74.7 ± 11.1 on the full benchmark, compared with PaperCoder (51.1 ± 1.4). On the 3-paper subset, DeepCode attains an average score of 77.4 ± 6.3, versus the reported human Best@3 score of 72.4; these use different aggregation protocols. Table 3 reports the AutoP2C comparison on these three papers.

Table 1 further selects a 5-paper subset (fre, rice, bam, pinn, mech-u) for comparison against AutoP2C and several widely used commercial code assistants (Codex, Claude Code, Cursor). Across these papers, DeepCode achieves the highest reported average score of 0.8614, compared with 0.7466 for AutoP2C and 0.5871 for the strongest commercial assistant. The AutoP2C FRE score is reused from the separate 3-paper evaluation, so these five-paper rows should not be interpreted as paired trial results.

Finally, Table 4 provides per-paper details for the Claude 4.5 Sonnet–based configuration, including three independent runs, their mean and sample standard deviation, as well as the associated average cost. Across a diverse set of targets—such as FRE, PINN, MECHANISTIC-UNDERSTANDING, and SEQUENTIAL-NEURAL-SCORE-ESTIMATION—DeepCode's reproduction scores typically lie in the 0.7–0.9 range with relatively small run-to-run variation, while the distribution of average cost across papers remains tight. This indicates strong cross-task generalization, stable behavior across repeated runs, and reasonable resource usage. Taken together, these appendix results reinforce the main conclusions of the paper: on realistic research code reproduction benchmarks, DeepCode not only achieves significantly higher average performance than existing automated reproduction and code assistance systems, but also demonstrates robust and consistent advantages in fine-grained, multi-paper, multi-run analyses.

Table 2: Average reproduction scores of automated agents and human experts. For DeepCode, $\pm$ is the sample standard deviation across per-paper mean scores. Rows below the double rule use the indicated 3-paper subset.

Model	**Average Replication Scores**
GEMINI-2.0-FLASH (BasicAgent)	5.0 ± 0.0
4o (BasicAgent)	7.7 ± 0.0
o3-mini (BasicAgent)	5.1 ± 0.8
o1 (BasicAgent)	19.5 ± 1.2
R1 (BasicAgent)	9.8 ± 0.0
CLAUDE-3-5-SONNET (BasicAgent)	35.4 ± 0.8
o3-mini (IterativeAgent)	16.4 ± 1.4
o1 (IterativeAgent)	43.3 ± 1.1
CLAUDE-3-5-SONNET (IterativeAgent)	27.5 ± 1.6
o1 [36 hours] (IterativeAgent)	42.4 ± 1.0
PaperCoder	51.1 ± 1.4
DeepCode	$\mathbf{74.7 \pm 11.1}$
Human [3 paper subset, Best@3]	72.4
AutoP2C [3 paper subset, Average]	71.59
DeepCode [3 paper subset, Average]	$\mathbf{77.4 \pm 6.3}$

Table 3: Matched AutoP2C comparison on three PaperBench Code-Dev tasks using Claude Sonnet 4.5. Entries are Replication Scores; the last row is the macro average.

Task	**DeepCode**	**AutoP2C**
FRE	84.35%	76.67%
ALL-IN-ONE	75.83%	69.36%
STAY-ON-TOPIC	72.00%	68.73%
Macro average	**77.39%**	71.59%

A.5 USE CASES FOR DEEPCODE

This appendix provides a series of visual artifacts generated by DeepCode, offering concrete evidence of its capabilities across different software development and research domains. These examples are intended to supplement the main paper by illustrating the practical utility and versatility of our system.

The initial set of examples, depicted in Figure 6, focuses on DeepCode's proficiency in generating sophisticated backend systems. The figures showcase automatically constructed administrative dashboards, which likely include functionalities for data monitoring, user management, and content

Table 4: DeepCode with Claude 4.5 Sonnet results. Mean and Std. Dev. summarize three runs; displayed run scores have up to four decimal places.

