Rubber Duck Research Preview: A Semantic Trust Layer for AI Coding Agents

Rubber Duck Research Team

SWE-bench Lite submission report – June 30, 2026

Abstract

AI coding agents have moved from local code completion to repository-level software engineering: they read issue reports, navigate large codebases, invoke tools, generate patches, and evaluate whether changes preserve existing behavior. This shift makes code understanding, context selection, and benchmark hygiene central systems problems rather than prompt-engineering details. We introduce the *Rubber Duck Research Preview*, a proprietary-but-auditable system for repository-level bug repair that combines a learned candidate-file routing component, a hosted Rubber Duck MCP interface, and a public Stage-2 orchestration scaffold. The preview intentionally exposes the evaluation harness, data contract, patch-emission format, and reproducibility boundary while abstracting the proprietary code representation, model internals, training data, and MCP implementation details.

On the 300-instance SWE-bench Lite split, the Research Preview resolves **232/300** tasks, for a **77.33%** Pass@1 resolve rate. The submitted run uses one final patch per task instance, no SWE-bench test knowledge, no hints, no web browsing, and per-instance base-commit indexing before MCP calls. The result is produced by the following high-level pipeline: *instance → proprietary file ranker → Claude Opus 4.8 + Rubber Duck MCP → structured patch emission → SWE-bench F2P/P2P evaluation*. This report explains the motivation, public/private boundary, evaluation methodology, aggregate results, comparison to recent open-scaffold reports, limitations, and next steps.

Contents

1 Introduction: software engineering in the age of AI coding agents

The software engineering workflow is being reorganized around AI coding agents. Earlier AI developer tools primarily offered local completions, short refactors, or natural-language-to-code generation. Modern agents are expected to operate at repository scope: they read issues, inspect multiple files, reason over APIs and tests, edit source code, and sometimes run validation loops. Benchmarks such as SWE-bench made this shift measurable by asking systems to resolve real GitHub issues from real Python projects, where a successful answer must edit the repository and pass the issue's tests [1].

This is a different problem from single-function code generation. A realistic issue may require recognizing where the bug lives, understanding the interaction between functions and files, preserving compatibility with existing behavior, and producing a patch that applies cleanly. SWE-bench's original formulation explicitly emphasized long contexts, execution environments, multi-file reasoning, and real project constraints as core difficulties [1]. Subsequent systems and reports show that progress comes from system design as much as from model quality: open-scaffold submissions such as ExpeRepair, Refact.ai Agent, SWE-agent, and KGCompass expose orchestration choices, memory, localization, or repository-structure components around frontier LLMs [15, 18, 17, 19].

The current moment is therefore not only about making a bigger model. It is about building reliable interfaces between models and codebases. For a developer, the question is no longer whether an LLM can write a plausible function. The question is whether an AI system can ground a change in the actual repository state, choose the right context, avoid stale or leaked information, and produce a minimal patch that is testable, reviewable, and reproducible.

Rubber Duck is our answer to that systems problem. The product vision is a semantic trust layer for AI coding: a layer that gives coding agents structured, repository-aware engineering knowledge instead of forcing them to infer an entire software system from raw text alone. The Research Preview described here is one evaluation snapshot of that broader program.

2 Motivation: why we built Rubber Duck

2.1 Autonomous software repair needs a engineer-like to represent code

In practice, the hardest part of an autonomous bug-fixing run is often not syntax generation. It is deciding what the model should look at. A repository can contain hundreds or thousands of files. The issue report may mention a symptom rather than the underlying function. A keyword match may retrieve the call site rather than the implementation. A vector retriever may retrieve semantically similar documentation rather than the file that actually controls the failing behavior. If the right file is missing, the model can still write a coherent patch - but it will be coherent in the wrong place.

This creates a structural bottleneck. Dumping more source code into the context window is expensive and can make the prompt worse rather than better. Under-retrieving misses the fix. Over-retrieving bloats the context with distractors. The central engineering problem is to put the right code in front of the model, with enough surrounding software-specific knowledge to patch correctly and without overwhelming the model's attention.

