DEBUGGING CODE WORLD MODELS

Babak Rahmani*
Tübingen AI Center, University of Tübingen, Microsoft Research
rahmani.b91@gmail.com

[Blog post] | **[CruxEval report]** **[HumanEval report]** **[Composition report]**

ABSTRACT

Code World Models (CWMs) are language models trained to simulate program execution by predicting explicit runtime state after every executed command. This execution-based world modeling enables internal verification within the model, offering an alternative to natural language chain-of-thought reasoning. However, the sources of errors and the nature of CWMs' limitations remain poorly understood. We study CWMs from two complementary perspectives: local semantic execution and long-horizon state tracking. On real-code benchmarks, we identify two dominant failure regimes. First, dense runtime state reveals produce token-intensive execution traces, leading to token-budget exhaustion on programs with long execution histories. Second, failures disproportionately concentrate in string-valued state, which we attribute to limitations of subword tokenization rather than program structure. To study long-horizon behavior, we use a controlled permutation-tracking benchmark that isolates state propagation under action execution. We show that long-horizon degradation is driven primarily by incorrect action generation: when actions are replaced with ground-truth commands, a Transformer-based CWM propagates state accurately over long horizons, despite known limitations of Transformers in long-horizon state tracking. These findings suggest directions for more efficient supervision and state representations in CWMs that are better aligned with program execution and data types.

1 INTRODUCTION

World modeling frameworks recast sequence modeling as learning an explicit state-transition process: given a current state and an action, the model predicts how the environment evolves (Ha & Schmidhuber, 2018; Bruce et al., 2024; Lehrach et al., 2025; Dainese et al., 2024; Tang et al., 2024; Kim et al., 2022). This formulation transforms LLMs from passive predictors into simulators of structured dynamics which could help with supporting multi-step reasoning, planning, and internal verification through rollouts.

Recent work extends this world-modeling paradigm to code (Liu et al., 2023; Ding et al., 2024a;b; Lehrach et al., 2025), where the environment is program execution and the state consists of runtime variables and control-flow state. Code World Models (CWMs) (Copet et al., 2025) instantiate this approach by training on Python execution traces that interleave each executed line of code with a complete snapshot of the runtime state. This yields a dense supervision regime in which all variables are revealed after every operation. CWMs have been shown to deliver substantial gains on software engineering benchmarks (Copet et al., 2025).

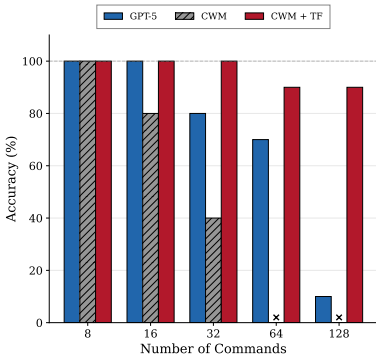

Figure 1: Accuracy on long-horizon state tracking via Code S_5 permutation tracking, where models apply sequences of permutation swaps (8–128 operations). CWM+TF (teacher forcing) maintains high accuracy, while GPT5 and CWM degrade with sequence length. $\times$ indicates zero accuracy.

Despite these gains, it remains unclear why dense action–state prediction is so effective and where its limitations arise. Revealing the full runtime state after every

*This work was conducted while at Microsoft Research.

operation reduces execution to a sequence of locally supervised state updates, effectively mitigating the known long-horizon state-tracking difficulties of Transformer-based models. But how fully does this supervision resolve long-horizon failures in practice? And even under dense reveals, can errors still arise from local semantic execution?

In this work, we study CWMs through these two complementary lenses. First, we evaluate semantic execution, asking whether models correctly apply sequences of deterministic operations across data types. Second, we analyze long-horizon state tracking, asking whether execution state can be maintained over many steps despite compounding autoregressive error and limited context.

Contributions. We begin with real-code benchmarks, CruxEval-O (Gu et al., 2024) and HumanEval (Chen et al., 2021), where we observe failures concentrate in two regimes (section 3.1): **(1)** token-budget exhaustion from long execution traces and **(2)** brittleness in string-valued state. The latter aligns with known representation discontinuities induced by subword tokenization (Xue et al., 2021; Clark et al., 2022; Pagnoni et al., 2025).

To isolate data type effects independent of program structure, we introduce controlled evaluations. Using functional programs with fixed structure and varying data types, we show that CWMs compose reliably for non-string data, while nested string transformations degrade sharply (sections 3.2 and 3.4).

To isolate long-horizon behavior, we study an existing controlled permutation-tracking benchmark (Siems et al., 2026) that stresses state propagation over extended execution histories (section 4). Despite full state reveals at every step, performance degrades with horizon length. We show that this degradation is driven primarily by action hallucination rather than local state-update errors: when actions are corrected, a Transformer-based CWM propagates state accurately for over 128 steps.

Finally, motivated by efficiency and truncation concerns, we discuss the implications of dense state supervision for scalability in code world modeling (section 6). While we do not resolve these trade-offs here, our analysis highlights the need for future work that studies execution under reduced observation. We will open-source our analysis across all datasets in the future.

2 BACKGROUND

Code World Models and execution traces. A world model learns a transition function for a sequential environment: given a state and an action, it predicts the next state (Ha & Schmidhuber, 2018). In code execution, actions correspond to executed statements and the state corresponds to the runtime configuration, including variable bindings and control-flow state.

CWMs instantiate this by representing program execution as an explicit sequence of action–state pairs derived from Python execution traces (Copet et al., 2025). Given a partial trace, the model is trained to predict the next action and the resulting state, effectively acting as a simulator of program execution. For example:

```
# Program
def f(x):
    y = x + 1
    return y

def main():  # << START_OF_TRACE
    z = f(3)
    return z

# Execution trace (action -> state)
Action: def main()                State: {}
Action: z = f(3)                  State: {}
Action: def f(x)                  State: {"x": 3}
Action: y = x + 1                 State: {"x": .., "y": 4}
Action: return y                  State: {"x": .., "y": ..}   -> returns 4
Action: return z                  State: {"z": 4}           -> returns 4
```

At each step, the model observes the executed statement (action) and a complete snapshot of the runtime state, where ".." denotes variables that remain unchanged. CWMs are trained to predict the next action–state pair given a code prefix. Full trace examples are provided in Appendix A.

Table 1: CWM accuracy on code execution benchmarks. Δ denotes absolute accuracy change after intervention.

Benchmark	Samples	Max Tokens	Correct	Incorrect	Truncated	After Intervention	Δ
CruxEval-O	800	8K	85.0%	14.2%	0.8%	**90.4%**	+5.4
HumanEval	723	8K	91.4%	4.5%	4.1%	**92%**	+0.6
Composition Zoo (depth=5, non-string)							
Boolean	10	8K	100%	0%	—	—	—
Bitwise	10	8K	100%	0%	—	—	—
Math	10	8K	100%	0%	—	—	—
Character	10	8K	100%	0%	—	—	—
List	10	8K	100%	0%	—	—	—
Set	10	8K	100%	0%	—	—	—
Dictionary	10	8K	100%	0%	—	—	—
Composition (string)							
depth=2	100	4K	75%	25%	—	**78%**	+3
depth=3	100	4K	58%	42%	—	**63%**	+5
depth=4	100	4K	39%	61%	—	**43%**	+4
depth=5	100	4K	25%	75%	—	**28%**	+3

3 Evaluation on Code Benchmarks

We evaluate CWMs on real code execution benchmarks to identify recurring failure modes and their underlying causes, and then use controlled experiments (section 3.2) to isolate these effects.

We begin with two standard benchmarks that probe complementary aspects of code execution. CruxEval-O (Gu et al., 2024) evaluates end-to-end output prediction for short Python programs, emphasizing direct reasoning from source code to final outputs. HumanEval (Chen et al., 2021) assesses functional correctness via unit tests. In our setting, both benchmarks are treated as execution probes: given a fixed program and input, the model must simulate execution to produce the correct output. Example prompt templates are provided in Appendix A. Under this setup, CWM achieves 85.0% accuracy on CruxEval-O and 91.4% on HumanEval.