Paper	Run 1	Run 2	Run 3	Mean	Std. Dev.	Avg. Cost
FRE	0.844	0.823	0.8635	0.8435	0.020	9.14
RICE	0.738	0.7150	0.761	0.7380	0.023	8.22
BAM	0.853	0.9870	0.719	0.8530	0.134	8.45
WILL-MODEL-FORGET	0.776	0.793	0.857	0.808	0.042	9.20
PINN	0.947	0.9122	0.983	0.9474	0.035	7.84
ALL-IN-ONE	0.769	0.747	0.759	0.759	0.011	9.43
ADAPTIVE-PRUNING	0.547	0.570	0.516	0.544	0.027	9.13
LBCS	0.689	0.732	0.820	0.747	0.066	10.01
MECHANISTIC-UNDERSTANDING	0.889	0.944	0.941	0.925	0.031	10.20
TEST-TIME-MODEL-ADAPTATION	0.717	0.578	0.652	0.649	0.069	7.90
SAMPLE-SPECIFIC-MASKS	0.690	0.740	0.583	0.671	0.080	8.30
BRIDGING-DATA-GAPS	0.552	0.566	0.626	0.581	0.039	7.98
STAY-ON-TOPIC-WITH-CLASSIFIER-FREE-GUIDANCE	0.734	0.800	0.626	0.720	0.088	9.12
STOCHASTIC-INTERPOLANTS	0.851	0.792	0.801	0.815	0.031	8.89
LCA-ON-THE-LINE	0.665	0.844	0.739	0.749	0.090	7.73
SEQUENTIAL-NEURAL-SCORE-ESTIMATION	0.930	0.862	0.817	0.870	0.057	10.01
SAPG	0.702	0.755	0.757	0.738	0.031	9.19
FTRL	0.558	0.606	0.631	0.598	0.037	7.06
ROBUST-CLIP	0.772	0.742	0.685	0.733	0.044	7.83
BBOX	0.620	0.681	0.631	0.644	0.033	11.90

moderation. Such pages are critical for the operational management of modern web applications but are often tedious and repetitive to build. DeepCode's ability to scaffold these complex, data-driven interfaces from high-level specifications demonstrates its potential to significantly reduce boilerplate engineering and accelerate the deployment of robust server-side infrastructure.

Building upon the backend logic, a system's utility is often defined by its user-facing presentation. Figure 7 illustrates DeepCode's capacity for generating intuitive and functional Web UIs. The generated interfaces, featuring elements such as data visualization charts and interactive forms, translate abstract user requirements into tangible, interactive components. This capability not only complements the backend generation by providing a corresponding frontend, but also empowers developers and designers to rapidly prototype and iterate on user experiences, thereby shortening the path from concept to a functional product.

Perhaps DeepCode's most ambitious application, however, lies in its potential to bridge the chasm between academic research and practical implementation. The Paper2Code functionality, illustrated in Figure 8, exemplifies this capability. The figure is twofold: on the left, it presents the high-level code structure that DeepCode inferred from a research paper, discerning the architectural blueprint of the proposed algorithm, including its modular components and file organization. On the right, it provides a concrete code sample, instantiating a specific function or class with precise logic. This powerful feature moves beyond conventional code generation by interpreting unstructured scientific language to produce structured, executable artifacts, thereby holding immense promise for enhancing research reproducibility and accelerating the adoption of novel scientific discoveries.

A.6 Sub-Agents Details of DeepCode

DeepCode decomposes the software engineering pipeline into a set of specialized agents with narrow, well-specified responsibilities and standardized communication interfaces, rather than relying on a single monolithic generative model. The individual agents and their responsibilities are summarized in Table 5. This modular design allows different stages of the lifecycle—ranging from requirement understanding to architectural planning and code synthesis—to be implemented as transformations over shared intermediate representations, while preserving global architectural and semantic consistency.

During the planning stage, DeepCode relies on explicit coordination between conceptual and algorithmic analysis agents to derive a coherent development blueprint from high-level specifications. The Central Orchestrating Agent first routes each input through the Document Parsing and/or Intent Understanding agents to obtain a structured specification, which then serves as the

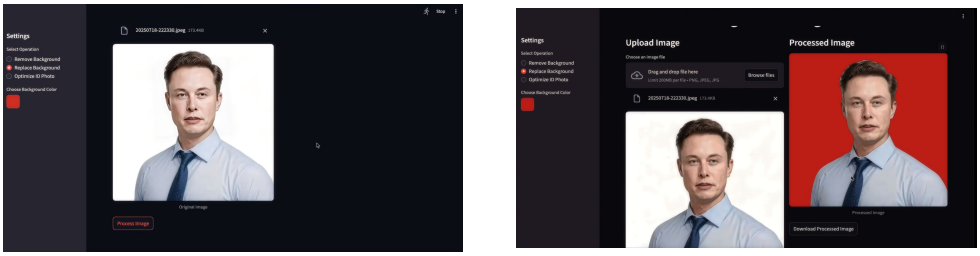


Figure 6: DeepCode-generated backend system pages.

