Code Is Not Just Text: Transformers as Bounded Symbolic Interpreters

Anonymous ACL submission

Abstract

Code generation is often treated either as shallow next-token prediction or as evidence that large language models have acquired something like unbounded program execution. We argue for a third position: code is not just text, and transformers are better understood as bounded symbolic interpreters over code contexts. Extending a trajectory-based bounded-interpreter framework, we define code interpretation as a structured state over syntax, binding, and procedure, and argue that successful code continuation requires transformer hidden and logit trajectories to preserve these nonlocal constraints. In a controlled synthetic code language, a small transformer recovers syntactic validity, branch-dependent binding, procedural condition, and final result with perfect held-out accuracy; local Jacobian analysis shows roughly 5.5× greater sensitivity to semantically relevant code tokens than to structural filler; and low-dimensional trajectory analysis finds that a 24-dimensional lifted representation explains 98.2% of hidden-state variance while preserving result-readout accuracy. These results support the view that code competence is neither mere autocomplete nor literal Turing-machine execution, but bounded context-conditioned symbolic interpretation implemented through finite transformer trajectories.

1 Introduction

Code generation exposes a weakness in two common descriptions of large language models. On one side, LLMs are often described as stochastic parrots or surface-level next-token predictors: systems that reproduce statistical regularities without deeper state (**?**). On the other side, their ability to generate and repair programs is sometimes taken as evidence that they have acquired something close to general symbolic computation, especially given the strong performance of code-specialized and code-capable models on synthesis and programming benchmarks (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021; Li et al., 2022; Fried et al., 2023). Both views miss the middle regime in which transformer code competence actually lives.

Code is a hostile domain for the stochastic-parrot account. Unlike ordinary prose, code is governed by constraints that are sparse, brittle, and nonlocal. A correct continuation may depend on an open bracket many tokens earlier, a variable introduced in another clause, a function signature, a branch condition, an imported library, or a procedural goal established by the prompt. This is consistent with a long line of work treating programs as structured objects rather than ordinary text, including work on machine learning for "big code," code representation learning, data-flow-aware pretraining, and identifier-sensitive code models (Allamanis et al., 2018; Feng et al., 2020; Guo et al., 2021; Wang et al., 2021). A model that only extends local surface statistics should fail whenever the locally plausible continuation differs from the contextually valid one. Yet modern LLMs often maintain enough bounded program state to generate syntactically valid code, preserve variable names, respect scopes, continue procedures, and repair errors across finite contexts (Chen et al., 2021; Li et al., 2022; Fried et al., 2023).

But code competence also does not make an LLM a classical Turing machine. Transformers do not have infinite tape, unbounded memory, exact execution semantics, or guaranteed algorithmic generalization. Their computation is finite-depth, finite-context, finite-dimensional, and approximate. Theoretical analyses of self-attention and formal-language recognition show that transformers have important limitations as sequence recognizers and do not straightforwardly implement unbounded automata (Hahn, 2020; Bhattamishra et al., 2020). They can track program-like structure over bounded horizons, but they do not thereby be-

come unbounded symbolic machines in the classical automata-theoretic sense (Hopcroft et al., 2006). The important question is therefore not whether LLMs are "just statistics" or "real computers." The question is what kind of finite computation transformer trajectories implement when they succeed on structured symbolic domains.

We argue that code generation is best understood as *bounded symbolic interpretation*. On this view, a transformer exposed to code constructs a finite prompt-conditioned state trajectory that approximates the symbolic constraints needed for continuation. This framing builds on the transformer architecture itself: attention mixes contextual value information into target-token states, layer updates evolve hidden representations, and output projections expose layerwise logit trajectories (Vaswani et al., 2017; Elhage et al., 2021). For code, the relevant state is organized around three coordinates: syntax, binding, and procedure. Syntax tracks structural well-formedness, such as delimiters, indentation, and local grammatical compatibility. Binding tracks references, names, scopes, and variable assignments. Procedure tracks control flow, branch conditions, and the continuation of computational intent. A successful code continuation is not merely a likely string; it is a continuation compatible with these bounded symbolic constraints.

