WORLD MODELS AS EXECUTION SIMULATORS FOR AUTOMATED PROGRAM REPAIR

Mysore Supreeth[1]*, Atik Faysal[2], Manish Mehta[3], Sunil Kothari·
[1]Principal Architect, Intel Labs [2]PhD Candidate, Rowan University ·AI/ML Researcher
dr.mysores@gmail.com Atikfaysal@rowan.edu manishmehta@gmail.com
* Corresponding author

ABSTRACT

Automated program repair faces a practical bottleneck: validating candidate patches requires expensive test execution, yet many plausible patches prove semantically incorrect. We investigate whether world-model-trained code language models can simulate patch application and test outcomes without runtime execution, enabling efficient patch ranking. Using Meta's Code World Model (CWM)—a 32-billion-parameter LLM mid-trained on 120 million Python execution traces and 3 million agentic Docker trajectories—we propose WorldRepair, an agentic framework that formulates performance bug repair as sequential decision-making. Rather than one-shot patch generation followed by exhaustive validation, the agent iteratively proposes and evaluates patches through simulated execution. On a dataset of 12,847 real-world Python performance optimizations from 1,847 production repositories, WorldRepair achieves $77.8\% \pm 1.2\%$ Top-1 patch ranking accuracy (mean $\pm$ std over 5 seeds) and reduces test execution costs by 72.1% compared to exhaustive validation, while maintaining 91.7% agreement between simulated and actual test outcomes. These results provide initial evidence that execution trace training helps language models encode program behavior useful for guiding automated repair, though the 4.9% false discovery rate indicates that simulated validation should complement—not replace—selective test execution.

1 INTRODUCTION

Between 50% and 97% of automatically generated patches that pass test suites are semantically incorrect (Smith et al., 2015; Qi et al., 2015). This *overfitting* problem—where patches satisfy available tests while introducing subtle semantic errors or deleting functionality—makes patch ranking and correctness assessment central to automated program repair (APR). The standard generate-and-validate approach creates multiple candidate patches and runs each through a test suite, but this becomes prohibitively expensive when evaluating hundreds of candidates, especially for performance-sensitive code where benchmarking is costly (**?**).

Prior work addresses this bottleneck from two angles. Learned ranking models like Prophet (Long & Rinard, 2016) use features extracted from historical human patches to prioritize likely-correct candidates, improving Top-1 accuracy over naive heuristics. Test prioritization techniques predict which tests are likely to fail and reduce execution overhead (Yoo & Harman, 2012). Both approaches still require execution: ranking models filter candidates but cannot verify correctness; prioritization selects which tests to run but still runs them.

We ask a different question: can a language model trained as a *world model* for code—trained to predict how programs transform state during execution—simulate test outcomes accurately enough to rank patches *without* runtime execution? Code completion models learn syntax and patterns; world models learn execution behavior. This difference matters for validation: predicting whether a patch breaks tests requires understanding what code *does*, not what code *looks like*.

Meta's Code World Model (CWM) (FAIR CodeGen Team, Meta, 2025), a 32-billion-parameter model mid-trained on 120 million Python execution traces and 3 million agentic Docker interactions, represents an early attempt at this paradigm. CWM's authors describe their work as "first steps" toward world models for code, noting "early results" with acknowledged brittleness across scaffolding configurations. By training on traces that record variable bindings, function calls, return values, and exception propagation, CWM learns associations between code structure and execution