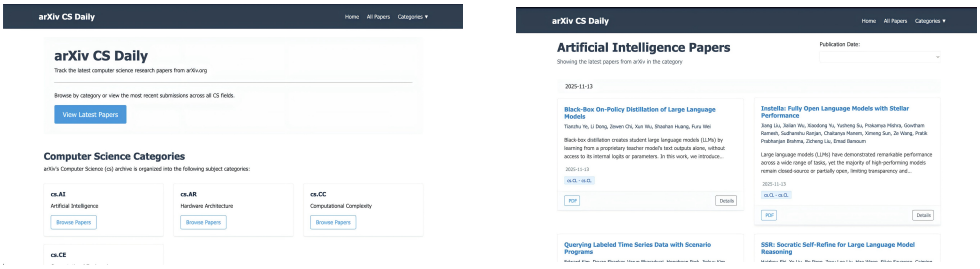


Figure 7: DeepCode-generated Web UI.

input to the Code Planning agent. Within this planning module, two internal analysis pipelines operate in parallel over the same intermediate representation. The conceptual analysis sub-agent is responsible for system-level decomposition: it identifies major subsystems, their responsibilities, and inter-module interfaces, and it constructs an architecture-level call topology. The algorithmic analysis sub-agent is responsible for computational aspects: it abstracts key algorithmic ideas, selects candidate data structures, reasons about time and space complexity constraints, and enumerates feasible implementation patterns. The partial plans produced by these two sub-agents are reconciled by a planning aggregation component (Code Analysis agent), which resolves inconsistencies and materializes a project-level development roadmap, including module boundaries, interface signatures, dependency relations, implementation priorities, and testing hooks. This roadmap serves as the design baseline that constrains all downstream code generation and refinement steps.

During the code synthesis stage, DeepCode couples retrieval-augmented reference mining with a global code memory, forming a closed-loop process that enforces repository-level consistency during incremental generation. On the retrieval side, the Code Reference Mining and Code Indexing agents implement a Retrieval-Augmented Generation (RAG) layer: they maintain multi-granularity indices over a corpus of prior implementations and expose to the Code Generation agent semantically relevant and structurally compatible code patterns, ranging from individual functions to reusable design idioms. In parallel, the Code Memory agent maintains a structured representation of the current repository state, including cross-file symbol tables, dependency graphs, and project-wide conventions such as naming schemes, error-handling strategies, and configuration mechanisms. Before emitting new code, the Code Generation agent issues queries to the Code Memory agent to obtain the up-to-date repository context and applicable constraints; after generation, it writes back the newly introduced symbols and dependencies, triggering an update of the global repository model. This query–constraint–update loop allows DeepCode to align local synthesis decisions with global architectural intent, reducing interface mismatches, naming drift, and latent coupling across the codebase.

Table 5: Functional Specifications of Specialized Sub-Agents in the DeepCode Framework

Agent Role	Functional Description
Central Orchestrating Agent	Functions as the central control unit, responsible for task decomposition, resource allocation, and the strategic coordination of sub-agents based on the complexity of the input requirements.
Intent Understanding Agent	Conducts semantic parsing of natural language inputs to extract functional requirements, converting ambiguous user descriptions into formal technical specifications.
Document Parsing Agent	Processes unstructured technical documents (e.g., research papers). It extracts multimodal information, including text, mathematical formulas, and diagrams, to establish a ground truth for implementation.
Concept Analysis Agent	Abstracts core theoretical concepts and logical flows from the parsed specifications, ensuring the computational model aligns with the theoretical underpinnings of the source material.
Algorithm Analysis Agent	Evaluates and selects appropriate algorithmic strategies and data structures. It focuses on optimizing computational complexity and feasibility before code synthesis begins.
Code Planning Agent	Formulates the software architecture and development roadmap. This agent determines the technology stack, designs modular file structures, and enforces design patterns to ensure scalability.
Code Reference Mining Agent	Retrieves external knowledge by identifying relevant open-source repositories. It analyzes dependency graphs to recommend integration patterns and library usages.
Code Memory Agent	Manages the state and context throughout the generation lifecycle. It utilizes hierarchical data structures to retain historical decisions and maintain semantic consistency across long-context interactions.
Code Generation Agent	Synthesizes executable source code based on the architectural plan and retrieved references. It implements functional interfaces and integrates distinct modules into a cohesive codebase.
Automated Validation Agent	Executes a rigorous quality assurance loop. It performs static analysis, generates unit tests, and iteratively debugs the codebase to verify functional correctness and adherence to specifications.