This paper develops the bounded-interpreter account in three steps. First, we formalize code interpretation as a structured state over syntax, binding, and procedure. Second, we connect this state to transformer mechanics through hidden-state trajectories, layerwise logits, and probe-mass readouts. This makes code competence a trajectory-level object of study: symbolic constraints should become visible in how probability mass moves across layers. This view is aligned with recent mechanistic and representation-level work showing that transformer internals can encode structured variables and support interpretable circuit- or trajectory-level analysis (Olah et al., 2020; Elhage et al., 2021; Nanda et al., 2023; Gurnee and Tegmark, 2024). Third, we test the account in a controlled synthetic code language where the relevant latent variables are known by construction.

Our empirical results support the bounded symbolic interpretation view. In the synthetic code setting, a small transformer recovers syntactic validity, branch-dependent binding, procedural condition, and final result with perfect held-out accuracy. A local Jacobian analysis shows that the result readout is substantially more sensitive to semantically relevant code tokens than to structural filler, with approximately 5.5 times greater sensitivity to relevant context. Finally, low-dimensional trajectory analysis shows that a 24-dimensional lifted representation explains 98.2% of hidden-state variance while preserving result-readout accuracy. Together, these results indicate that the model is not merely matching surface code patterns; it is learning a bounded internal state that tracks the symbolic variables required for continuation.

The contribution of this paper is not the claim that transformers execute programs in the classical sense. Nor is it the claim that next-token prediction is irrelevant. Rather, we argue that next-token prediction can induce finite state trajectories that approximate symbolic interpretation over bounded contexts. Code makes this visible because its local tokens are constrained by nonlocal syntax, binding, and procedure. Thus code is not just text, and transformers are not best understood as either stochastic parrots or Turing machines. They are bounded symbolic interpreters: finite neural systems that approximate structured computation through prompt-conditioned trajectories.

2 Code Between Autocomplete and Computation

Code generation makes two common descriptions of LLMs look incomplete. If LLMs were merely stochastic parrots, code should be a hostile domain: correct continuation depends not only on fluent local text, but on sparse and nonlocal constraints such as delimiters, variable bindings, scope, branch conditions, and procedural goals (**?**). This is why prior work on code modeling treats programs as structured objects involving identifiers, data flow, syntax, and code-specific representations, rather than as ordinary prose (Allamanis et al., 2018; Feng et al., 2020; Guo et al., 2021; Wang et al., 2021). The strong performance of modern models on synthesis, infilling, and programming benchmarks suggests that next-token prediction can induce structured internal behavior rather than only shallow continuation (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021; Li et al., 2022; Fried et al., 2023).

At the same time, code competence does not make a transformer a classical Turing machine. A transformer computes a finite sequence of hidden-

state updates over a bounded context window (Vaswani et al., 2017). It has finite depth, finite dimensionality, no guaranteed unbounded tape, and no promise of exact execution. Theoretical work on self-attention and formal-language recognition shows that transformers have important limitations as sequence recognizers and do not straightforwardly implement unbounded automata (Hahn, 2020; Bhattamishra et al., 2020). Thus code competence is evidence of structured state, but not evidence of unrestricted symbolic computation in the automata-theoretic sense (Hopcroft et al., 2006).

We argue that the right middle category is *bounded symbolic interpretation*. A bounded symbolic interpreter is a finite system that maps a context into an internal state sufficient to support some class of context-sensitive continuations. For code, this state is organized around three coordinates: *syntax*, *binding*, and *procedure*. Syntax tracks structural well-formedness; binding tracks identifiers, references, and scope-like relations; procedure tracks control flow, branch conditions, and computational intent. The model is still trained by next-token prediction, but when the next token depends on an earlier binding, an unresolved syntactic structure, or a branch condition, prediction pressures the model to represent those facts. This view is consistent with evidence that language-model objectives induce structured representations (Radford et al., 2019; Brown et al., 2020; Kaplan et al., 2020), and with mechanistic work showing that transformer internals can encode variables accessible through circuits, probes, and trajectory-level analyses (Olah et al., 2020; Elhage et al., 2021; Nanda et al., 2023; Gurnee and Tegmark, 2024).

3 Transformer Trajectories as Bounded Code Interpreters

We now connect bounded symbolic interpretation to transformer computation. The claim is simple: if a model successfully continues code whose correct continuation depends on syntax, binding, and procedure, then some part of its finite computation must preserve information about those constraints. This preservation need not be exact, explicit, or unbounded. It is enough that the model's hidden and logit trajectories separate code states that require different continuations.