2.2 Long context does not remove the need for structure

Long-context models make repository-scale prompting possible, but they do not eliminate the need for selection. *Lost in the Middle* showed that models can perform worse when relevant information appears in the middle of long contexts rather than near the beginning or end [3]. RULER further showed that many long-context models that pass simple needle-in-a-haystack tests degrade on longer, more complex retrieval, tracing, and aggregation tasks [4]. LongBench v2 extends this picture to realistic long-context

tasks, including code repository understanding, and reports that deeper reasoning remains challenging even when very large contexts are available [5].

For software engineering, these findings matter because a codebase is not just a long document. It is a set of interdependent artifacts: source files, symbols, control paths, tests, dependencies, conventions, and history. A long prompt cannot by itself tell the model which relationships are semantically relevant. Without a code-aware prior, long context becomes a lossy compression of a repository rather than a reliable representation of the repository.

2.3 Naive RAG is not enough

Retrieval-augmented generation improves static LLMs by injecting external evidence at inference time, but RAG introduces its own failure modes. Recent surveys and experience reports highlight retrieval quality, context selection, grounding, robustness, latency, and validation as recurring challenges [6, 7, 8]. In a codebase, these problems become sharper: the retrieved artifact must correspond to the exact commit under evaluation, must preserve file and symbol identity, and must support edits that apply byte-for-byte.

A generic text retriever can answer "what looks relevant?" but a repair agent needs stronger guarantees: "what code exists at this base commit?", "which files define or call this behavior?", "which source span can be edited safely?", and "is this retrieved context stale?" Rubber Duck was built because we believe repository-aware retrieval and model-assisted repair require a representation that is more structured than plain text chunks and more controlled than ad-hoc agent browsing.

2.4 Agent loops can spend their budget navigating instead of repairing

Another failure mode is over-delegating navigation to the LLM. Multi-turn agents can read files, search, revise plans, and run tools, but each turn increases cost and creates opportunities for state drift. Agentless argued that a simpler localization-repair-validation pipeline can be competitive with more complex autonomous agents, suggesting that careful decomposition can outperform broad tool autonomy in some SWE-bench settings [2]. Our own pilot runs were consistent with that lesson: the model was strongest when it spent most of its budget reasoning over a well-chosen part of the software rather than wandering the repository.

2.5 Benchmark hygiene is part of the system

SWE-bench results are meaningful only if the model never sees the gold patch or tests. This is especially important for tool-using agents. A code-search service that indexes public repository HEAD can leak the fixed version of a file if the benchmark evaluates an earlier base commit. Studies around SWE-bench have also raised concerns about benchmark stability, insufficient tests, and data contamination [9, 10, 11]. For a hosted code-intelligence system, therefore, base-commit discipline is not an implementation detail; it is a core part of the evaluation protocol.

3 The Rubber Duck vision

3.1 A semantic verification and trust layer for AI coding

Rubber Duck is building a semantic verification and trust layer for AI coding agents. The goal is to make AI systems reason about a codebase closer to how a senior engineer reviews a change: by using structure, behavior, and repository knowledge rather than raw token similarity alone. Trust requires the agent to ground its actions in the exact repository state it is supposed to edit.

At a high level, Rubber Duck uses a proprietary code representation that captures multiple views of source code and repository. We use this representation during both training and inference. During training, it lets our internal models learn associations between issues, code locations, and repair-relevant knowledge from curated and/or proprietary data. During inference, it lets the system present a frontier LLM with a compact, high-value view of the repository and, when needed, expose additional controlled code-intelligence capabilities through MCP.

Disclosure boundary. We intentionally do not disclose the internal representation, construction rules, feature schema, training recipes, model architecture, checkpoints, or the implementation details of the hosted MCP services in this public report. The public contributions are the open source scaffold we released, the evaluation methodology, the submission result, and the data contract that allows the Stage-2 scaffold to be driven by an external ranker.