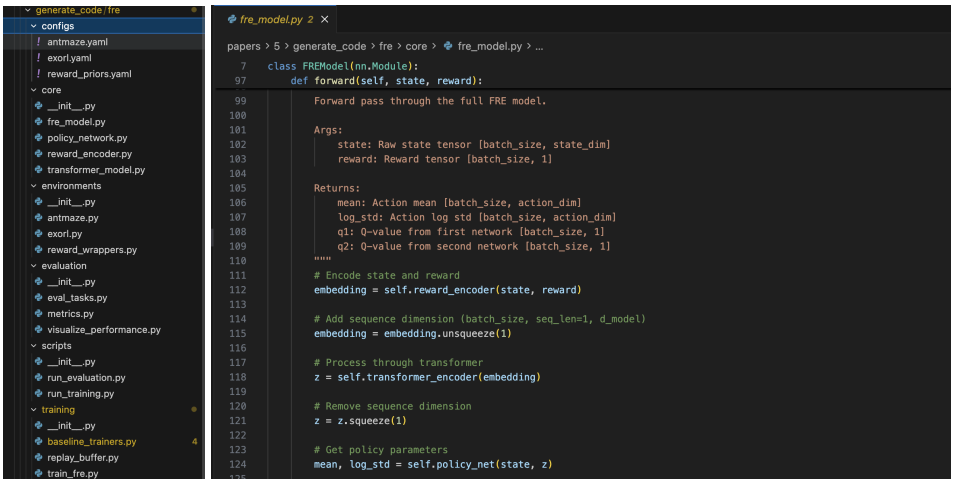


Figure 8: Paper2Code Samples of DeepCode. Left: Code Structure, Right: Code Sample

A.7 MCP Tool Stack in DeepCode

Table 6 summarizes the Model Context Protocol (MCP) tools integrated into DeepCode. The tools are grouped into three functional categories: *Perception & Retrieval*, *Cognitive Processing*, and *Action & Execution*. This organization makes the main stages of the system explicit. Perception & Retrieval tools give the model access to up-to-date web search results, web pages, and binary documents such as research papers and technical manuals, which helps mitigate the effects of the model's knowledge cut-off. Cognitive Processing tools then convert large codebases and long documents into semantic indexes and context-window-compatible segments, so that the model can issue natural language queries over existing artifacts and work with long technical materials. Action & Execution tools finally operate on the local development environment by reading and writing project files, executing shell commands, and interacting with the version control system.

Taken together, the tools in Table 6 form an end-to-end loop for assisted software development. The system can retrieve external and local information, reorganize it into internal structures that fit within the model's context window, and then apply code changes while observing their effects through commands such as tests or package installations. The table also shows that operations with side effects on the environment (file I/O, command execution, and Git operations) are confined to the *Action & Execution* layer and are described as sandboxed and path-validated. This separation between information access, semantic processing, and environment manipulation makes the extension of the base language model through MCP tools transparent and easier to reason about.

A.8 Discussion: Challenges and Future Directions

While DeepCode demonstrates the efficacy of principled information-flow management in high-fidelity repository synthesis, the transition from episodic coding tasks to autonomous, cost-effective, and self-evolving engineering remains fraught with challenges. We identify three critical frontiers that define the future trajectory of agentic software engineering.

(1) Agentic Capability and Computational Efficiency. SOTA performance in agentic coding currently relies on massive, proprietary LLMs (*e.g.* GPT-5, Claude 4.5), which incur prohibitive deployment costs and high latency. Conversely, smaller, open-weight models offer efficiency but lack the complex reasoning capabilities required for autonomous decision-making in open-ended engineering tasks. Bridging this gap presents a dichotomy of challenges. *(i) Fine-tuning limits:* Enhancing small models via supervised fine-tuning (SFT) is constrained by a data bottleneck—while raw code is abundant, high-quality agentic trajectories are scarce and expensive to curate. *(ii) Knowledge injection limits:* Merely augmenting small models with external knowledge is often insufficient; retrieved contexts may lack direct relevance to the specific coding task, and small models struggle to integrate complex inputs without suffering from attention dilution.

Table 6: Specification of Model Context Protocol (MCP) Tools Integrated into DeepCode. These tools extend the Large Language Model's capabilities across perception, cognitive processing, and environment manipulation domains