Let $x = (x_1, \ldots, x_T)$ be an input sequence and let t be the target position. A transformer with L layers produces hidden states $H^{(0)}, \ldots, H^{(L)}$, where $H^{(\ell)} \in \mathbb{R}^{T \times d}$. For target state $h_t^{(\ell)}$, the layerwise logits are

$$z_t^{(\ell)} = h_t^{(\ell)} W_U.$$

The target-token logit trajectory is

$$\mathcal{T}_t(x) = (z_t^{(0)}, \ldots, z_t^{(L)}).$$

This is the output-space trace of the model's finite hidden-state computation.

For a code context c and continuation point u, define the code interpretation state as

$$I_{\text{code}}(c, u) = (S(c, u), B(c, u), P(c, u)),$$

where S, B, and P denote syntax, binding, and procedural state. Let Ω_S, Ω_B, and Ω_P be continuation classes compatible with these coordinates. For any logit vector z, define probe mass as

$$M_\Omega(z) = \sum_{i \in \Omega} softmax(z)_i.$$

The layerwise code-interpreted state is

$$m_t^{(\ell)} = \big(M_{\Omega_S}(z_t^{(\ell)}), M_{\Omega_B}(z_t^{(\ell)}), M_{\Omega_P}(z_t^{(\ell)})\big),$$

and the code-interpreted trajectory is

$$\widetilde{\mathcal{T}}_{\text{code},t} = (m_t^{(0)}, \ldots, m_t^{(L)}).$$

When the context is clear, we suppress the dependence of the probe sets on (c, u). These quantities do not require the model to store symbolic variables in explicit registers. They only test whether its finite trajectory assigns distinguishable output-space support to continuations compatible with syntax, binding, and procedure.

The central claim is that successful code continuation requires such trajectory-level separation. Consider two code prompts (c_1, u) and (c_2, u) with the same local continuation point but different binding or procedural state:

$$B(c_1, u) \neq B(c_2, u) \quad \text{or} \quad P(c_1, u) \neq P(c_2, u).$$

If a transformer successfully continues both prompts, then their code-interpreted trajectories must differ:

$$\widetilde{\mathcal{T}}_{\text{code},t}(c_1, u) \neq \widetilde{\mathcal{T}}_{\text{code},t}(c_2, u).$$

Otherwise their final interpreted states would be identical, and the model could not distinguish continuations requiring different bindings or procedures. The full formal statement and proof are given in Appendix A.

This claim is deliberately bounded. It does not say that transformers execute arbitrary programs, maintain an unbounded stack, or simulate a Turing machine. It says that, over finite contexts and finite depths, successful code continuation requires enough symbolic state to distinguish syntax-compatible, binding-compatible, and procedure-compatible continuations. The model is therefore a bounded code interpreter: not because it performs exact execution, but because its finite trajectory supports structured continuation under nonlocal code constraints.

The bounded-interpreter view predicts three empirical signatures. First, final states should recover the relevant symbolic coordinates: syntax, binding, procedure, and result. Second, result predictions should be more sensitive to semantically relevant context tokens than to structural filler. Third, hidden-state trajectories should admit a compact representation in which code-relevant regimes are separable. We test these predictions next.

4 Empirical Evidence

We test the bounded-interpreter account in a controlled synthetic code language. The goal is not to claim that the task captures the full complexity of Python, JavaScript, or real software engineering. Rather, the goal is to isolate the symbolic variables that the theory says a code-generating transformer must track: syntax, binding, procedure, and result. Because these variables are known by construction, the setting allows direct evaluation of whether a finite transformer can learn a bounded code state rather than merely matching local token patterns.

4.1 Synthetic Code Language and Model

The synthetic language consists of short code-like programs with declarations, branch conditions, and print-style result statements. A representative valid program has the form:

```
let x = 1; let y = 4; if T then print x
else print y
```

The latent labels are constructed from the program state. The syntax label indicates whether the program is structurally valid. The binding label indicates which variable is selected by the active branch. The procedure label indicates which branch condition is active. The result label indicates the value that should be produced by the selected binding. Thus, success requires the model to recover not only local well-formedness, but also the interaction between branch condition, variable binding, and output value.

We train a small decoder-only transformer on this language. The model uses a vocabulary of synthetic code tokens, maximum sequence length 24, hidden dimension 96, 4 transformer layers, and 4 attention heads. It is trained for 6 epochs and evaluated on a held-out test split containing 1,800 examples, of which 1,401 are syntactically valid. We evaluate four readouts from the final representation: syntax, binding, procedure, and result. Binding, procedure, and result are evaluated on valid examples, since invalid programs do not define a unique executable state.