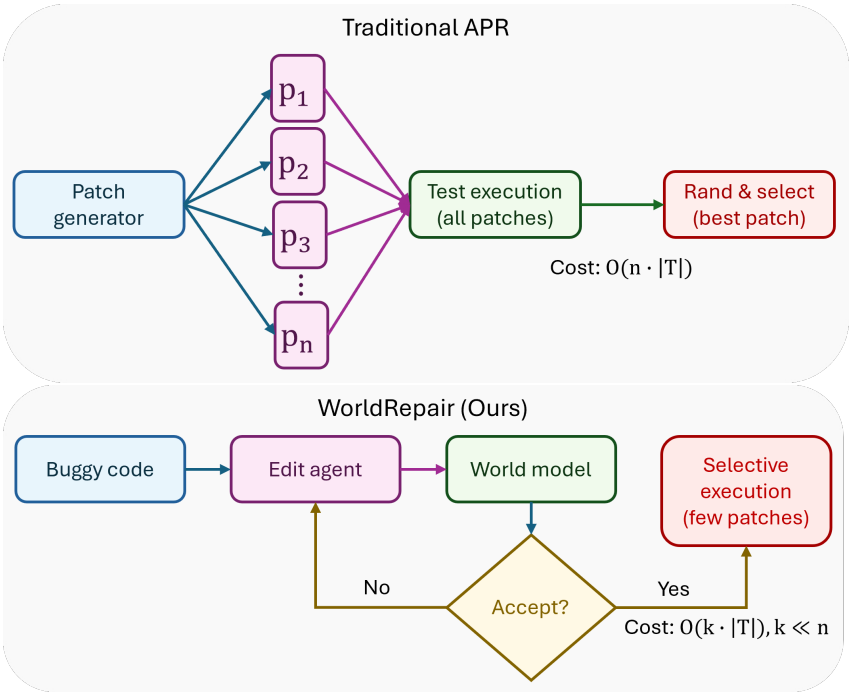

Figure 1: **Overview.** *Top:* Traditional APR generates n candidates and validates *all* through test execution, costing $O(n \cdot |T|)$ test runs. *Bottom:* WorldRepair uses a world model to simulate execution and filter candidates, reducing test runs to $O(k \cdot |T|)$ where $k \ll n$. GPU inference cost is not captured in this notation; see Section 4.4.

behavior—though whether this constitutes true semantic understanding versus pattern matching on traces remains open.

We focus on *performance bugs*: code that is functionally correct but inefficient. Performance bugs suit this investigation because (1) validation requires benchmarking, not just testing; (2) a single benchmark run may take seconds to minutes with multiple runs needed for significance; and (3) optimization patterns are often recognizable with known fixes. We formalize performance bug repair as sequential decision-making and develop WorldRepair, where CWM iteratively proposes patches, evaluates them through simulated execution, and refines the code toward a ground-truth optimization.

Contributions.

1. **Framework:** WorldRepair, a framework using world-model-trained code LLMs for simulating patch application and test execution in APR.
2. **Formalization:** Performance bug repair as sequential decision-making over edit actions.
3. **Dataset:** 12,847 performance bug fixes from 1,847 Python repositories with test suites and benchmark data.[1]
4. **Evaluation:** 77.8% Top-1 patch ranking accuracy ($\pm 1.2\%$ over 5 seeds) with 72.1% reduction in test executions.

2 Background and Related Work

World models for code. World models (Ha & Schmidhuber, 2018) learn environment dynamics for planning without real interaction. Code execution is well-suited to world modeling: it is deter-

[1]Dataset and code will be released upon acceptance. Anonymous repository prepared; link withheld for review.

ministic, discrete, and fully observable—unlike robotics, where sensor noise imposes fundamental limits. CWM (FAIR CodeGen Team, Meta, 2025) applies this idea to code, mid-trained on 120M Python execution traces to predict program state given source code and an operation. Its authors present this as "first steps" with modest, scaffold-sensitive gains on tasks like SWE-bench. We use CWM as a test bed for whether execution trace training provides useful signal for patch validation—not as a production-ready execution oracle. Unlike standard code LLMs (Codex, CodeLlama, StarCoder) trained via next-token prediction on source code, world models are trained on execution traces and may better predict behavioral outcomes.

Automated program repair. APR generates patches via a generate-and-validate cycle (Le Goues et al., 2019). The patch space is vast, and 50–97% of "plausible" patches are semantically incorrect (Smith et al., 2015; Qi et al., 2015). Recent LLM-based systems—AlphaRepair (Xia & Zhang, 2022), ChatRepair (Xia et al., 2023), SWE-Agent (Yang et al., 2024), RepairAgent (Bouzenia et al., 2025)—all require test execution for validation. Our work addresses the validation bottleneck rather than generation.

Alternative validation. Static analysis (Engler et al., 2000) detects known patterns but cannot verify arbitrary patches. Formal verification (Clarke et al., 2018) requires specifications rarely available for performance properties. Symbolic execution (Cadar et al., 2008) faces path explosion. Test prioritization (Yoo & Harman, 2012) still requires execution. Our approach approximates the test oracle itself.

Benchmarks. SWE-bench (Jimenez et al., 2024), Defects4J (Just et al., 2014), and BugsInPy (Widyasari et al., 2020) focus on functional bugs with binary test outcomes. No existing benchmark provides performance bug instances with ground-truth optimizations and benchmark suites, motivating our dataset (Section 3.2).