Figure 2, top shows the distribution of non-truncation failures (i.e., incorrect predictions) by output data type. String-valued outputs dominate failure cases in both benchmarks. In CruxEval-O, strings account for 73% of failures despite constituting only 46% of outputs (1.58$\times$ overrepresentation), while integers (11%), lists (9%), and other types fail far less frequently. HumanEval shows a similar pattern: string-related failures comprise 44% of failure cases even though strings represent just 17% of outputs (2.58$\times$ overrepresentation). Full type distributions are provided in Appendix D.

To contextualize these errors, fig. 2, bottom presents a taxonomy of failure modes on CruxEval-O, separating semantic prediction errors from trace truncation due to token-budget exhaustion. We next focus on these two failure categories.

3.1 Where Code World Models Fail?

Figure 2, bottom summarizes failure modes on CruxEval-O, grouped by output data type and truncation. A complete catalog of failure cases, including code, inputs, and predictions, is provided in Appendix E. We focus on two dominant failure families.

Failure family 1: trace truncation. Although both benchmarks consist largely of short functions, dense trace supervision can fail when execution induces traces that exceed the token budget. As illustrated in fig. 2, truncation commonly arises from (i) deep but linear iteration chains (e.g., repeated parenthesis depth tracking), (ii) control-flow patterns that generate unbounded execution through in-place mutation during iteration, and (iii) per-element string processing that produces long, token-expensive traces even for moderate input lengths. In these cases, failure is not caused by incorrect local execution but by the cumulative cost of revealing state at every step.

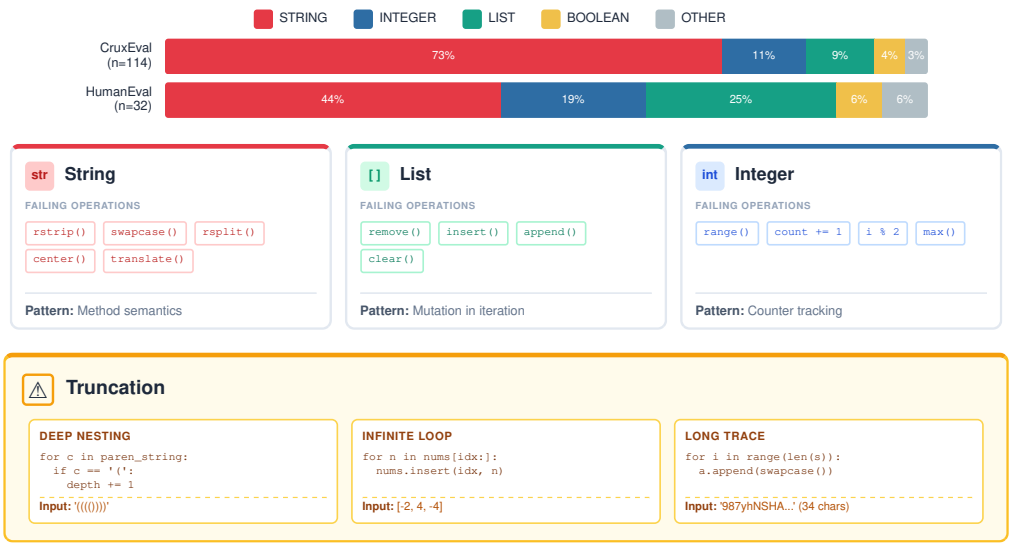

Figure 2: **Top:** Distribution of CWM non-truncation failures by output data type on CruxEval-O and HumanEval, excluding truncation cases. String-valued outputs dominate failures **Bottom:** Failure taxonomy on CruxEval-O. Top row shows examples of data type failures. Bottom row shows truncation failure patterns that cause trace overflow.

Failure family 2: string-valued state. Beyond truncation, failures disproportionately involve string-valued state. As illustrated in fig. 2, these errors are dominated by incorrect handling of common string operations, including method semantics (e.g., `rstrip`, `swapcase`, `rsplit`, `center`) and boundary-sensitive indexing or slicing. The resulting outputs are often syntactically well-formed but semantically incorrect, indicating failures in how string transformations are represented rather than in control flow.

Notably, these errors arise even in short programs with minimal execution depth, and similar string-related failures appear in HumanEval. This raises a central question: do these failures reflect limitations in CWMs' semantic execution, or instability in how string state is represented under tokenization?

Takeaway. On real code benchmarks, CWM failures under dense supervision concentrate in two regimes: *(i)* token-budget limitations from long execution traces, and *(ii)* representation brittleness in string-valued state.

3.2 Functional composition across data types

To isolate the source of string-related failures observed in real-code benchmarks, we introduce a controlled test based on functional composition. Composing deterministic single-argument functions to depth d induces multi-step computation without loops or branching, allowing us to hold program structure fixed while varying only the data type.

Composition Zoo (non-string data types). We first evaluate a multi-domain Composition Zoo consisting of depth-5 compositions across seven non-string categories (boolean, bitwise, math, character, list, set, dictionary), with 10 held-out samples per category. Functions are deterministic and side-effect free, and compositions are generated by randomly nesting functions within each category. At depth 5, CWM achieves 100% accuracy across all categories, demonstrating reliable composition when intermediate state is represented stably. Appendix B.3 lists the function definitions.

Composition (string). We next apply the same compositional scaffold to string transformations using the composition string dataset from (Yuan et al., 2025). The dataset contains deterministic string-manipulation functions, including case alternation, prefix/suffix insertion, joining, slicing, rotation, and simple loop-based transformations. The full function set is listed in Appendix B.3.

In contrast to the non-string results, accuracy degrades sharply with composition depth (Table 1), despite programs being short, deterministic, and purely functional. To factor out per-operation difficulty, we first evaluate atomic calls (depth 1) for 25 string functions (10 samples each) and retain the 15 functions with $\geq$90% atomic accuracy, yielding an average atomic accuracy of 95.3%. Per-function accuracy is reported in Appendix B.2. We then generate 100 random compositions per depth for $d \in \{1, 2, 3, 4, 5\}$, with ground-truth outputs computed by executing the Python code. A representative evaluation prompt is shown in Appendix B.1. Performance drops from 75% at depth 2 to 25% at depth 5, in stark contrast to non-string compositions, which maintain 100% accuracy at depth 5 under identical structure.

These results support the hypothesis that the primary limitation lies in string representation rather than execution structure, motivating the analysis in section 3.4.

> **Takeaway.** Functional composition is not a bottleneck for CWMs. When program structure is held fixed, CWMs compose reliably across non-string data types but degrade sharply on string-valued state, isolating string representation as the source of failure.

3.3 Semantics-Preserving Code Decomposition Interventions

To probe even further whether some CWM failures arise from hidden intermediate state rather than incorrect execution, we apply a simple code decomposition intervention to failing samples across all dataset. The intervention rewrites programs using semantics-preserving transformations that expose intermediate values explicitly in the execution trace, including (i) decomposition of nested expressions into temporary assignments and (ii) decomposition of selected single-character string methods into explicit character-level loops. This yields consistent but limited improvements on the observed failure cases (Table 1, "After Intervention"), indicating that a subset of errors is driven by state visibility. However, the approach is not scalable: string decompositions can inflate trace length and trigger truncation, and decomposition does not address failures caused by semantic errors. Implementation details and examples are deferred to the Appendix F.

3.4 Tokenization discontinuity

The composition experiments show that string-valued state is uniquely brittle under depth, even when program structure is held fixed. This behavior is consistent with tokenization discontinuity, where the same character sequence produces different tokens depending on surrounding context.

Under Byte-Pair Encoding (BPE) Sennrich et al. (2016) tokenization, context-dependent merging causes the same characters to map to different token IDs. As shown in fig. 3, the separator `"-."` tokenizes as a single token (ID 14863) when encoded alone, but within `"a-.-.b"`, BPE produces an entirely different sequence: `[64, 12, 12898, 13, 65]`. Critically, token 14863 never appears in this sequence. Consequently, when evaluating `rsplit`, the model cannot locate the separator at the token level, producing incorrect output.