4.2 Structured State Readout

Table 1 shows that the transformer recovers all four structured labels with perfect held-out accuracy. This result is important because the labels are not independent surface properties. Correct result prediction depends on the interaction between the active branch and the variable bindings established earlier in the program. A model can only solve the task robustly if it maintains a bounded representation of the relevant code state.

This does not show that the model has learned an unbounded interpreter. The language is finite, the context window is short, and the symbolic state is deliberately controlled. However, the result does show that a finite transformer can learn the kind of state posited by the theory: a compact internal representation that jointly tracks syntax, binding, procedure, and result over bounded contexts.

4.3 Local Jacobian Sensitivity

A possible objection is that the model may solve the task through shallow correlations rather than through context-sensitive state. To test this, we compute a local Jacobian sensitivity measure. For each valid example, we take the logit corresponding to the correct result and compute the gradient norm with respect to input token embeddings. We then compare sensitivity to semantically relevant tokens against sensitivity to structural filler.

The relevant tokens are the branch condition and branch-selected variables. The irrelevant or structural tokens include fixed syntax markers such as `let`, `=`, `;`, `then`, and `else`. If the result readout depends on symbolic state rather than uniform token memorization, its gradient should concentrate more strongly on the branch and binding tokens than on filler tokens.

Metric	Accuracy
Syntax accuracy	1.000
Binding accuracy, valid examples	1.000
Procedure accuracy, valid examples	1.000
Result accuracy, valid examples	1.000

Table 1: Structured readout accuracy on the synthetic code test set. The model recovers syntactic validity, branch-dependent binding, procedural condition, and final result.

Quantity	Value
Samples	400
Mean relevant gradient norm	0.982
Mean irrelevant gradient norm	0.177
Mean sensitivity ratio	5.492
Median sensitivity ratio	5.498

Table 2: Local Jacobian sensitivity of the correct result logit. The result readout is substantially more sensitive to semantically relevant branch and binding tokens than to structural filler.

Table 2 shows that the correct result logit is approximately 5.5 times more sensitive to relevant code tokens than to structural filler. This supports the bounded-interpreter view: the model's output is not equally driven by all nearby code-like tokens, but is selectively sensitive to the variables that determine the program state. In this setting, the model's final prediction depends most strongly on precisely the tokens that define binding and procedure.

4.4 Low-Dimensional Trajectory Dynamics

The theory also predicts that code interpretation should be visible as a trajectory-level phenomenon, not only as a final readout. We therefore analyze the model's hidden states across layers. We project layerwise hidden states into a 24-dimensional PCA space and fit linear transition operators to the resulting lifted trajectories. We compare a single global transition operator against regime-conditioned operators, where regimes are defined by procedure and syntactic validity.

This analysis asks whether the model's layerwise computation has a compact trajectory structure, and whether code-relevant regimes explain differences in state evolution. If the model merely memorized output labels, we would not expect a low-dimensional trajectory representation to preserve result information while also showing improved regime-conditioned dynamics.

Table 3 shows three findings. First, the 24-dimensional PCA representation explains 98.2% of hidden-state variance, indicating that the learned computation is highly compressible. Second, the reduced final state preserves perfect result-readout accuracy. Third, regime-conditioned transition operators improve over a single global operator, reducing mean relative dynamics error from 0.296 to 0.247. This suggests that the model's layerwise computation is not a single undifferentiated trajectory, but varies systematically with code-relevant regimes.

Quantity	Value
PCA dimension	24
Explained variance	0.982
Global relative dynamics error	0.296
Mean regime relative dynamics error	0.247
Mean improvement over global	0.049
Reduced-state result accuracy	1.000

Table 3: Low-dimensional trajectory analysis. A 24-dimensional lifted representation captures most hidden-state variance, preserves result-readout accuracy, and yields better regime-conditioned dynamics than a single global transition operator.

For completeness, Table 4 reports the regime-level operator errors. The largest improvement appears for syntactically invalid examples, where a regime-specific operator substantially improves over the global transition fit. Procedure-conditioned operators also improve over the global fit, though more modestly.

4.5 Interpretation

Across the three analyses, the same pattern appears. The model recovers the explicit symbolic coordinates of the synthetic code language; its result logit is far more sensitive to branch and binding tokens than to filler; and its hidden-state trajectory admits a compact, regime-sensitive representation. These are precisely the signatures predicted by the bounded-interpreter account.