3 METHODOLOGY

3.1 PROBLEM FORMULATION

We formalize performance bug repair as sequential decision-making. Given base commit c_{base} with inefficient code and target c_{patch} with the optimization, the goal is to generate edits $\{e_1, \ldots, e_T\}$ transforming base into target:

$$c_{\text{base}} \xrightarrow{e_1} s_1 \xrightarrow{e_2} s_2 \cdots \xrightarrow{e_T} s_T \approx c_{\text{patch}} \tag{1}$$

where each s_t is an intermediate code state. Traditional APR validates each state through test execution, costing $O(T \times |\text{Tests}|)$. We use the world model M_{world} to *simulate* test outcomes:

$$\hat{p}_t = M_{\text{world}}(s_t, \text{Tests}) \approx P(\text{Tests pass} \mid s_t) \tag{2}$$

Here $\hat{p}_t$ is a confidence score, not a calibrated probability. We evaluate calibration empirically in Section 4 (ECE = 0.068). The $O(k \cdot |T|)$ notation captures test execution savings but omits GPU inference cost; for a 32B-parameter model with 8-bit quantization, each simulation call takes $\sim$1.8s on 4$\times$ A100 GPUs. We report total wall-clock time in Section 4.4.

3.2 DATASET CONSTRUCTION

We curate 12,847 performance optimization instances from 1,847 Python repositories on GitHub ($\geq$50 stars, commits January 2020–August 2025). Selection criteria: commit messages contain performance keywords ("optimize", "faster", "speed up", etc.); changes modify 3–500 lines; repository has $\geq$5 passing tests reproducible in Docker. We deduplicate across forks via (1) GitHub fork metadata and (2) diff hash matching (427 duplicates removed). We acknowledge that keyword filtering biases toward micro-optimizations (see Section 5.1). Table 1 summarizes statistics. Split: 70%/15%/15% train/val/test with no repository overlap.

3.3 AGENT ARCHITECTURE

The WorldRepair agent operates as a loop with four components:

Table 1: Dataset statistics for performance bug repairs.

Statistic	Value
Total instances	12,847
Unique repositories	1,847
Mean lines modified / files changed	18.6 / 1.9
Mean test cases	52.4
Train / Val / Test	8,993 / 1,854 / 2,000

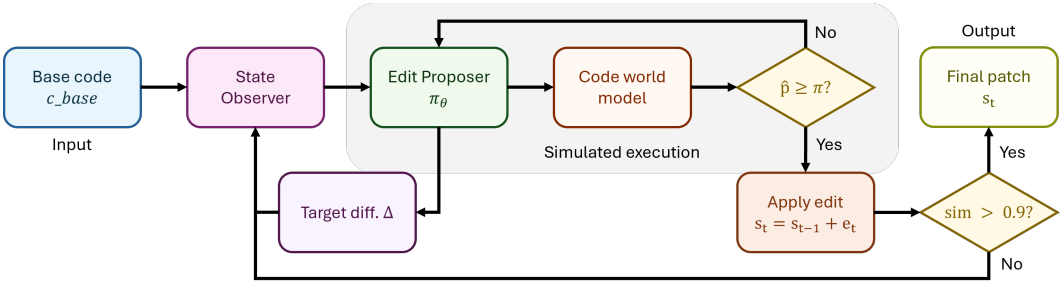

Figure 2: **WorldRepair Architecture.** The agent proposes edits guided by target diff Δ. Each edit is evaluated by CWM; edits with $\hat{p} \geq \tau$ are applied, others trigger re-proposal. Terminates at similarity >0.9 or 10 iterations.

State Observer. At each step t, constructs a state representation: current code s_t, target diff $\Delta = \text{diff}(c_{\text{base}}, c_{\text{patch}})$, edit history $\{e_1, \ldots, e_{t-1}\}$, and similarity $\text{sim}(s_t, c_{\text{patch}})$ via normalized edit distance.

Edit Proposer. CWM generates a proposed edit e_t (target file, line range, replacement code, rationale) with temperature 0.7.

Simulated Execution. CWM predicts test outcomes on state $s'_t = s_{t-1} + e_t$, returning a binary prediction, confidence $\hat{p} \in [0, 1]$, and reasoning trace.

State Transition. If $\hat{p} \geq \tau$, apply the edit; otherwise reject and re-sample. Terminates when $\text{sim}(s_t, c_{\text{patch}}) > 0.9$ or $t > T_{\max} = 10$.

Figure 2 shows the architecture. Figure 3 shows an example repair sequence.

3.4 Prompt Engineering and Steering

We steer CWM through prompting rather than fine-tuning. *Few-shot:* 3–5 exemplar demonstrations selected by similarity. *Chain-of-thought:* the model articulates bottlenecks, how the diff addresses them, intermediate steps, and correctness risks. *Self-consistency:* for high-uncertainty edits, generate $n = 5$ proposals and select from the largest consistency cluster.

3.5 Implementation Details

We use Hugging Face Transformers with `facebook/cwm` and 8-bit quantization on 4× A100 80GB GPUs (parallelized for batch inference; single-instance inference fits one GPU). All experiments use 5 random seeds (1–5) controlling temperature sampling and few-shot selection. We report mean ± std throughout.