3.2 How the approach addresses the motivating failures

Rubber Duck addresses the limitations above with a separation of responsibilities.

- **Context bloating.** The proprietary Stage-1 component routes each issue to a small set of candidate files before the LLM call. The LLM is not asked to absorb the full repository.
- **RAG fragility.** MCP access is backed by base-commit-indexed repository intelligence rather than a generic, stale document store. The model can request additional knowledge through a controlled interface, but the interface is reset to the benchmark commit before each instance.
- **Agent drift.** Stage 2 is intentionally simple: top-K full files, one model call with structured patch emission, and only same-configuration recovery when the patch payload is empty or malformed.
- **Patch-format failures.** A typed `emit_patches` tool asks the model to produce explicit `SEARCH/REPLACE` edits instead of prose. The public scaffold validates and applies these blocks.
- **Reproducibility.** The public open scaffold exposes prompt construction, candidate-file ingestion, tool registration, patch application, prediction writing, and trace sanitization under an MIT license.

4 What is public today

4.1 Rubber Duck MCP

Rubber Duck exposes hosted MCP services as part of the product boundary. MCP is an open standard for connecting AI applications to external systems, tools, data sources, and workflows [12]. Rubber Duck uses MCP to give coding agents repository-aware capabilities such as code navigation and semantic retrieval over the relevant codebase state.

Reviewers can use the public Stage-2 scaffold without MCP through a file-only smoke path, or use Rubber Duck MCP credentials to reproduce the submitted configuration.

4.2 Research Preview scope

The Research Preview contains a minimal public slice of a larger Rubber Duck architecture. The exact submission used:

1. a proprietary file-ranking component, exposed to Stage 2 only through a JSONL candidate-file contract;

2. Claude Opus 4.8 as the patch-generating LLM;
3. Rubber Duck MCP services as an external repository-intelligence interface;
4. a public Stage-2 scaffold that performs prompt construction, MCP orchestration, structured patch emission, patch application, prediction export, and sanitized trace generation.

Internally, the ranking component is part of a broader Rubber Duck model architecture consisting of multiple proprietary model components plus the infrastructure currently supporting MCP. These components are trained or fine-tuned on curated and/or proprietary data. The Research Preview should therefore be read as an *open scaffold* release, not as an open release of the full Rubber Duck model stack.

4.3 Open scaffold contribution

The public scaffold is designed to be useful to reviewers and to the open-source community without releasing proprietary IP. It includes:

- the Stage-2 Opus + MCP orchestration loop;
- the Stage-1 JSONL schema consumed by Stage 2;
- a minimal public `SEARCH/REPLACE` patching fallback;
- a file-only smoke test that requires only an Anthropic API key;
- a trace sanitizer that removes raw hidden reasoning, raw MCP payloads, tokens, local paths, S3 paths, and private service logs;
- SWE-bench-compatible prediction export.

The scaffold is released under the MIT License. The license covers the public scaffold files only; it does not grant rights to Rubber Duck's private ranker, internal representation, hosted MCP implementation, private indexing services, training data, or model checkpoints.

5 SWE-bench Lite evaluation

5.1 Evaluation setup and constraints

We evaluated the Research Preview on SWE-bench Lite, a 300-instance subset of the original SWE-bench benchmark. Each instance contains a real GitHub issue, repository metadata, a base commit, and test information used by the evaluator. At inference time, the system sees the issue and base-commit code context, but it does not see `FAIL_TO_PASS`, `PASS_TO_PASS`, hints, web results, or gold patches.

Table 1: Submission configuration.