The results should be read carefully. They do not show that transformers perform exact program execution, generalize to arbitrary program length, or implement an unbounded stack or tape. They also do not show that frontier-scale code models use the same mechanism in all real programming settings. What they show is narrower but important: when code interpretation is made explicit and bounded, a small transformer learns finite internal trajectories that preserve syntax, binding, procedure, and result. This is enough to reject the view that successful code continuation is merely local text mimicry, while avoiding the stronger and unsupported claim that LLMs are classical symbolic

Regime	Value	Error	Gain
Procedure	0	0.252	0.045
Procedure	1	0.281	0.016
Syntax valid	0	0.180	0.116
Syntax valid	1	0.276	0.021

Table 4: Regime-conditioned lifted dynamics. Gain denotes improvement over the global relative dynamics error of 0.296.

machines.

5 Discussion

The empirical results support a middle account of transformer code competence. The model is not merely reproducing code-shaped token sequences: it recovers syntax, binding, procedure, and result in a setting where these variables are independently defined. Its result prediction is also selectively sensitive to branch and binding tokens rather than uniformly sensitive to surrounding syntax. Finally, its hidden-state evolution admits a compact trajectory representation in which code-relevant regimes improve the fit of layerwise dynamics. Together, these findings are difficult to reconcile with a purely local autocomplete account.

At the same time, the results do not justify treating the model as an unbounded program executor. The synthetic language is finite, the context length is bounded, and the model is small. The learned computation is better described as approximate state tracking than exact symbolic execution. This distinction matters because code generation often invites overinterpretation. A model that continues a branch correctly has not necessarily learned operational semantics in the classical sense. It may instead have learned a bounded internal representation sufficient to preserve the relevant syntactic, binding, and procedural constraints over the observed horizon.

The bounded-interpreter view therefore explains both the strength and fragility of LLM code behavior. When the relevant state fits inside the model's finite context and representational capacity, code generation can look strikingly symbolic: identifiers are preserved, branches are followed, and results are propagated. When the required state exceeds that capacity, or when the prompt demands exact execution outside the learned distribution, failures are expected. This is not a contradiction. It is exactly what one should expect from a finite interpreter-like system without unbounded memory, external tape, or formal correctness guarantees.

This framing also clarifies the role of the training objective. Next-token prediction is not denied. Rather, the claim is that next-token prediction over structured domains can induce internal states that are more organized than surface continuation. If predicting the next token requires knowing which variable is bound, which branch is active, or which syntactic form is open, then the model is pressured to represent those facts. The result is a finite trajectory that behaves interpreter-like over bounded contexts, even though the objective remains distributional.

More broadly, code provides a clean instance of a general pattern. In natural-language interpretation, the relevant latent variables may be discourse role, stance, outcome, or action. In safety behavior, they may be refusal-compatible and unsafe-compatible continuation regions. In code, they are syntax, binding, and procedure. These domains differ in content, but they share a trajectory-level structure: successful behavior requires the model to preserve context-sensitive variables through finite hidden-state updates and expose them through the output distribution. Code is especially useful because the variables can be specified and tested directly.

The phrase *bounded symbolic interpreter* is therefore meant literally but carefully. The system is symbolic in the limited sense that its behavior depends on structured variables such as names, branches, and syntactic states. It is interpretive in the sense that the same local continuation point can require different completions depending on earlier context. It is bounded because all of this occurs within finite context, finite depth, finite dimensionality, and finite training support. This middle category avoids two mistakes at once: reducing code competence to shallow mimicry, and inflating it into unbounded computation.

6 Conclusion

Code is not just text. Its local tokens are governed by nonlocal constraints over syntax, binding, and procedure. If LLMs were only stochastic parrots, code should expose them as brittle surface continuers. If they were classical Turing machines, their code competence would imply unbounded memory and exact symbolic execution. Neither description fits.

We have argued that transformers are better understood as bounded symbolic interpreters over

code contexts. They construct finite prompt-conditioned trajectories that preserve enough symbolic state to support structured continuation, without becoming unbounded program executors. In a controlled synthetic code language, a small transformer recovers syntax, binding, procedure, and result; shows substantially greater sensitivity to semantically relevant code tokens than to structural filler; and exhibits compact, regime-sensitive trajectory dynamics.