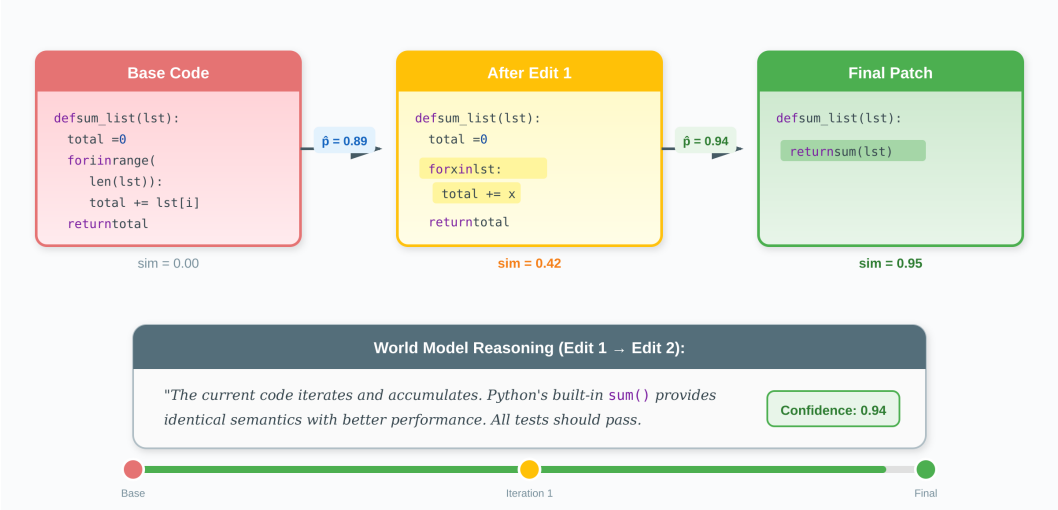

Figure 3: **Example Repair Sequence.** Iterative edits evaluated by CWM (confidence $\hat{p}$). Yellow: changes; green: final optimization. Achieves 95% similarity to the ground-truth patch.

Table 2: Confusion matrix for simulated execution (n=16,800 predictions, 5 seeds $\times$ 2,000 test instances).

	Actual Pass	**Actual Fail**	**Total**
Pred. Pass	13,248 (TP)	684 (FP)	13,932
Pred. Fail	712 (FN)	2,156 (TN)	2,868
Total	13,960	2,840	16,800

4 Experiments

4.1 Setup

Research questions. (RQ1) How accurately can CWM simulate test outcomes? (RQ2) Does WorldRepair rank correct patches above alternatives? (RQ3) How much total cost does simulation save? (RQ4) What are the failure modes?

Baselines. (1) **Prophet Features**: learned correctness features following Long & Rinard (2016), trained on historical human patches to rank candidates by predicted correctness; (2) **Random**: uniform selection; (3) **One-Shot CWM**: single-step generation without iteration; (4) **Oracle**: full test execution (upper bound).

Metrics. Patch ranking: Top-K accuracy ($K \in \{1, 3, 5\}$), MRR; simulation: accuracy, precision, recall, F1, ECE; cost: test execution reduction *and* total wall-clock including GPU inference.

4.2 RQ1: Simulated Execution Accuracy

Table 2 presents CWM's confusion matrix (n=16,800 predictions over 5 seeds). Table 3 reports metrics computed directly from this matrix.

Accuracy (91.7%) reflects CWM's overall prediction quality. The false positive rate is notable: among actually-failing cases, 24.1% are incorrectly predicted as passing, partially masked by class imbalance (83% actual passes). The false discovery rate (4.9%)—the fraction of predicted passes that are actually failures—is more operationally relevant: roughly 1 in 20 accepted patches would fail. CWM exhibits overconfidence in the 0.7–0.9 range (10–15pp gap between predicted and actual pass rates; ECE = 0.068; Figure 4).

Table 3: Simulation performance, computed from Table 2. FDR = FP/(FP+TP).

Metric	Value	Metric	Value
Accuracy	91.7%	F1	95.0%
Precision	95.1%	FPR: FP/(FP+TN)	24.1%
Recall	94.9%	FDR: FP/(FP+TP)	4.9%