Field	Value
Dataset split	SWE-bench Lite, 300 task instances
Model	Claude Opus 4.8
Run mode	Single sequential worker, one instance at a time
Candidate context	File-level top-K, with $K = 10$
MCP mode	Rubber Duck MCP available in the submitted run; file-only mode available for smoke tests
Patch format	Structured `emit_patches` tool call with `SEARCH/REPLACE` blocks
Attempts	One final submitted patch per instance; same-configuration retry only on empty or malformed payload
Inference restrictions	No SWE-bench tests, no hints, no web browsing, no oracle access

5.2 Pipeline

Figure 1 shows the submitted pipeline in its public, IP-safe form. The proprietary ranker narrows the repository to candidate files. Stage 2 inlines the top-K files, allows additional controlled MCP queries when needed, requests a structured patch from the LLM, applies the patch, writes `predictions.jsonl`, and delegates verdict computation to the official SWE-bench Docker evaluation harness.

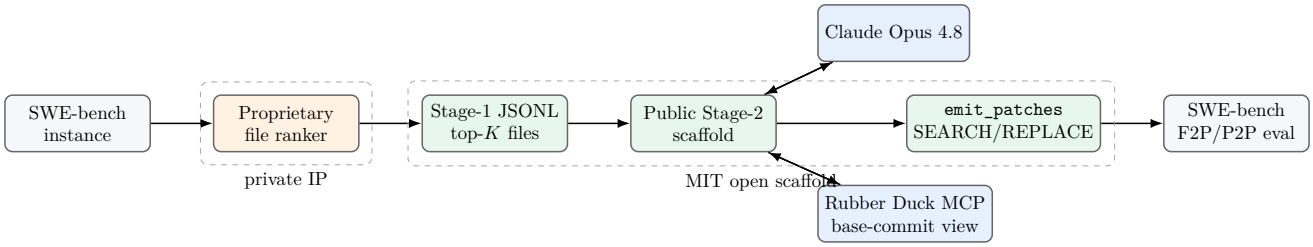

Figure 1: High-level submitted pipeline. The public scaffold exposes Stage-2 orchestration and the data contract; the proprietary ranker and hosted MCP internals remain private.

5.3 Base-commit discipline

Before each model call, the local repository is reset to the SWE-bench `base_commit`. The MCP-facing repository view is then initialized from that checked-out state. This step prevents the hosted MCP service from serving content from public repository HEAD, which could contain the upstream fix. If this initialization fails, the instance is marked as an indexing failure rather than sent to the model.

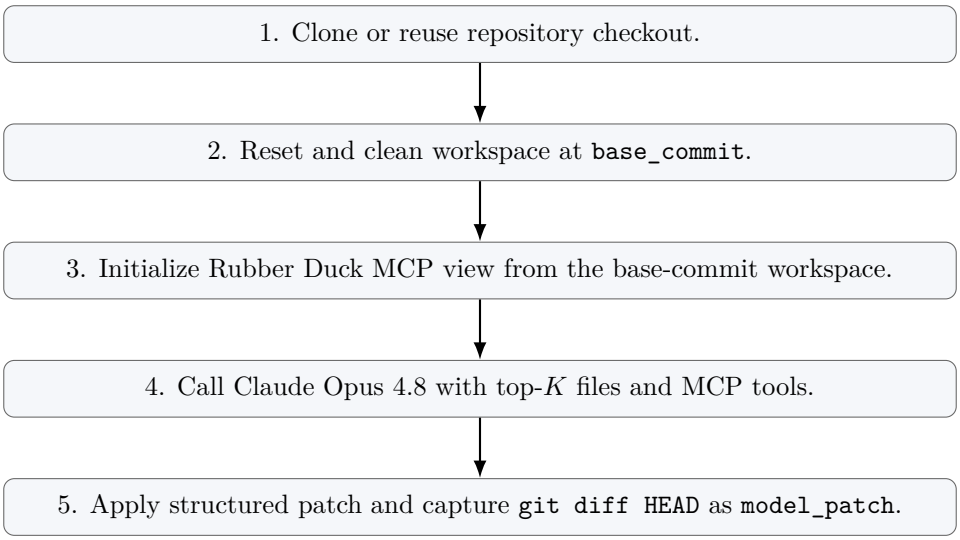

Figure 2: Base-commit discipline in the submitted run. The MCP view is derived from the exact evaluated commit, not from repository HEAD.