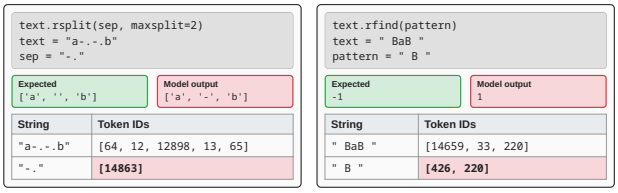

Figure 3: Tokenization discontinuity: **Left**: the separator `"-."` tokenizes as ID 14863 alone, but this token never appears in `"a-.-.b"`'s token sequence, causing `rsplit` to fail. **Right**: the pattern `" B "` tokenizes as `[426, 220]`, but token 426 is absent from `" BaB "`'s tokens, causing `rfind` to hallucinate a match.

Similarly, for `rfind(" B ", " BaB ")`, the pattern tokenizes as `[426, 220]`, but the text yields `[14659, 33, 220]` where token 426 is absent. The model hallucinates a match at position 1 when the correct answer is -1.

This instability explains the sharp error compounding observed in string compositions. Each transformation can substantially perturb the token sequence, making subsequent state updates unreliable even in short, deterministic programs.

4 Long-Horizon State Tracking: The Code S_5 Benchmark

Beyond representation-specific failures such as string brittleness, a central question is whether CWMs can faithfully track execution state over long horizons. This question is orthogonal to local semantic execution and arises even when code semantics are simple, deterministic, and fully observed, as in CWMs with full state reveals at every step.

CWMs are implemented using attention-based architectures. Prior work shows that Transformer-style models struggle with faithful state tracking and automaton simulation as sequence length grows, even when trained successfully on short horizons (Hahn, 2020; Bhattamishra et al., 2020; Delétang et al., 2022; Merrill & Sabharwal, 2023).

To evaluate this limitation in isolation, we adopt a controlled state-tracking benchmark based on permutation groups expressed as executable code, introduced by Siems et al. (2026).

Permutation tracking on S_n. Let S_n denote the symmetric group on n elements. Given a sequence of permutations $(\sigma_1, \ldots, \sigma_N)$, the cumulative state after t steps is

$$\sigma_{\leq t} = \sigma_t \circ \sigma_{t-1} \circ \cdots \circ \sigma_1 \in S_n.$$

Equivalently, initializing from $x_0 = (1, \ldots, n)$, the tracked state is $x_t = \sigma_{\leq t}(x_0)$, and the task is to predict the final state x_N after N operations.

To mirror code execution, we serialize permutations as Python variable assignments over n variables, using full permutations that simultaneously reassign all n variables. An example of S_5 expressed in code over five variables is shown in fig. 4, left.

4.1 Code Execution State Tracking in Code World Models

We evaluate long-horizon state tracking using permutation-tracking traces over S_5. Models must iteratively apply deterministic transitions and predict the final variable assignment after $N \in \{8, 16, 32, 64, 128\}$ steps. This benchmark isolates state propagation over long horizons. We report exact-match accuracy of the final state.

CWM is evaluated in its native execution-trace format, and GPT-5 is evaluated on the same traces as a general-purpose LLM baseline. Full prompt details are provided in Appendix C. Results are shown in fig. 1.

At first glance, CWM accuracy degrades sharply with horizon length, suggesting a failure of state propagation. However, inspection reveals a different dominant failure mode: errors typically arise from generating an incorrect next command (see fig. 4, left), after which all subsequent states are deterministically corrupted.

4.2 Action Hallucination vs. State Propagation

In the baseline setting, long-horizon failures are dominated by action hallucination: once an incorrect command is generated (e.g., an incorrect swap), all subsequent states are necessarily wrong because execution is conditioned on a corrupted history (see fig. 4).

To isolate state propagation from action generation, we perform a **teacher-forcing** evaluation (see fig. 4, right), injecting the ground-truth operation at each step and evaluating only the predicted state. Under teacher forcing, CWM maintains 90% accuracy at 128 steps, whereas baseline accuracy collapses beyond 64 steps (fig. 1).

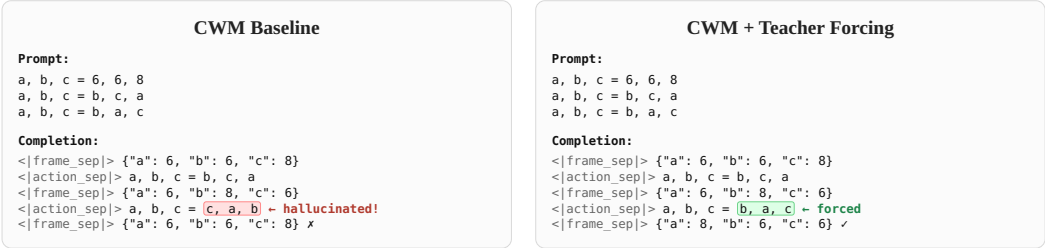

Figure 4: Illustration of action hallucination and teacher forcing in a CWM trace. **Left**: In the baseline setting, an incorrect next command corrupts the execution history and forces all subsequent states to be wrong. **Right**: Under teacher forcing, the ground-truth command is injected at each step and evaluation isolates state prediction.

This demonstrates that, conditioned on correct actions, a Transformer-based CWM can reliably propagate state over long horizons. Crucially, this capability is enabled by dense full-state reveals, which reduce long-horizon tracking to a sequence of locally supervised state updates.

Takeaway. Long-horizon failures in CWMs are dominated by action hallucination rather than state-update errors. When conditioned on correct actions, Transformer-based models can faithfully propagate state for hundreds of steps. Dense full-state supervision is the key mechanism enabling this behavior.

5 Related Work

Trace-supervised execution models. A growing line of work augments code models with execution signals to capture dynamic semantics. CodeExecutor (Liu et al., 2023) uses mutation-based augmentation to generate large-scale programs with line-by-line variable traces. SEMCODER (Ding et al., 2024a) trains on execution monologues that describe control-flow and state evolution in natural language. TRACED (Ding et al., 2024b) combines source code with quantized execution states for improved value/path prediction. At larger scale, CWMs (Copet et al., 2025) train on action–state traces where each executed line is paired with an explicit runtime snapshot, and report strong gains on software engineering benchmarks. We focus on CWMs and study their performance.

When do traces help (and when do they not)? Recent studies question whether simply adding traces reliably improves general code reasoning. Zhang et al. (2025) find traced versions of human-written functions can *hurt* performance, suggesting models may pattern-match on trace format rather than internalize execution. Wang et al. (2025) and Haque et al. (2025) report mixed benefits from trace-augmented fine-tuning across model families, and highlight that traces can be more useful as a test-time tool than as a universal training signal. Armengol-Estapé et al. (2025) show that models can produce long scratchpads with high trace accuracy while yielding limited gains on downstream code generation. These results motivate our focus on diagnosing trace-based execution: we characterize dominant failure modes on real benchmarks and isolate their causes with controlled tests.

State tracking, supervision density, and architectural limits. Our analysis connects trace-based execution to classic state-tracking benchmarks studied through regular languages, automata emulation, and length generalization (Hahn, 2020; Bhattamishra et al., 2020; Delétang et al., 2022; Merrill & Sabharwal, 2023). A recurring finding is that Transformer-based models can fit short-horizon training distributions yet struggle to extrapolate faithful state updates to longer horizons (Liu et al., 2022). Dense trace supervision alters this regime by revealing the full execution state after every operation, effectively reducing long-horizon state tracking to a sequence of locally supervised updates. At the same time, prior work shows that recurrent linear and state-space architectures can maintain stable state representations under sparse observation and extended horizons (Orvieto et al., 2023; Grazzi et al., 2025; Siems et al., 2025; Schöne et al., 2025), highlighting an architectural dependence of state tracking on both supervision density and model class. Our results on code s_5 benchmark suggest that dense state supervision in CWMs plays a role analogous to the inductive

bias provided by linear recurrent architectures, compensating for limitations in Transformer-based state propagation.