These results support the broader bounded-interpreter view of LLM inference. The important question is not whether next-token prediction is "mere statistics" or whether LLMs are full symbolic machines. The important question is what kinds of finite state trajectories next-token prediction induces. For code, the answer is clear: successful continuation requires bounded symbolic state.

Limitations

This study uses a controlled synthetic code language rather than full-scale real-world programming languages. This choice is deliberate: it allows syntax, binding, procedure, and result variables to be specified exactly and evaluated directly. The resulting evidence therefore supports the bounded symbolic interpretation hypothesis in a clean finite setting, but does not by itself establish that all frontier code models use the same mechanisms across arbitrary languages, libraries, or program lengths. The claims should be read as mechanistic and conceptual rather than benchmark-comparative: we show that finite transformer trajectories can learn and expose bounded symbolic state under controlled code-like constraints, not that transformers perform exact execution or possess unbounded algorithmic generalization.

Ethics Statement

This work studies transformer representations in a synthetic code setting and does not introduce a deployed system, user-facing model, or dataset containing personal or sensitive information. The main ethical risk is interpretive overclaiming: describing LLMs as symbolic interpreters could be misread as implying reliable program execution or formal correctness. We avoid that claim explicitly. Our framing emphasizes boundedness, approximation, and finite context, and is intended to improve scientific understanding of model behavior rather than encourage blind reliance on generated code. Any practical use of code-generating systems should still involve testing, verification, and human review.

References

Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys*, 51(4):1–37.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. 2020. On the ability and limitations of transformers to recognize formal languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 7096–7116. Association for Computational Linguistics.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, and 12 others. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, and 6 others. 2021. A mathematical framework for transformer circuits. Transformer Circuits Thread.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547. Association for Computational Linguistics.

Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder:

A generative model for code infilling and synthesis. In *International Conference on Learning Representations*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training code representations with data flow. In *International Conference on Learning Representations*.

Wes Gurnee and Max Tegmark. 2024. Language models represent space and time. In *International Conference on Learning Representations*.

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

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Advances in Neural Information Processing Systems Datasets and Benchmarks Track*.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2006. *Introduction to Automata Theory, Languages, and Computation*, 3 edition. Pearson.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Remi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, and 7 others. 2022. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097.

Neel Nanda, Lawrence Chan, Tom Liberum, Jess Smith, and Jacob Steinhardt. 2023. Progress measures for grokking via mechanistic interpretability. In *International Conference on Learning Representations*.

Chris Olah, Nick Cammarata, Ludwig Schubert, Gabriel Goh, Michael Petrov, and Shan Carter. 2020. Zoom in: An introduction to circuits. *Distill*.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. OpenAI technical report.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 8696–8708. Association for Computational Linguistics.

A Formal Setup and Proofs

A.1 Transformer and Trajectory Notation

Let an input sequence be

$$x = (x_1, \ldots, x_T),$$

where each x_i is a token from vocabulary V. A transformer with L layers induces hidden-state matrices

$$H^{(0)}, H^{(1)}, \ldots, H^{(L)},$$

where $H^{(\ell)} \in \mathbb{R}^{T \times d}$. For a target position t, let $h_t^{(\ell)} \in \mathbb{R}^d$ denote the hidden state at layer ℓ. With output projection $W_U \in \mathbb{R}^{d \times |V|}$, the layerwise logits are

$$z_t^{(\ell)} = h_t^{(\ell)} W_U.$$

The corresponding logit trajectory is

$$\mathcal{T}_t(x) = (z_t^{(0)}, \ldots, z_t^{(L)}).$$

This trajectory is the output-space projection of the model's finite hidden-state computation.

For a code prompt, we write the input as a pair (c, u), where c is the code context and u is the target continuation point. The target position t is the position at which continuation is evaluated.

A.2 Code Interpretation Variables

We define code interpretation as a bounded structured state over three coordinates:

$$I_{\text{code}}(c, u) = (S(c, u), B(c, u), P(c, u)).$$

Here, $S(c, u)$ is the syntactic constraint state, $B(c, u)$ is the binding or reference state, and $P(c, u)$ is the procedural continuation state.

The syntactic state includes local grammar compatibility, delimiter state, statement/expression compatibility, and structural validity. The binding state includes variables, references, assignments, and scope-like relations. The procedural state includes branch conditions, control-flow continuation, and result propagation.

This decomposition is not a full semantics for programming languages. It is a controlled abstraction that isolates the symbolic constraints most relevant to bounded code continuation.