5.4 Headline result

The submitted run resolves **232 of 300** instances. The tally is closed: 232 resolved and 68 not resolved (59 wrong patches plus 9 empty patches).

Table 2: Headline result on SWE-bench Lite.

Metric	**Value**
Dataset split	SWE-bench Lite (300)
Resolved	232
Wrong patch: patch applied, `FAIL_TO_PASS` failed	59
Empty patch, `FAIL_TO_PASS` failed	9
No generation / no logs	0
Resolve rate	**77.33%**
Attempts per instance	1 final patch (Pass@1)
Web browsing / hints / oracle access	None
Wall-clock, single sequential worker	Approximately 24 hours

5.5 Per-repository results

Table 3: Resolve rate by repository.

Repository	Resolved	Total	Rate
pytest-dev / pytest	15	17	88.24%
matplotlib / matplotlib	20	23	86.96%
django / django	97	114	85.09%
scikit-learn / scikit-learn	19	23	82.61%
sphinx-doc / sphinx	12	16	75.00%
astropy / astropy	4	6	66.67%
pallets / flask	2	3	66.67%
psf / requests	4	6	66.67%
pylint-dev / pylint	4	6	66.67%
sympy / sympy	50	77	64.94%
pydata / xarray	3	5	60.00%
mwaskom / seaborn	2	4	50.00%
Total	**232**	**300**	**77.33%**

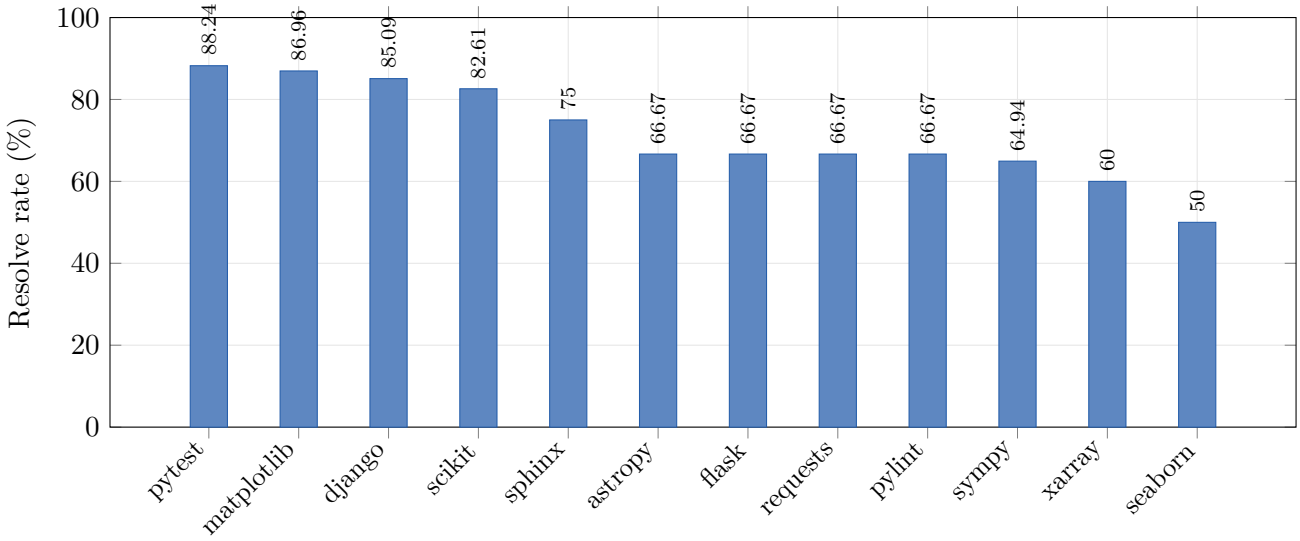

Figure 3: Per-repository resolve rate. The strongest repositories clear 85%; the weakest observed categories are dominated by numerically or behaviorally subtle tasks.