6 Discussion and Conclusion

Our results highlight the limitations of CWMs. In the short-horizon regime, dense action–state supervision makes execution largely a sequence of locally supervised state updates, enabling strong compositional performance across most data types (section 3.2). However failures concentrate in string-valued state, consistent with tokenization-driven representation brittleness (sections 3.1 and 3.4). In the long-horizon regime, performance degradation is dominated by action hallucination: once an incorrect command is generated, the remainder of the trace is deterministically corrupted; when we correct actions via teacher forcing, CWMs can propagate state accurately (section 4). Across both regimes, the approach is inherently token-intensive, and long traces lead to truncation, making efficiency a central bottleneck (section 3.1).

The efficiency bottleneck, introduced by dense execution traces, also points to an architectural consideration. When state information is revealed less frequently, successful execution requires models to propagate state over extended horizons with limited observation. Recent concurrent work (Siems et al., 2026) shows that linear recurrent architectures can maintain stable state tracking under sparse reveals, whereas Transformer-based models degrade more rapidly in this regime. This suggests that dense supervision may compensate for architectural limitations in attention-based simulators, and that reducing token cost fundamentally alters the learning regime for code execution.

Looking ahead, building world-modeling agents that execute and verify code internally, rather than relying on external tools, will require rethinking both supervision and architecture to improve efficiency. Moving beyond dense execution traces calls for models that can propagate state under sparse observation, where linear recurrent (Yang et al., 2025; Siems et al., 2025) or state-space architectures (Gu & Dao, 2024) are natural candidates. At the same time faithful execution will also require revisiting text representation itself, for example through byte-level or tokenizer-free approaches that avoid brittle subword boundaries and support stable character-level computation (Choe et al., 2019; Xue et al., 2021; Clark et al., 2022; Pagnoni et al., 2025).

References

Jordi Armengol-Estapé, Quentin Carbonneaux, Tianjun Zhang, Aram H Markosyan, Volker Seeker, Chris Cummins, Melanie Kambadur, Michael FP O'Boyle, Sida Wang, Gabriel Synnaeve, et al. What i cannot execute, i do not understand: Training and evaluating llms on program execution traces. *arXiv preprint arXiv:2503.05703*, 2025.

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. On the ability and limitations of transformers to recognize formal languages. *arXiv preprint arXiv:2009.11264*, 2020.

Jake Bruce, Michael Dennis, Ashley Edwards, Jack Parker-Holder, Yuge Shi, Edward Hughes, Matthew Lai, Aditi Mavalankar, Richie Steigerwald, et al. Genie: Generative interactive environments. *arXiv preprint arXiv:2402.15391*, 2024.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Dokook Choe, Rami Al-Rfou, Mandy Guo, Heeyoung Lee, and Noah Constant. Bridging the gap for tokenizer-free language models. *arXiv preprint arXiv:1908.10322*, 2019.

Jonathan H. Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. Canine: Pre-training an efficient tokenization-free encoder for language representation. In *Transactions of the Association for Computational Linguistics*, 2022.

Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, et al. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.

Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. Generating code world models with large language models guided by monte carlo tree search. *Advances in Neural Information Processing Systems*, 37:60429–60474, 2024.

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, et al. Neural networks and the chomsky hierarchy. *arXiv preprint arXiv:2207.02098*, 2022.

Yangruibo Ding, Jinjun Peng, Marcus Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. Semcoder: Training code language models with comprehensive semantics reasoning. *Advances in Neural Information Processing Systems*, 37:60275–60308, 2024a.

Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail Kaiser, Wei Le, and Baishakhi Ray. Traced: Execution-aware pre-training for source code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pp. 1–12, 2024b.

Riccardo Grazzi, Julien Siems, Arber Zela, Jörg KH Franke, Frank Hutter, and Massimiliano Pontil. Unlocking State-Tracking in Linear RNNs Through Negative Eigenvalues. In *The Thirteenth International Conference on Learning Representations*, 2025.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *First Conference on Language Modeling*, 2024.

Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. Cruxeval: A benchmark for code reasoning, understanding and execution. *arXiv preprint arXiv:2401.03065*, 2024.

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

Michael Hahn. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171, 2020.

Mirazul Haque, Petr Babkin, Farima Farmahinifarahani, and Manuela Veloso. Towards effectively leveraging execution traces for program repair with code llms. In *Proceedings of the 4th International Workshop on Knowledge-Augmented Methods for Natural Language Processing*, pp. 160–179, 2025.

Minsoo Kim, Yeonjoon Jung, Dohyeon Lee, and Seung-won Hwang. Plm-based world models for text-based games. In *EMNLP*, pp. 1324–1341, 2022.

Wolfgang Lehrach, Daniel Hennes, Miguel Lazaro-Gredilla, Xinghua Lou, Carter Wendelken, Zun Li, Antoine Dedieu, Jordi Grau-Moya, Marc Lanctot, Atil Iscen, et al. Code world models for general game playing. *arXiv preprint arXiv:2510.04542*, 2025.

Bingbin Liu, Jordan T Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Transformers learn shortcuts to automata. *arXiv preprint arXiv:2210.10749*, 2022.

Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. Code execution with pre-trained language models. In *Findings of the Association for Computational Linguistics: ACL 2023*, pp. 4984–4999, 2023.

William Merrill and Ashish Sabharwal. The parallelism tradeoff: Limitations of log-precision transformers. *Transactions of the Association for Computational Linguistics*, 11:531–545, 2023.

Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. Resurrecting recurrent neural networks for long sequences. In *ICML*, 2023.

Artidoro Pagnoni et al. Byte latent transformer: Patches scale better than tokens. In *Proceedings of ACL*, 2025.

Mark Schöne, Babak Rahmani, Heiner Kremer, Fabian Falck, Hitesh Ballani, and Jannes Gladrow. Implicit Language Models are RNNs: Balancing Parallelization and Expressivity. In *Forty-second International Conference on Machine Learning*, 2025.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. In *Proceedings of the 54th annual meeting of the association for computational linguistics (volume 1: long papers)*, pp. 1715–1725, 2016.

Julien Siems, Timur Carstensen, Arber Zela, Frank Hutter, Massimiliano Pontil, and Riccardo Grazzi. DeltaProduct: Improving State-Tracking in Linear RNNs via Householder Products. *arXiv preprint arXiv:2502.10297*, 2025.

Julien Siems, Riccardo Grazzi, Kirill Kalinin, Hitesh Ballani, and Babak Rahmani. Learning state-tracking from code: REPL traces and probabilistic automata. In *Workshop on Latent & Implicit Thinking – Going Beyond CoT Reasoning*, 2026. URL `https://openreview.net/forum?id=FqvePVmsKJ`.

Hao Tang, Darren Key, and Kevin Ellis. Worldcoder: A model-based llm agent for building world models by writing code and interacting with the environment. In *NeurIPS*, volume 37, pp. 70148–70212, 2024.

Jian Wang, Xiaofei Xie, Qiang Hu, Shangqing Liu, and Yi Li. Do code semantics help? a comprehensive study on execution trace-based information for code large language models. *arXiv preprint arXiv:2509.11686*, 2025.

Linting Xue, Aditya Barua, Noah Constant, Adam Roberts, Mihir Kale, Rami Al-Rfou, and Aditya Siddhant. Byt5: Towards a token-free future with pre-trained byte-to-byte models. *arXiv preprint arXiv:2105.13626*, 2021.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. In *The Thirteenth International Conference on Learning Representations*, 2025.

Lifan Yuan, Weize Chen, Yuchen Zhang, Ganqu Cui, Hanbin Wang, Ziming You, Ning Ding, Zhiyuan Liu, Maosong Sun, and Hao Peng. From f(x) and g(x) to f(g(x)): Llms learn new skills in rl by composing old ones. *arXiv preprint arXiv:2509.25123*, 2025.

David W Zhang, Micha"el Defferrard, Corrado Rainone, and Roland Memisevic. Grounding code understanding in step-by-step execution. *arXiv preprint*, 2025.

APPENDIX OVERVIEW

- **Appendix A: HumanEval and CruxEval: Prompt Examples**
- **Appendix B: Compositionality: Additional Details**
- **Appendix C: Code S_5 Long-Horizon State-Tracking: Experiment Setup and Prompts (GPT-5 and CWM)**
- **Appendix D: Data Type Distribution Analysis**
- **Appendix E: Failure Catalog by Data Type**
- **Appendix F: Code Decomposition Interventions**

A HUMANEVAL AND CRUXEVAL: PROMPT EXAMPLES

We include representative CWM prompt templates used for trace-based evaluation on HumanEval and CruxEval.