Category	**MCP Server Name**	**Functional Description & Academic Specification**
Perception & Retrieval	`brave_search`	A real-time information retrieval interface leveraging the Brave Search API. It provides the agent with temporal-aware access to web indices, enabling the retrieval of up-to-date documentation and resolving knowledge cut-off limitations.
	`bocha_mcp`	A specialized search module delivering structured "modal cards" and semantic summaries. It serves as a secondary knowledge source, optimizing token efficiency by returning structured entities rather than raw HTML.
	`fetch`	A web content ingestion engine that retrieves URL endpoints and normalizes heterogeneous HTML structures into clean Markdown. It acts as the agent's primary reading interface for external documentation.
	`pdf_downloader`	Binary resource acquisition tool designed for academic papers and technical manuals. It handles HTTP streams to ingest non-textual document formats (PDF/DOCX) for downstream processing.
Cognitive Processing	`code_reference_indexer`	A Retrieval-Augmented Generation (RAG) module for local codebases. It constructs a vector or semantic index of the project files, allowing the agent to perform natural language queries over the existing code structure.
	`document_segmentation`	A pre-processing utility implementing semantic chunking algorithms. It partitions large technical documents into context-window-compliant segments, facilitating the "Paper2Code" workflow for complex algorithm implementation.
Action & Execution	`filesystem`	A sandboxed file I/O interface allowing controlled read/write operations within the project directory. It enforces path validation security policies to prevent unauthorized system access during code generation.
	`code_implementation`	The core generative engine encapsulated as an MCP tool. It orchestrates the synthesis of functional code blocks, integrating logic planning with atomic file writing operations to ensure code coherence.
	`command_executor`	A runtime environment interface permitting the execution of shell commands (e.g., `pytest`, `pip install`). It establishes a feedback loop by capturing `stdout`/`stderr` for iterative debugging and self-correction.
	`git_command`	Version control management interface. It abstracts Git plumbing commands, enabling the agent to manage repository state, branch for experimental features, and maintain a clean commit history.

We envision a shift toward hybrid agentic architectures that synergize models of varying scales, employing large models for high-level reasoning and efficient small models for routine implementation. Besides, distilling knowledge from large models helps reduce the data bottleneck.

(2) From Episodic to Evolving Agents. Current coding agents typically operate in an episodic manner: they reset after each project, failing to carry over experience or tacit knowledge to subsequent tasks. Enabling agents to self-evolve and accumulate expertise mirrors human professional growth but faces significant hurdles. *(i) Reinforcement Learning constraints:* While RL-based optimization theoretically allows agents to learn from feedback, it requires well-defined reward functions, which are difficult to formulate for complex, multi-objective software engineering tasks. Moreover, this approach is inapplicable to closed-source LLMs where parameter updates are impossible. *(ii) Memory scalability issues:* The alternative approach—stacking historical experiences into a long-term memory—introduces severe noise. Simply accumulating raw interaction logs leads to context bloat, where retrieving relevant past experiences becomes a "needle in a haystack" problem.

Beyond relying on extensive manual annotation and training, a scalable solution involves automating the abstraction of past experiences. Future agents can implement post-task reflection to condense execution traces into reusable skills or heuristics. Storing these refined insights allows agents to retrieve corresponding high-level guidance, enabling self-evolution while avoiding context explosion.

(3) Dynamic Planning and Adaptability. Most existing frameworks utilize a linear Plan-then-Code workflow, assuming that all constraints are knowable a priori. In real-world engineering, specifications often evolve, and critical implementation constraints are frequently discovered only during the coding process. Separation between planning and execution leads to fragility: if the initial blueprint is flawed, the coding agent is often constrained by a stale plan, leading to suboptimal workarounds or failure.

Future researches advance toward dynamic, bidirectional planning frameworks. Agents are able to adapt their initial blueprints when encountering unforeseen constraints during implementation. Establishing a feedback mechanism where execution insights directly inform and update the high-level plan is crucial for handling the complex realities of large-scale software development.

A.9 LIMITATIONS

It is worth noting that DeepCode still, to some extent, follows the conventional plan-then-code paradigm in its workflow design: the system typically derives a high-level blueprint from the initial requirements and readily observable constraints, and then proceeds with implementation accordingly. This design offers strong controllability and efficiency when requirements are relatively stable and constraints are well specified. In practical software engineering, however, important constraints and implementation details—such as dependency interactions, interface boundaries, and performance or compatibility considerations—often surface only during coding, debugging, or integration. At present, DeepCode mainly incorporates such execution-time findings through localized adjustments, while support for continuously updating the high-level plan or conducting systematic re-planning remains limited. Consequently, when requirements or constraints evolve, additional human involvement may be needed to keep the plan and implementation aligned. Beyond these technical limitations, DeepCode may have positive impacts by lowering the cost and expertise barrier for reproducing scientific work and accelerating the translation of complex technical specifications into executable repositories. At the same time, its autonomous code-generation capability may introduce risks if generated repositories are adopted without sufficient human review, including insecure, brittle, or misleading implementations. Future work will investigate tighter closed-loop feedback mechanisms that allow insights from execution to directly inform and revise the plan, thereby improving adaptability and robustness in dynamic development settings.