5.6 Per-year results

Table 4: Resolve rate by issue-creation year.

Year	Resolved / Total	Rate
2012	1 / 1	100.00%
2014	2 / 3	66.67%
2015	1 / 1	100.00%
2016	0 / 4	0.00%
2017	11 / 16	68.75%
2018	17 / 21	80.95%
2019	50 / 59	84.75%
2020	53 / 66	80.30%
2021	34 / 42	80.95%
2022	40 / 57	70.18%
2023	23 / 30	76.67%

5.7 MCP usage

MCP acted as a controlled safety net rather than the primary driver of every solution. Among the 232 resolved instances, 118 used zero MCP roundtrips. The mean number of MCP tool calls among resolved instances was approximately 3.6.

Table 5: MCP tool-call distribution across resolved instances.

MCP call bucket	Resolved instances	Share of resolved
0	118	50.9%
1–2	41	17.7%
3–5	37	15.9%
6–10	21	9.1%
11–25	12	5.2%
>25	3	1.3%

5.8 Patch shape

Resolved patches were usually small. In 87.9% of resolved instances, the patch touched exactly one file. Median patch shape was one file, one `SEARCH/REPLACE` block, four inserted lines, and two deleted lines.

Table 6: Patch shape among resolved instances.

Metric	P50	P90	Max
Files modified per patch	1	1	4
`emit_patches` blocks per patch	1	3	14
Inserted lines per patch	4	21	142
Deleted lines per patch	2	15	109

5.9 Failure breakdown and limitations

Table 7: Failure breakdown, reconciled against `all_preds.jsonl`.

Category	Count
Resolved (real code-change patch, `FAIL_TO_PASS` passed)	232
Wrong patch: real patch applied, `FAIL_TO_PASS` failed	59
Empty patch: no real patch emitted, `FAIL_TO_PASS` failed	9
Patch-apply failed	0
No-logs	0
Total resolved (official)	**232**
Total	300

All 232 resolved instances are backed by a real code-change patch that made the `FAIL_TO_PASS` tests pass. Verified independently against the corrected submission with the official `analysis/get_results.py` tool from `swe-bench/experiments`: `Resolved 232 instances (77.33%)`.

Manual inspection of the 59 real-patch failures suggests that most are not pure localization failures. In many cases, the correct file is present in the candidate context, but the hidden test exercises a stricter or more subtle behavioral contract than the issue statement makes obvious. This is especially visible in symbolic, numerical, and equivalence-sensitive repositories. Future work should therefore improve patch critique and behavioral specification inference without using SWE-bench tests at inference.

5.10 Comparison to recent open-scaffold reports

We reviewed recent high-performing open-scaffold approaches to ensure that this report includes the information SWE-bench reviewers expect: system description, public/private boundary, result breakdown, compliance checklist, artifact description, and metadata posture. Table 8 is not a claim about final leaderboard ordering; it is an approach-shape comparison against public submission artifacts and papers. We deliberately describe approaches by architectural family rather than by system name.

Table 8: Context from recent open-scaffold approaches on SWE-bench Lite.

Approach family	Resolved	Rate	Attempts	Report pattern
Dual-memory repair framework	181/300	60.33%	2+	Dual-memory repair framework; detailed trajectories and report link [15, 16].
Product-oriented open-source agent	180/300	60.00%	1	Product-oriented open-agent report with repo breakdown and compliance checklist [17].
Single-attempt scaffolded agent loop	170/300	56.67%	1	Concise single-attempt report with config link and compliance checklist [18].
Repository-graph structural repair	175/300	58.33%	2+	Repository-structure-driven repair report with arXiv link and repo breakdown [19, 20].
Retrieve-then-reason with proprietary ranker + hosted MCP (this work)	232/300	77.33%	1	Open Stage-2 scaffold, proprietary ranker/MCP boundary, sanitized traces, detailed result breakdown.