A.1 HUMANEVAL (OUTPUT PREDICTION WITH TRACE)

CWM Prompt (HumanEval)

```
Given a python code function and an assert statement containing a specific input, provide the assertion
      with the exact literal output that the function returns with that input. Do not include any
      mathematical expressions or function calls -- only the final literal value. Your response should be
      solely the assertion, enclosed within [ANSWER] and [/ANSWER] tags.

You are a computational world model and can predict the program execution.
Your execution trace prediction format MUST follow this structure:
1. The execution trace prediction starts with the <|trace_context_start|> token and ends with a final <|
      frame_sep|> token.
2. For each code execution step:
  - Begin with <|frame_sep|> followed by the event token which can be <|call_sep|>, <|line_sep|>, <|
        return_sep|> or <|exception_sep|>.
  - After <|call_sep|> or <|line_sep|> put the local variable states as dictionary in JSON format followed
        by the <|action_sep|> token and the current source code line.
  - After <|return_sep|>, <|exception_sep|> directly put the <|action_sep|> token and the current source
        code line followed by an <|arg_sep|> token and the return or exception arguments.
3. Provide the full assertion with the correct output that you obtained after <|return_sep|> in [ANSWER]
      and [/ANSWER] tags

Here is an example of how you would predict the output of the program using your trace prediction
      capability:

Python function:
def f(a,b):
   y = a
   for i in range(b):
      y += y * i
   return y
assert f(1,3) == ??

Let's verify this by putting the code into a trace context and call the function in the main() function and
       then trace the execution of the main function.
We indicate the entry point of the execution trace with a # << START_OF_TRACE marker.

def f(a,b):
   y = a
   for i in range(b):
      y += y * i
   return y

def main(): # << START_OF_TRACE
   return f(1,3)

<|frame_sep|><|call_sep|>{}<|action_sep|>def main(): # << START_OF_TRACE
<|frame_sep|><|line_sep|>{}<|action_sep|> return f(1,3)
<|frame_sep|><|call_sep|>{"a": "1", "b": "3"}<|action_sep|>def f(a,b):
<|frame_sep|><|line_sep|>{"a": "..", "b": ".."}<|action_sep|> y = a
<|frame_sep|><|line_sep|>{"a": "..", "b": "..", "y": "1"}<|action_sep|> for i in range(b):
<|frame_sep|><|line_sep|>{"a": "..", "b": "..", "y": "..", "i": "0"}<|action_sep|> y += y * i
<|frame_sep|><|line_sep|>{"a": "..", "b": "..", "y": "..", "i": ".."}<|action_sep|> for i in range(b):
<|frame_sep|><|line_sep|>{"a": "..", "b": "..", "y": "..", "i": "1"}<|action_sep|> y += y * i
<|frame_sep|><|line_sep|>{"a": "..", "b": "..", "y": "2", "i": ".."}<|action_sep|> for i in range(b):
<|frame_sep|><|line_sep|>{"a": "..", "b": "..", "y": "..", "i": "2"}<|action_sep|> y += y * i
<|frame_sep|><|line_sep|>{"a": "..", "b": "..", "y": "6", "i": ".."}<|action_sep|> for i in range(b):
<|frame_sep|><|line_sep|>{"a": "..", "b": "..", "y": "..", "i": ".."}<|action_sep|> return y
<|frame_sep|><|return_sep|><|action_sep|> return y
<|arg_sep|>"6"<|frame_sep|><|return_sep|><|action_sep|> return f(1,3)
<|arg_sep|>"6"<|frame_sep|>

Now let us analyze the trace. The return argument of the function call f(1,3) in the main() function is "6"
       in JSON format, so the return value is 6.

[ANSWER]
assert f(1,3) == 6
[/ANSWER]

Now solve this problem:

Python function:
<FUNCTION_CODE>
assert <FUNCTION_NAME>(<INPUT_ARGS>) == ??

Let's verify this by putting the code into a trace context and call the function in the main() function and
       then trace the execution of the main function.
We indicate the entry point of the execution trace with a # << START_OF_TRACE marker.

<FUNCTION_CODE>
```

```
def main(): # << START_OF_TRACE
   return <FUNCTION_NAME>(<INPUT_ARGS>)
```

A.2 CRUXEVAL (OUTPUT PREDICTION WITH TRACE)

CWM Prompt (CruxEval-O)

```
Given a python code function and an assert statement containing a specific input, provide the assertion
      with the exact literal output that the function returns with that input. Do not include any
      mathematical expressions or function calls -- only the final literal value. Your response should be
      solely the assertion, enclosed within [ANSWER] and [/ANSWER] tags.

You are a computational world model and can predict the program execution.
Your execution trace prediction format MUST follow this structure:
1. The execution trace prediction starts with the <|trace_context_start|> token and ends with a final <|
      frame_sep|> token.
2. For each code execution step:
  - Begin with <|frame_sep|> followed by the event token which can be <|call_sep|>, <|line_sep|>, <|
        return_sep|> or <|exception_sep|>.
  - After <|call_sep|> or <|line_sep|> put the local variable states as dictionary in JSON format followed
        by the <|action_sep|> token and the current source code line.
  - After <|return_sep|>, <|exception_sep|> directly put the <|action_sep|> token and the current source
        code line followed by an <|arg_sep|> token and the return or exception arguments.
3. Provide the full assertion with the correct output that you obtained after <|return_sep|> in [ANSWER]
      and [/ANSWER] tags

Now solve this problem:

Python function:
<FUNCTION_CODE>
assert f(<INPUT>) == ??

Let's verify this by putting the code into a trace context and call the function in the main() function and
       then trace the execution of the main function.
We indicate the entry point of the execution trace with a # << START_OF_TRACE marker.

<FUNCTION_CODE>

def main(): # << START_OF_TRACE
   return f(<INPUT>)
```

B COMPOSITIONALITY: ADDITIONAL DETAILS

This appendix supplements the compositionality analysis in section 3.2.

B.1 PROMPT (DEPTH 3)

For reproducibility, we include an example prompt used to evaluate compositionality at depth 3.