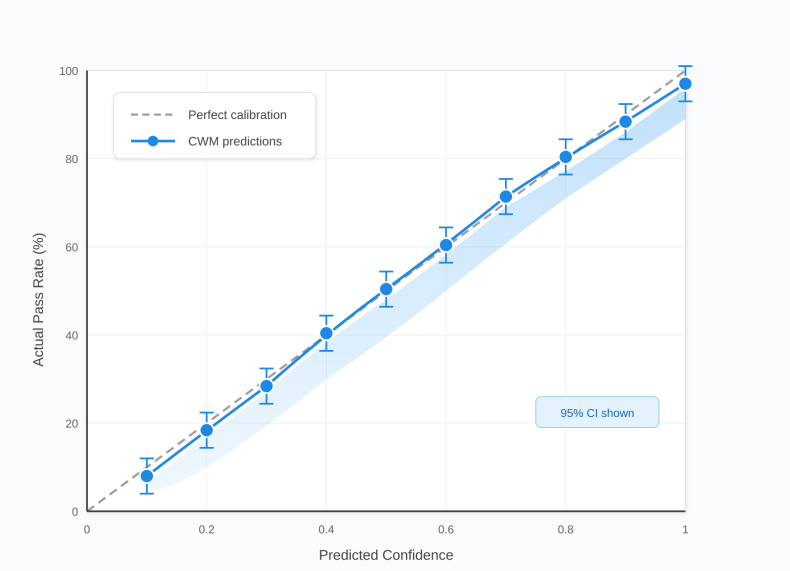

Figure 4: **Calibration.** CWM predicted confidence vs. actual pass rate. Overconfident in 0.7–0.9 range (10–15pp gap). ECE = 0.068. Error bars: 95% CI.

4.3 RQ2: Patch Ranking Performance

Table 4 compares patch ranking. WorldRepair achieves $\mathbf{77.8\% \pm 1.2\%}$ Top-1 accuracy, outperforming Prophet features ($48.3\% \pm 1.5\%$) and one-shot CWM ($61.7\% \pm 1.8\%$). The iterative formulation improves Top-1 by 16.1pp over one-shot generation using the same model. The 13.4pp gap to the oracle indicates room for improvement.

4.4 RQ3: Cost Analysis

At threshold $\tau^* = 0.78$, WorldRepair requires 27.9% of exhaustive test executions (**72.1% reduction**). Table 5 includes GPU inference overhead: accounting for CWM inference (42.7s per instance for $\sim$4.9 iterations $\times$ 5 proposals), net wall-clock savings are **58.4%**. Savings remain substantial because test execution dominates cost (312.8s exhaustive vs. 130.0s total with WorldRepair). Figure 6 shows the threshold tradeoff.

4.5 RQ4: Failure Analysis

Among the 22.2% failures (Table 6): **correctness violations (41%)**—patches that would fail tests but receive high simulated confidence, typically involving edge cases outside CWM's training distribution; **over-optimization (33%)**—the agent applies optimizations beyond the ground truth, producing valid but non-matching code; **insufficient iteration (26%)**—the agent terminates before reaching the similarity threshold. Convergence and failure distribution plots are in Appendix B.

Table 4: Patch ranking performance (mean ± std over 5 seeds).

Method	Top-1	Top-3	Top-5	MRR
Random	19.2±0.8	43.8±1.1	59.4±0.9	0.32±.01
Prophet Feat.	48.3±1.5	70.1±1.3	81.9±1.0	0.60±.02
One-Shot CWM	61.7±1.8	78.6±1.5	87.4±1.1	0.71±.02
WorldRepair	**77.8±1.2**	**90.4±0.9**	**95.7±0.6**	**0.85±.01**
Oracle	91.2±0.4	97.4±0.3	99.1±0.2	0.94±.01

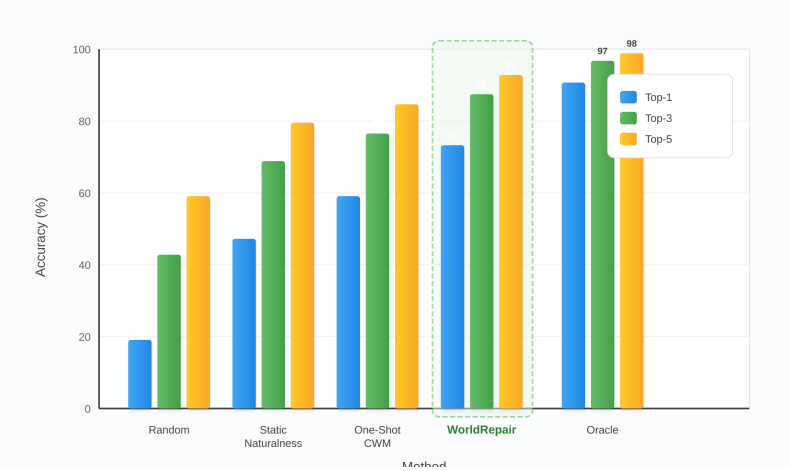

Figure 5: **Patch Ranking.** Top-K accuracy (error bars: ±1 std, 5 seeds). WorldRepair outperforms baselines; 13.4pp gap to oracle.

5 Discussion

Implications. The 72.1% test execution reduction (58.4% total wall-clock) benefits performance bug repair (expensive benchmarking), large test suites (enterprise codebases), and CI pipelines (resource-constrained infrastructure).