The SWE-bench submission checklist asks every submission to provide a technical report or blog post, include system details, and declare compliance with pass@1, test use, hints, web browsing, metadata, openness, and attempts fields [14]. This report follows that structure while redacting proprietary model and MCP internals.

6 Compliance and reproducibility boundary

Table 9: Submission compliance checklist.

Check	Status
Pass@1	✓ One final submitted patch per task instance. Same-configuration retry is used only for empty or malformed patch payloads; no candidate selection is performed.
No SWE-bench test knowledge	✓ `FAIL_TO_PASS` and `PASS_TO_PASS` are not visible to the model or the ranker at inference.
No hints	✓ The SWE-bench `hints` field is not used.
No web browsing	✓ The system has no web-browsing capability during inference.
No oracle access	✓ MCP is initialized from base-commit code before each instance and does not serve repository HEAD.
Open scaffold	✓ Stage-2 scaffold, schema, smoke test, patching fallback, and sanitizer are released publicly under MIT.
Open weights	✗ The LLM and Rubber Duck proprietary components are not open weights.

6.1 Artifacts intended to be public

Public artifacts should include the open scaffold, `predictions.jsonl`, sanitized trajectories, evaluation reports, and the SWE-bench metadata files. Sanitized trajectories may contain instance identifiers, candidate file names, high-level harness summaries, redacted MCP tool summaries, the final emitted `SEARCH/REPLACE` payload, and links to evaluation artifacts.

6.2 Artifacts intentionally kept private

The following must not be published: raw model hidden reasoning, raw MCP payloads, access tokens, private service logs, internal S3 or filesystem paths, training data locations, internal model names, internal code paths, proprietary representation details, model checkpoints, and raw per-instance debug JSONL. This boundary is essential: it lets the community inspect the scaffold and reproduce the orchestration behavior without disclosing Rubber Duck's proprietary ranker architecture or hosted MCP implementation.

7 Next steps

- **SWE-bench Full.** Extend the same architecture to the full SWE-bench split and report the result under the official evaluation protocol.
- **SWE-bench Pro.** Evaluate the stack on newer benchmark variants as they become part of the public evaluation workflow.
- **Proprietary models release.** Release descriptions of our proprietary model architectures, including from-scratch trained models and modified architectures, as soon as we can do so without compromising customer data, safety, or core IP.
- **Open-scaffold improvements.** Make it easier to plug in third-party rankers, local MCP test doubles, and alternative patch appliers.

- **Patch critique inside Pass@1.** Explore a second same-configuration critique pass over the draft patch without test feedback, web browsing, or candidate selection.
- **Continued research.** Improve repository representations, context routing, semantic retrieval, tool safety, trace sanitization, and contamination-resistant evaluation.

8 Conclusion

The Rubber Duck Research Preview shows that repository-level repair benefits from a specialized code-understanding layer around a frontier LLM. On SWE-bench Lite, the system resolves 232/300 instances under a clean Pass@1 protocol. The contribution is not only the score. It is the architecture boundary: a public, runnable Stage-2 scaffold that demonstrates how to compose a candidate-file prior, controlled MCP access, structured patch emission, and SWE-bench-compatible artifacts while protecting proprietary model and MCP IP. We believe this is the right trade-off for a research preview: enough openness for reviewers and the community to inspect the scaffold, enough discipline to avoid benchmark leakage, and enough abstraction to preserve the internal technology we are still developing.