CWM Prompt (Composition)

```
Given a python code function and an assert statement containing a specific input, provide the assertion
      with the exact literal output that the function returns with that input. Do not include any
      mathematical expressions or function calls -- only the final literal value. Your response should be
      solely the assertion, enclosed within [ANSWER] and [/ANSWER] tags.

You are a computational world model and can predict the program execution.
Your execution trace prediction format MUST follow this structure:
1. The execution trace prediction starts with the <|trace_context_start|> token and ends with a final <|
      frame_sep|> token.
2. For each code execution step:
  - Begin with <|frame_sep|> followed by the event token which can be <|call_sep|>, <|line_sep|>, <|
        return_sep|> or <|exception_sep|>.
  - After <|call_sep|> or <|line_sep|> put the local variable states as dictionary in JSON format followed
        by the <|action_sep|> token and the current source code line.
  - After <|return_sep|>, <|exception_sep|> directly put the <|action_sep|> token and the current source
        code line followed by an <|arg_sep|> token and the return or exception arguments.
3. Provide the full assertion with the correct output that you obtained after <|return_sep|> in [ANSWER]
      and [/ANSWER] tags

Here is an example of how you would predict the output of the program using your trace prediction
      capability:

Python function:
def f(a,b):
   y = a
```

```
   for i in range(b):
      y += y * i
   return y
assert f(1,3) == ??

Let's verify this by putting the code into a trace context and call the function in the main() function and
       then trace the execution of the main function.
We indicate the entry point of the execution trace with a # << START_OF_TRACE marker.

def f(a,b):
   y = a
   for i in range(b):
      y += y * i
   return y

def main(): # << START_OF_TRACE
   return f(1,3)

<|frame_sep|><|call_sep|>{}<|action_sep|>def main(): # << START_OF_TRACE
<|frame_sep|><|line_sep|>{}<|action_sep|> return f(1,3)
<|frame_sep|><|call_sep|>{"a": "1", "b": "3"}<|action_sep|>def f(a,b):
<|frame_sep|><|line_sep|>{"a": "..", "b": ".."}<|action_sep|> y = a
<|frame_sep|><|line_sep|>{"a": "..", "b": "..", "y": "1"}<|action_sep|> for i in range(b):
<|frame_sep|><|line_sep|>{"a": "..", "b": "..", "y": "..", "i": "0"}<|action_sep|> y += y * i
<|frame_sep|><|line_sep|>{"a": "..", "b": "..", "y": "..", "i": ".."}<|action_sep|> for i in range(b):
<|frame_sep|><|line_sep|>{"a": "..", "b": "..", "y": "..", "i": "1"}<|action_sep|> y += y * i
<|frame_sep|><|line_sep|>{"a": "..", "b": "..", "y": "2", "i": ".."}<|action_sep|> for i in range(b):
<|frame_sep|><|line_sep|>{"a": "..", "b": "..", "y": "..", "i": "2"}<|action_sep|> y += y * i
<|frame_sep|><|line_sep|>{"a": "..", "b": "..", "y": "6", "i": ".."}<|action_sep|> for i in range(b):
<|frame_sep|><|line_sep|>{"a": "..", "b": "..", "y": "..", "i": ".."}<|action_sep|> return y
<|frame_sep|><|return_sep|><|action_sep|> return y
<|arg_sep|>"6"<|frame_sep|><|return_sep|><|action_sep|> return f(1,3)
<|arg_sep|>"6"<|frame_sep|>

Now let us analyze the trace. The return argument of the function call f(1,3) in the main() function is "6"
       in JSON format, so the return value is 6.

[ANSWER]
assert f(1,3) == 6
[/ANSWER]

Now solve this problem:

Python functions:
def func_1(s, pre):
   return pre + s

def func_12(s):
   return ''.join(ch.lower() if i % 2 == 0 else ch.upper() for i, ch in enumerate(s))

def func_14(s, sep):
   return sep.join(s)


assert main_solution("qgjucy") == ??

Let's verify this by putting the code into a trace context and call the function in the main() function and
       then trace the execution of the main function.
We indicate the entry point of the execution trace with a # << START_OF_TRACE marker.

def func_1(s, pre):
   return pre + s

def func_12(s):
   return ''.join(ch.lower() if i % 2 == 0 else ch.upper() for i, ch in enumerate(s))

def func_14(s, sep):
   return sep.join(s)


def main_solution(x):
   return func_12(func_12(func_14(x, '-')))

def main(): # << START_OF_TRACE
   return main_solution("qgjucy")
```

B.2 Atomic Sring function accuracy distribution

Before measuring composition depth effects, we evaluate all 25 atomic string functions to separate intrinsic operation difficulty from compositional degradation. Figure 5 summarizes the per-function atomic accuracies used to select the high-accuracy subset for the depth experiment.

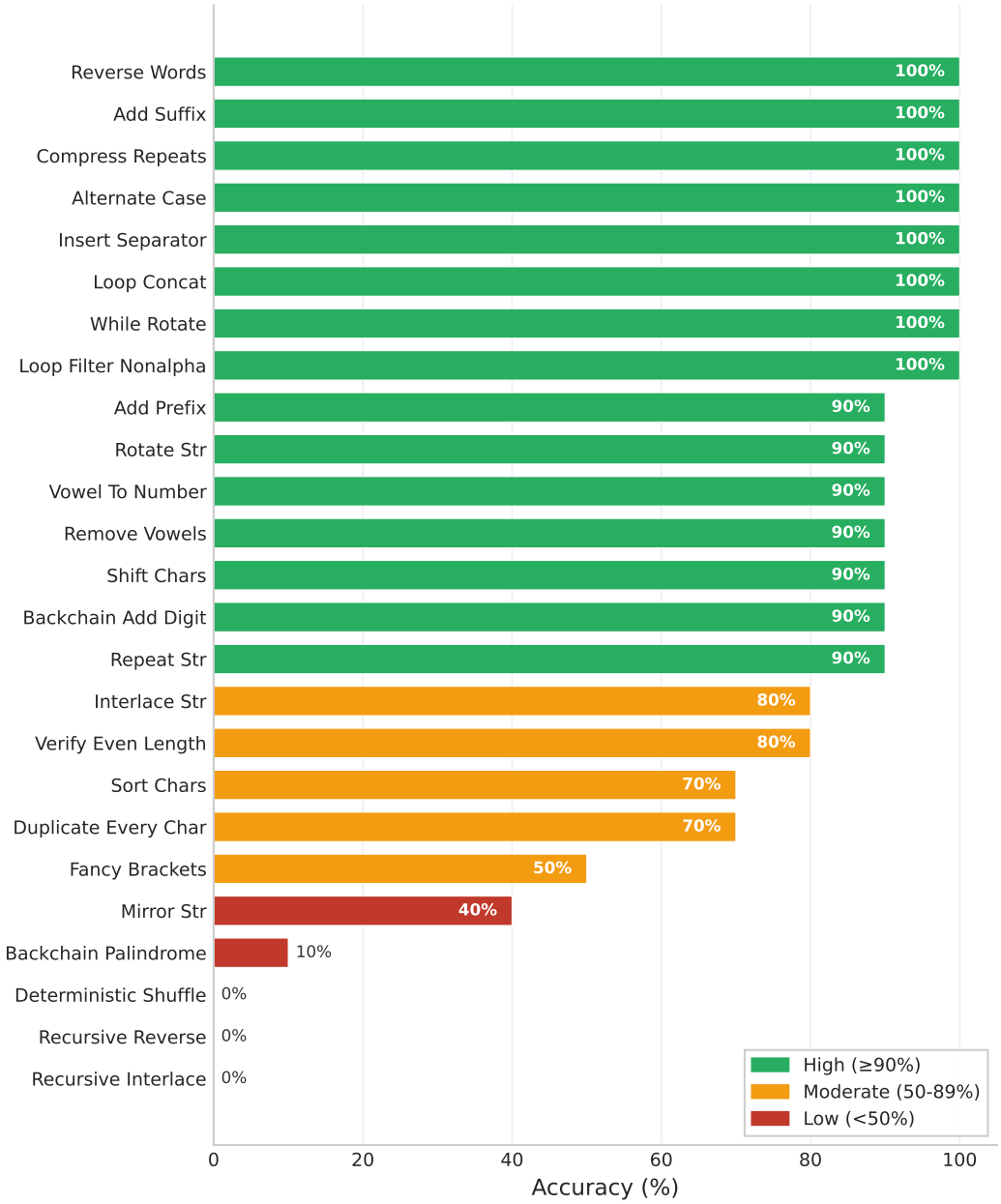

Figure 5: Atomic accuracy report for the 25 string-manipulation functions used in the compositionality study.

B.3 Composition Zoo function definitions by category

For the multi-domain compositionality study ("Composition Zoo"), each category uses five deterministic, single-argument functions that are composed to depth 5. We list the exact function definitions used to generate the evaluation prompts.