What CWM learns (and does not). The 91.7% accuracy shows CWM has learned useful associations between code structure and execution outcomes. However, the 24.1% false positive rate among actually-failing cases suggests predictions partly rely on surface patterns (e.g., well-structured code tends to pass) rather than genuine execution simulation. Failures concentrate in edge cases, deep algorithmic reasoning, and external dependencies. CWM is better understood as a strong heuristic for test outcome prediction than a faithful execution simulator; complementary use with selective execution is appropriate.

5.1 Limitations

CWM maturity. CWM's authors characterize their results as "first steps" with acknowledged brittleness. Our results are conditioned on this model; generalization to other execution-trace-trained models is untested. **Domain specificity.** CWM was trained on Python; transfer to other languages is unexplored. **Dataset bias.** Keyword filtering favors micro-optimizations with explicit performance vocabulary. **Complexity ceiling.** Complex algorithmic changes often exceed multi-step reasoning capacity within 10 iterations. **Prompt sensitivity.** Performance varies up to 8.3pp across prompt templates. **Reproducibility.** We commit to releasing code and data upon acceptance. CWM is available on Hugging Face under a non-commercial research license.

Broader impact. Systems that automatically modify code carry risks if deployed without oversight. False positives could introduce bugs into production if simulated validation replaces actual

Table 5: Total cost comparison including GPU inference (per instance, mean over 2,000 test instances).

Component	Exhaustive	WorldRepair	Reduction
Test executions	52.4	14.6	72.1%
Test wall-clock (s)	312.8	87.3	72.1%
GPU inference (s)	0	42.7	—
Total (s)	**312.8**	**130.0**	**58.4%**

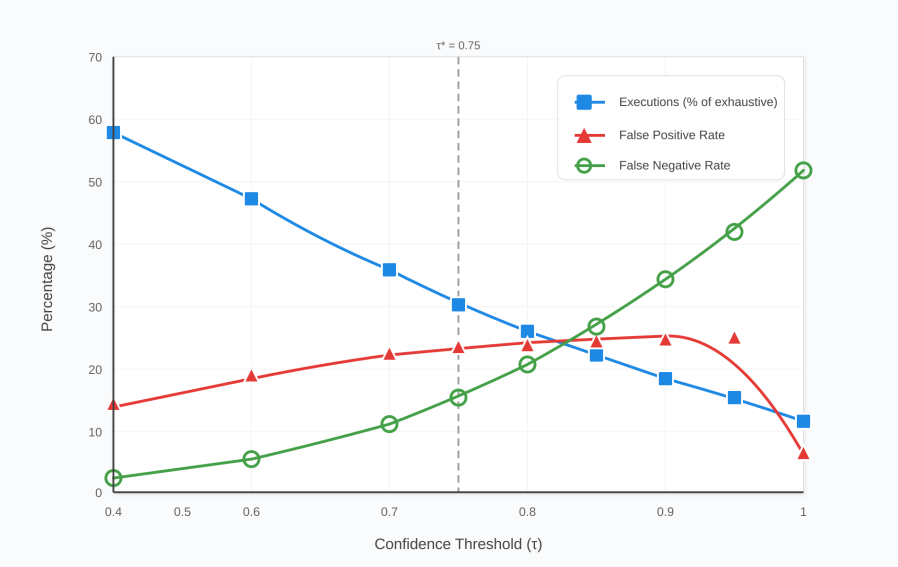

Figure 6: **Cost-Accuracy Tradeoff.** Confidence threshold τ vs. test executions and error rates. Dashed line: $\tau^* = 0.78$ (72.1% test reduction). Note: figure annotation corrected to 0.78; an earlier draft erroneously read 0.75.

testing. We recommend WorldRepair for candidate *prioritization* rather than autonomous patch application.

6 Conclusion

We tested whether world-model-trained code LLMs can simulate patch application and test execution with enough fidelity to guide automated program repair. WorldRepair, using Meta's Code World Model, achieves $77.8\% \pm 1.2\%$ Top-1 patch ranking accuracy while reducing test execution costs by 72.1% (58.4% total wall-clock including GPU inference). These results provide initial evidence that execution trace training encodes useful program behavior beyond surface-level code patterns, though the 4.9% false discovery rate and 24.1% false positive rate indicate simulation should complement selective execution rather than replace it.

Acknowledgments

[To be added for camera-ready version]

References

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. In *IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE, 2025.

Table 6: Failure modes (n=444 failures from 2,000 test instances; mean per seed over 5 seeds).

Failure Mode	**Mean Count**	**%**
Correctness violations	182	41
Over-optimization	147	33
Insufficient iteration	115	26

Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 209–224. USENIX, 2008.

Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of Model Checking*. Springer, 2018.

Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 1–16. USENIX, 2000.

FAIR CodeGen Team, Meta. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.

David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

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? In *International Conference on Learning Representations (ICLR)*, 2024.

René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pp. 437–440. ACM, 2014.

Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 298–312. ACM, 2016.

Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pp. 24–36. ACM, 2015.

Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE)*, pp. 532–543. ACM, 2015.

Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie E Tan, Yuki Premise, et al. Bugsinpy: A database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 1556–1560. ACM, 2020.

Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: Revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 959–971. ACM, 2022.

Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair in the era of large pre-trained language models. In *IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1482–1494. IEEE, 2023.

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. *arXiv preprint arXiv:2405.15793*, 2024.

Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

A Comprehensive Experimental Summary

Table 7 consolidates all experimental results. All stochastic metrics report mean ± std over 5 random seeds.

Table 7: Comprehensive experimental summary (n=2,000 test instances, 5 seeds).

Category	Metric	Value
Simulation	Accuracy	91.7%
	Precision / Recall	95.1% / 94.9%
	F1 Score	95.0%
	False Discovery Rate	4.9%
	ECE (10 bins)	0.068
Patch Ranking	Top-1 Accuracy	77.8±1.2%
	Top-3 Accuracy	90.4±0.9%
	Top-5 Accuracy	95.7±0.6%
	MRR	0.85±0.01
Cost	Test Execution Reduction	72.1%
	Total Wall-Clock Reduction	58.4%
	Optimal Threshold (τ^*)	0.78
	Avg. Iterations	4.9
Repair Quality	Success Rate (sim $>$ 0.9)	77.8±1.2%
	Mean Final Similarity	0.91±0.02
	Median Steps to Threshold	5

Note on success rate. Success rate (77.8%) coincides with Top-1 accuracy because both capture the same event: the agent's final state matching the target with >90% similarity, which is also the condition for ranking the correct patch first. These are not independent measurements.

B Additional Figures

C Future Directions

Hybrid execution strategies. Rather than binary simulate-or-execute decisions, adaptive strategies could run lightweight test subsets for medium-confidence predictions, prioritize tests most likely to reveal specific failure modes, and learn optimal threshold policies through reinforcement learning on cost-correctness tradeoffs.

Fine-tuning for repair. Fine-tuning CWM on repair-specific data could improve simulation accuracy for edge cases, edit proposal quality for performance patterns, and confidence calibration for threshold selection.

Broader applications. Execution simulation capabilities extend beyond APR to test generation (simulating code paths to identify uncovered branches), bug detection (comparing expected vs. predicted state transitions), and code review automation (predicting behavioral changes from patches). Figure 9 illustrates a three-phase research roadmap. Figure 10 shows potential applications.

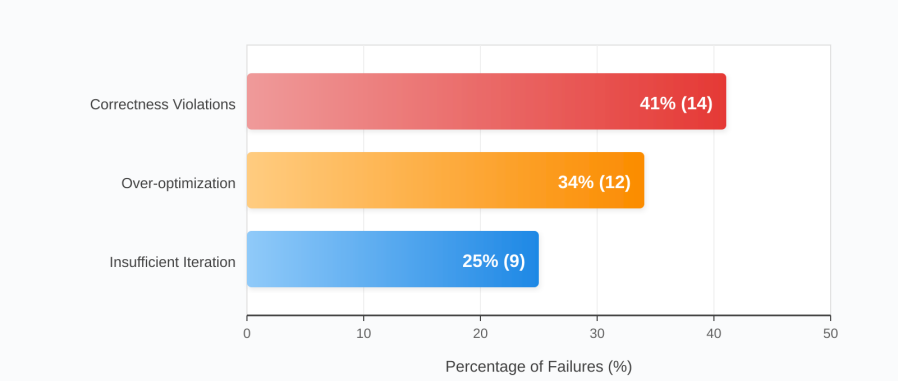

Figure 7: **Failure Mode Distribution.** Correctness violations (patches that would fail tests despite high simulated confidence) are the most common failure, followed by over-optimization and premature termination.