References

[1] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* arXiv:2310.06770, 2023. https://arxiv.org/abs/2310.06770

[2] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. *Agentless: Demystifying LLM-based Software Engineering Agents.* arXiv:2407.01489, 2024. https://arxiv.org/abs/2407.01489

[3] Nelson F. Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. *Lost in the Middle: How Language Models Use Long Contexts.* Transactions of the Association for Computational Linguistics, 2024. https://arxiv.org/abs/2307.03172

[4] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. *RULER: What's the Real Context Size of Your Long-Context Language Models?* arXiv:2404.06654, 2024. https://arxiv.org/abs/2404.06654

[5] Yushi Bai, Shangqing Tu, Jiajie Zhang, Hao Peng, Xiaozhi Wang, Xin Lv, Shulin Cao, Jiazheng Xu, Lei Hou, Yuxiao Dong, and Jie Tang. *LongBench v2: Towards Deeper Understanding and Reasoning on Realistic Long-context Multitasks.* arXiv:2412.15204, 2024. https://arxiv.org/abs/2412.15204

[6] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Meng Wang, and Haofen Wang. *Retrieval-Augmented Generation for Large Language Models: A Survey.* arXiv:2312.10997, 2023. https://arxiv.org/abs/2312.10997

[7] Scott Barnett, Stefanus Kurniawan, Srikanth Thudumu, Zach Brannelly, and Mohamed Abdelrazek. *Seven Failure Points When Engineering a Retrieval Augmented Generation System.* arXiv:2401.05856, 2024. https://arxiv.org/abs/2401.05856

[8] Chaitanya Sharma. *Retrieval-Augmented Generation: A Comprehensive Survey of Architectures, Enhancements, and Robustness Frontiers.* arXiv:2506.00054, 2025. https://arxiv.org/abs/2506.00054

[9] Boxi Yu, Yuxuan Zhu, Pinjia He, and Daniel Kang. *UTBoost: Rigorous Evaluation of Coding Agents on SWE-Bench.* arXiv:2506.09289, 2025. https://arxiv.org/abs/2506.09289

[10] Linghao Zhang et al. *SWE-bench Goes Live!* arXiv:2505.23419, 2025. https://arxiv.org/abs/2505.23419

[11] Shanchao Liang, Spandan Garg, and Roshanak Zilouchian Moghaddam. *The SWE-Bench Illusion: When State-of-the-Art LLMs Remember Instead of Reason.* arXiv:2506.12286, 2025. https://arxiv.org/abs/2506.12286

[12] Model Context Protocol. *What is the Model Context Protocol (MCP)?* https://modelcontextprotocol.io/docs/getting-started/intro

[13] Rubber Duck. *Real semantics and a trust layer for AI coding.* https://www.rubberduck.com/

[14] SWE-bench Experiments. *Submission Checklist.* https://github.com/swe-bench/experiments/blob/main/checklist.md

[15] Fangwen Mu, Junjie Wang, Lin Shi, Song Wang, Shoubin Li, and Qing Wang. *ExpeRepair: Dual-Memory Enhanced LLM-based Repository-Level Program Repair.* arXiv:2506.10484, 2025. https://arxiv.org/abs/2506.10484

[16] SWE-bench Experiments. *ExpeRepair-v1.0 + Claude 4 Sonnet submission README.* https://github.com/SWE-bench/experiments/tree/main/evaluation/lite/20250625_ExpeRepair-v1_claude-4-sonnet-20250514

[17] SWE-bench Experiments. *Refact.ai Agent submission README.* https://github.com/SWE-bench/experiments/tree/main/evaluation/lite/20250425_Refact_Agent

[18] SWE-bench Experiments. *SWE-agent + Claude 4 Sonnet submission README.* https://github.com/SWE-bench/experiments/tree/main/evaluation/lite/20250526_sweagent_claude-4-sonnet-20250514

[19] SWE-bench Experiments. *KGCompass + Claude 4 Sonnet submission README.* https://github.com/SWE-bench/experiments/tree/main/evaluation/lite/20250906_KGCompass_claude-4-sonnet-20250514

[20] Boyang Yang, Jiadong Ren, Shunfu Jin, Yang Liu, Feng Liu, Bach Le, and Haoye Tian. *Enhancing repository-level software repair via repository-aware knowledge graphs.* arXiv:2503.21710, 2025. https://arxiv.org/abs/2503.21710