Composition Zoo Functions (Boolean)

```
def bool_and_true(x):
    return x and True
```

```python
def bool_or_false(x):
    return x or False

def bool_not(x):
    return not x

def bool_identity(x):
    return x

def bool_xor_true(x):
    return x != True
```

Composition Zoo Functions (Bitwise)

```python
def bit_and_15(x):
    return x & 15

def bit_or_8(x):
    return x | 8

def bit_xor_7(x):
    return x ^ 7

def bit_shift_left_1(x):
    return x << 1

def bit_shift_right_1(x):
    return x >> 1
```

Composition Zoo Functions (Math)

```python
def math_abs(x):
    return abs(x)

def math_negate(x):
    return -x

def math_double(x):
    return x * 2

def math_halve(x):
    return x // 2

def math_mod_10(x):
    return x % 10
```

Composition Zoo Functions (Character)

```python
def char_next(c):
    return chr((ord(c) - ord('a') + 1) % 26 + ord('a'))

def char_prev(c):
    return chr((ord(c) - ord('a') - 1) % 26 + ord('a'))

def char_shift_3(c):
    return chr((ord(c) - ord('a') + 3) % 26 + ord('a'))

def char_shift_5(c):
    return chr((ord(c) - ord('a') + 5) % 26 + ord('a'))

def char_identity(c):
    return c
```

Composition Zoo Functions (List)

```python
def list_append_0(lst):
    return lst + [0]

def list_prepend_1(lst):
    return [1] + lst
```

```
def list_reverse(lst):
   return lst[::-1]

def list_drop_first(lst):
   return lst[1:] if len(lst) > 1 else lst

def list_drop_last(lst):
   return lst[:-1] if len(lst) > 1 else lst
```

Composition Zoo Functions (Set)

```
def set_add_1(s):
   return s | {1}

def set_add_2(s):
   return s | {2}

def set_remove_1(s):
   return s - {1}

def set_remove_2(s):
   return s - {2}

def set_intersect_123(s):
   return s & {1, 2, 3}
```

Composition Zoo Functions (Dictionary)

```
def dict_set_a_1(d):
   return {**d, 'a': 1}

def dict_set_b_2(d):
   return {**d, 'b': 2}

def dict_remove_a(d):
   return {k: v for k, v in d.items() if k != 'a'}

def dict_remove_b(d):
   return {k: v for k, v in d.items() if k != 'b'}

def dict_inc_a(d):
   return {**d, 'a': d.get('a', 0) + 1}
```

Composition Zoo Functions (String)

```
def reverse_words(s):
   words = s.split()
   return ' '.join(reversed(words))

def add_suffix(s, suf):
   return s + suf

def compress_repeats(s):
   if not s:
      return s
   result = [s[0]]
   for ch in s[1:]:
      if ch != result[-1]:
         result.append(ch)
   return ''.join(result)

def alternate_case(s):
   return ''.join(ch.lower() if i % 2 == 0 else ch.upper() for i, ch in enumerate(s))

def insert_separator(s, sep):
   return sep.join(s)

def loop_concat(s, n):
   result = ""
   for _ in range(n):
      result += s
   return result

def while_rotate(s, n):
   count = 0
   while count < n and s:
      s = s[1:] + s[0]
```

```
        count += 1
    return s

def loop_filter_nonalpha(s):
    result = ""
    for ch in s:
        if ch.isalpha():
            result += ch
    return result

def add_prefix(s, pre):
    return pre + s

def rotate_str(s, n):
    if not s:
        return s
    n = n % len(s)
    return s[n:] + s[:n]

def vowel_to_number(s):
    mapping = {
        'a': '1', 'e': '2', 'i': '3', 'o': '4', 'u': '5',
        'A': '1', 'E': '2', 'I': '3', 'O': '4', 'U': '5'
    }
    return ''.join(mapping.get(ch, ch) for ch in s)

def remove_vowels(s):
    vowels = 'aeiouAEIOU'
    return ''.join(ch for ch in s if ch not in vowels)

def shift_chars(s, shift):
    def shift_char(ch):
        if 'a' <= ch <= 'z':
            return chr((ord(ch) - ord('a') + shift) % 26 + ord('a'))
        elif 'A' <= ch <= 'Z':
            return chr((ord(ch) - ord('A') + shift) % 26 + ord('A'))
        return ch

    return ''.join(shift_char(ch) for ch in s)

def backchain_add_digit(s, depth):
    def has_digit(t):
        return any(ch.isdigit() for ch in t)

    transformations = [
        lambda t: t + "1",
        lambda t: "2" + t,
        lambda t: t.replace("a", "3"),
        lambda t: t[::-1],
    ]

    def helper(t, d):
        if has_digit(t):
            return t
        if d == 0:
            return None
        for trans in transformations:
            new_t = trans(t)
            res = helper(new_t, d - 1)
            if res is not None:
                return res
        return None

    result = helper(s, depth)
    return result if result is not None else s

def repeat_str(s, n):
    return s * n
```

C CODE S_5 LONG-HORIZON STATE-TRACKING: EXPERIMENT SETUP AND PROMPTS (GPT-5 AND CWM)

This appendix documents the evaluation setup and the prompts used for the controlled S_5 REPL-trace benchmark (fig. 1 and fig. 4).

Task. Each example initializes five variables (a, b, c, d, e) with integer values and applies N swap operations implemented as Python simultaneous assignment. We evaluate lengths $N \in \{8, 16, 32, 64, 128\}$. Accuracy is exact match of the final variable assignment.

Models and interfaces. GPT-5 is evaluated via a structured chat prompt and is required to emit only the final assignment. CWM is evaluated in its native trace format and generates an execution trace with explicit JSON states at each step; we extract the last state frame for scoring.

C.1 GPT-5 PROMPT

GPT-5 Prompt (S5)

```
### System prompt
You are a Python code execution tracer. Your task is to trace through Python code that performs variable
      assignments and swaps, then determine the final values of ALL variables.

## Task Description
Given a Python function that:
1. Initializes 5 variables (a, b, c, d, e) with integer values
2. Performs a series of simultaneous variable swaps (e.g., a, b, c, d, e = c, e, b, a, d)

You must trace through all the operations step by step and provide the final values of ALL five variables.

## Example
Code:
def execute_repl_trace():
   a = 1
   b = 2
   c = 3
   d = 4
   e = 5
   a, b, c, d, e = c, e, b, a, d
   a, b, c, d, e = e, b, c, d, a

def main():
   execute_repl_trace()

Step-by-step trace:
1. Initial: a=1, b=2, c=3, d=4, e=5
2. After a, b, c, d, e = c, e, b, a, d: a=3, b=5, c=2, d=1, e=4
3. After a, b, c, d, e = e, b, c, d, a: a=4, b=5, c=2, d=1, e=3

Answer: a=4,b=5,c=2,d=1,e=3

## Instructions
- Trace through each assignment carefully
- Remember that tuple unpacking in Python happens simultaneously (all right-hand values are evaluated
      before any assignment)
- Provide the final values of ALL variables in the format: a=X,b=X,c=X,d=X,e=X
- Do not include any explanation, just the comma-separated values

### User prompt (example with 8 swap operations)
Trace through the following Python code and provide the final values of ALL variables.

def execute_repl_trace():
   """Execute the REPL trace operations."""
   a = 8
   b = 4
   c = 7
   d = 8
   e = 7
   a, b, c, d, e = c, e, b, a, d
   a, b, c, d, e = e, b, c, d, a
   a, b, c, d, e = b, e, a, c, d
   a, b, c, d, e = a, b, e, d, c
   a, b, c, d, e = b, c, e, a, d
   a, b, c, d, e = e, a, c, b, d
   a, b, c, d, e = a, e, c, b, d
   a, b, c, d, e = b, d, e, c, a

def main():
   execute_repl_trace()

What are the final values of all variables? Provide in the format: a=X,b=X,c=X,d=X,e=X
```

C.2 CWM PROMPT (NATIVE TRACE FORMAT)

CWM Prompt (S5)

```
<|begin_of_text|><|trace_context_start|>
def execute_repl_trace():
   """Execute the REPL trace operations."""
   a = 8
   b = 4
   c = 7
   d = 8
   e = 7
   a, b, c, d, e = c, e, b, a, d
   a, b, c, d, e = e, b, c, d, a
   a, b, c, d, e = b, e, a, c, d
   a, b, c, d, e = a, b, e, d, c
   a, b, c, d, e = b, c, e, a, d
   a, b, c, d, e = e, a, c, b, d
   a, b, c, d, e = a, e, c, b, d
   a, b, c, d, e = b, d, e, c, a
   print(f"c = {c}")

def main(): # << START_OF_TRACE
   execute_repl_trace()
<|frame_sep|>
```

D DATA TYPE DISTRIBUTION ANALYSIS

Here we provide the full data type distribution for both CruxEval-Output and HumanEval, comparing the distribution in the benchmark to the distribution among wrong answer failures (excluding truncation cases).