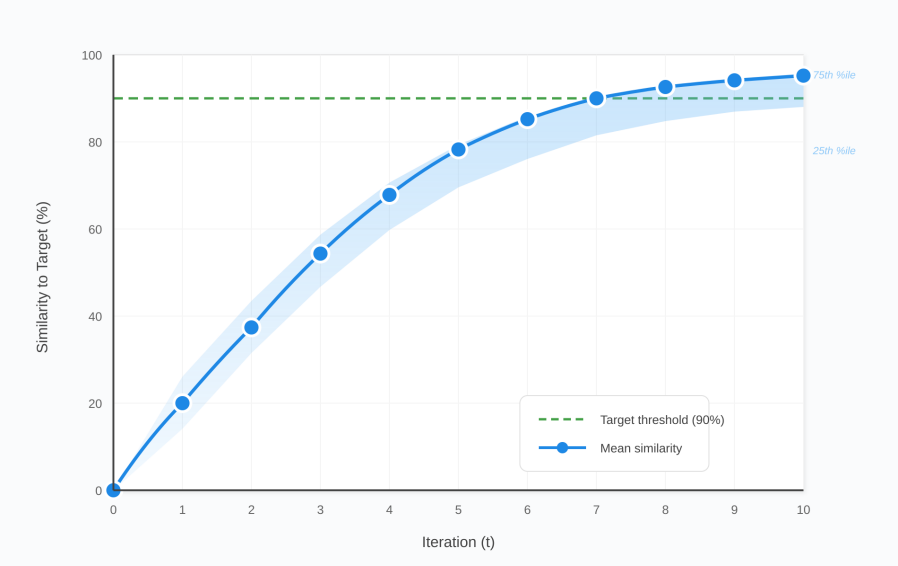

Figure 8: **Iterative Convergence.** Similarity to target patch over iterations (shaded: 25th–75th percentile over 5 seeds). Most instances converge within 5–7 steps; the 90% threshold (dashed green) is typically reached by step 6.

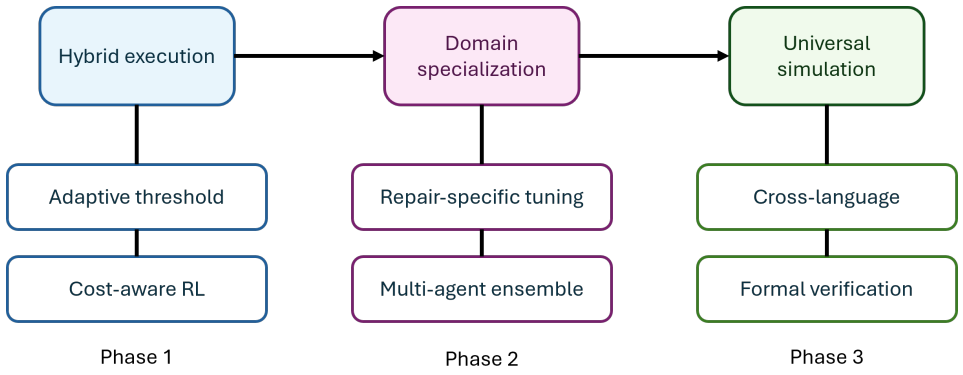

Figure 9: **Research Roadmap.** Progression from current work toward broader code execution simulation.

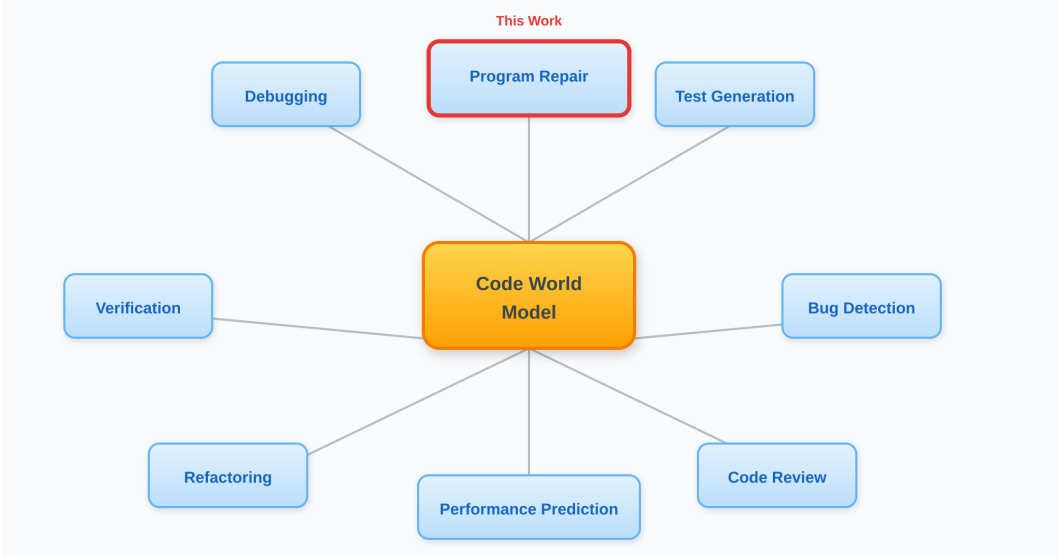

Figure 10: **Applications.** Execution simulation enables diverse software engineering tasks. Program repair (highlighted) provides initial evidence; the same capability may extend to adjacent problems.