Table 2: CruxEval-O: Data type distribution in benchmark vs. wrong answer failures (n=800 samples, 114 wrong answer failures)

Type	**Benchmark %**	**Failure %**	**Ratio**
str	46.4%	72.8%	1.57×
list	24.6%	8.8%	0.36×
int	12.1%	11.4%	0.94×
dict	8.4%	0.0%	—
bool	6.1%	3.5%	0.57×
tuple	2.0%	2.6%	1.32×
bytes	0.2%	0.9%	3.51×
float	0.1%	0.0%	—

Table 3: HumanEval: Data type distribution in benchmark vs. wrong answer failures (n=723 samples, 32 wrong answer failures)

Type	**Benchmark %**	**Failure %**	**Ratio**
str	17.2%	44.0%	2.56×
list	24.6%	25.0%	1.02×
int	31.5%	18.8%	0.60×
bool	19.2%	6.2%	0.32×
float	1.4%	6.2%	4.52×
tuple	4.7%	0.0%	—
dict	0.7%	0.0%	—
NoneType	0.7%	0.0%	—

The "Ratio" column shows the overrepresentation factor: values > 1 indicate the data type appears more frequently in wrong answer failures than expected from its benchmark frequency. In CruxEval, strings are 1.57× overrepresented in wrong answer failures (72.8% of failures vs 46.4% of benchmark). In HumanEval, string-related failures are 2.56× overrepresented (44% of failures vs 17.2% of benchmark).

E FAILURE CATALOG BY DATA TYPE

This appendix provides a detailed catalog of CruxEval-O failure cases, organized by output data type. Each entry shows the function code, input, expected output, and CWM's incorrect prediction.

E.1 STRING FAILURES (71% OF CRUXEVAL-O FAILURES)

String operations dominate failures due to method semantics errors and tokenization brittleness.

E.1.1 METHOD SEMANTICS ERRORS

Sample	Code Snippet	Expected	Predicted
s_113 swapcase	`if count%2==0: a.append(line[i].swapcase())`	'987YhnShAShD...'	'987YhNShAshD...'
s_114 rsplit	`text.rsplit(sep, maxsplit=2)`	['a', '', 'b']	['a', '-', 'b']
s_136 center	`line.center(width)`	' bc '	' bc '
s_23 rstrip	`text.rstrip(chars)`	'...XQuery 2.'	'...XQuery 2.2'
s_239 lstrip/rstrip	`text.lstrip(froms) text.rstrip(froms)`	'1co'	't 1cos'
s_168 translate	`text.translate(key)`	'spaib'	'spai'
s_211 rindex/index	`if s.rindex(c) != s.index(c): count+=1`	10	11

Table 4: String method semantics errors. Model misunderstands Python built-in method behavior.

E.1.2 INDEX AND SLICE ERRORS

Sample	Code Snippet	Expected	Predicted
s_198 reverse+strip	`text[::-1].strip(chars)[::-1]`	'tcmfsm'	'tcmfs'
s_201 reverse digits	`chars[::-1]` after `isdigit()` filter	'641524'	'61452'
s_218 count+reverse	`((string+sep) * cnt)[::-1]`	'bacfbacfcbaac..."	'cfbafcabcfcaab...'
s_220 format+slice	`text[:m],text[n:]` in format	'bagfedcacbagfedc"	'cbagfeddaacbafedc'
s_237 partition	`text.partition(char)` with slice	'uuzlwaqiaj'	'zlwaqiaj'

Table 5: String indexing and slicing errors involving reverse operations and complex slice expressions.

E.2 LIST FAILURES (12% OF CRUXEVAL-O FAILURES)

List failures primarily involve mutation during iteration.

Sample	Code Snippet	Expected	Predicted
s_112 remove in loop	for letter in ls: if not letter.istitle(): ls.remove(letter)	'XYZLtRRdn...'	'XYZLRergH...'
s_149 popitem loop	dict.fromkeys(...).popitem()[0] in loop	'2,4,2,0,'	'2,,,,,3,,,,,1...'
s_150 insert loop	for n in numbers[index:]: numbers.insert(index, n)	[-2, 4, -4]	null (truncated)

Table 6: List mutation errors. Modifying a list while iterating causes tracking failures or infinite loops.

E.3 Integer Failures (8% of CruxEval-O failures)

Integer failures involve counter tracking and range calculation errors.

Sample	Code Snippet	Expected	Predicted
s_118 range loop	for i in range(num_applies): extra_chars += chars	'zbzquiuqnmfkx"	zbzquiuqnfkx'
s_163 dynamic range	range(size-len(text))	'w))))))))))))"	w))))))'
s_244 rjust count	text.rjust(len(text) + count*2)	' '	' '

Table 7: Integer counter and range errors. Model loses track of iteration counts or range bounds.

E.4 Dictionary Failures

Sample	Code Snippet	Expected	Predicted
s_130 items+format swap	items = list(m.items()) loop + .format(*m.keys(), **m)	'h=l'	'l=4'

Table 8: Dictionary iteration and format string errors.

E.5 Truncation Failures (8% of CruxEval-O failures)

Truncation occurs when execution traces exceed the token budget.

Pattern	Example Code	Cause
List mutation	`for n in numbers[index:]:` `numbers.insert(index, n)`	Infinite loop from growing list
Nested iteration	`for i in range(len(text)):` `for j in range(i, len(text)):`	$O(n^2)$ trace expansion
Long input	Input string >30 characters with per-char loop	Dense trace per character

Table 9: Truncation patterns. Dense state snapshots cause traces to exceed the 8K token budget.

F CODE DECOMPOSITION INTERVENTIONS

The intervention rewrites programs so that intermediate values implicit in standard Python syntax become explicit variables in the execution trace. This allows CWMs to condition on intermediate dummy variables at each generated command. We implement the intervention using Python's Abstract Syntax Tree (AST) and apply it only to failing samples.

We extract complex subexpressions into temporary variables while preserving evaluation order and program semantics. For example:

Expression Decomposition

```
# Before:
output.append((nums.count(n), n))

# After:
_t0 = nums.count(n)
_t1 = (_t0, n)
output.append(_t1)
```

We traverse the AST and identify subexpressions whose structural complexity exceeds a threshold. For each such subexpression, we:

1. Generate a fresh temporary variable name.
2. Insert an assignment that extracts the subexpression.
3. Replace the original subexpression with the temporary variable reference.

Atomic nodes (`Name`, `Constant`) are left unchanged. Extraction is applied to `BinOp`, `Call`, `Subscript`, `Compare`, and container literals. Control-flow structure is preserved: extracted assignments are inserted immediately before the original statement.

F.1 STRING OPERATION DECOMPOSITION

String methods such as `.index`, `.find`, and `.replace` often fail when the target character is embedded inside a single BPE token, rendering the relevant character-level state invisible to the model. To address this, we expand selected string operations into explicit character-level loops.

We decompose the following single-character string operations:

`index`, `rindex`, `find`, `rfind`, `count`, `replace`, `c in s`

Decomposition is applied only when the search target is statically verified to be a single character.

Example:

String Operation Decomposition (`index`)

```
# Original:
pos = text.index(char)

# Transformed:
_t0 = list(text)
_t1 = -1
for _t2 in range(len(_t0)):
    if _t0[_t2] == char:
        _t1 = _t2
        break
if _t1 == -1:
    raise ValueError("substring not found")
pos = _t1
```

This expansion makes each character comparison explicit in the trace, restoring per-position visibility. For each rewritten program, semantic equivalence is verified by executing both the original and transformed code on the same inputs and checking for identical outputs.

F.2 Failure Modes and Limitations

While decomposition recovers a subset of failures, it introduces two inherent limitations.

Token explosion. Loop-based string decomposition generates one trace step per character. For long strings, this leads to trace lengths that exceed the context window, resulting in truncation.

Unrecoverable errors. Failures caused by incorrect semantic reasoning (e.g., misunderstanding Python method behavior), long-horizon state drift through loops, or complex control flow are not corrected by decomposition, as these errors do not arise from missing intermediate visibility.

The aggregate effect of these interventions is reported in Table 1 under "After Intervention. This result illustrates the limited but consistent improvement in controlled string composition after decomposition. We emphasize that decomposition is used here as a diagnostic tool to isolate visibility-related failures, rather than as a scalable execution strategy.