A.3 Probe Masses and Interpreted Trajectories

For each coordinate, define a compatible continuation set:

$$\Omega_S, \quad \Omega_B, \quad \Omega_P \subseteq V.$$

When necessary, these sets depend on the context pair (c, u), but we suppress this dependence for readability.

For any logit vector z, define probe mass:

$$M_\Omega(z) = \sum_{i \in \Omega} \mathrm{softmax}(z)_i.$$

The layerwise code-interpreted state is

$$m_t^{(\ell)} = \big(M_{\Omega_S}(z_t^{(\ell)}), M_{\Omega_B}(z_t^{(\ell)}), M_{\Omega_P}(z_t^{(\ell)})\big).$$

The code-interpreted trajectory is

$$\widetilde{\mathcal{T}}_{\mathrm{code},t} = (m_t^{(0)}, \ldots, m_t^{(L)}).$$

This representation does not require the model to store symbolic variables in explicit registers. It only requires that its hidden/logit trajectory expose distinguishable support for syntax-compatible, binding-compatible, and procedure-compatible continuation classes.

A.4 Code Continuation Support

Claim. If a transformer successfully continues a code prompt (c, u), then its final logit state must assign nonzero support to continuation classes compatible with the required syntactic, binding, and procedural constraints.

Proof. Let the final predictive distribution be

$$p_t^{(L)} = \mathrm{softmax}(z_t^{(L)}).$$

Suppose the continuation requires compatibility with a code coordinate $Y \in \{S, B, P\}$, and let Ω_Y be the corresponding compatible continuation class. If

$$M_{\Omega_Y}(z_t^{(L)}) = 0,$$

then the final predictive distribution assigns zero probability mass to all continuations compatible with Y. A successful continuation requiring Y-compatibility would then be impossible. Therefore

$$M_{\Omega_Y}(z_t^{(L)}) > 0.$$

Applying the same argument to syntax, binding, and procedure yields the claim.

A.5 Binding/Procedure-Sensitive Divergence

Claim. Let (c_1, u) and (c_2, u) be code-context pairs with the same local continuation point u. Suppose they are locally syntax-compatible but differ in binding or procedural state:

$$B(c_1, u) \neq B(c_2, u) \quad \text{or} \quad P(c_1, u) \neq P(c_2, u).$$

If a transformer successfully continues both prompts, then their code-interpreted trajectories differ:

$$\widetilde{\mathcal{T}}_{\mathrm{code},t}(c_1, u) \neq \widetilde{\mathcal{T}}_{\mathrm{code},t}(c_2, u).$$

Proof. Because the prompts differ in binding or procedure, there is at least one coordinate $Y \in \{B, P\}$ for which the compatible continuation class differs across contexts. Successful continuation on (c_1, u) requires final support for $\Omega_Y(c_1, u)$, while successful continuation on (c_2, u) requires final support for $\Omega_Y(c_2, u)$. Since the required compatible classes differ, the final interpreted states must differ in a code-relevant coordinate:

$$m_t^{(L)}(c_1, u) \neq m_t^{(L)}(c_2, u).$$

If the full interpreted trajectories were identical, their final elements would also be identical, contradicting the final-state difference. Therefore the interpreted trajectories differ.

A.6 Latent Code-State Necessity

Claim. If two code prompts have the same local continuation point but require different binding or procedural continuations, then successful continuation requires the model to encode a context-sensitive latent code state in its hidden/logit trajectory.

Proof. By the binding/procedure-sensitive divergence claim,

$$\widetilde{\mathcal{T}}_{\mathrm{code},t}(c_1, u) \neq \widetilde{\mathcal{T}}_{\mathrm{code},t}(c_2, u).$$

Therefore there exists some layer ℓ such that

$$m_t^{(\ell)}(c_1, u) \neq m_t^{(\ell)}(c_2, u).$$

Since $m_t^{(\ell)}$ is a function of $z_t^{(\ell)}$, and $z_t^{(\ell)} = h_t^{(\ell)} W_U$, the distinction must be present in the hidden/logit trajectory in a direction detectable by the code-probe readout. Thus the model must encode a context-sensitive latent code state sufficient to distinguish the two continuation requirements.

A.7 Bounded Symbolic Interpreter Theorem

Claim. A transformer that successfully continues code prompts whose correct continuations depend on syntax, binding, and procedure functions as a bounded symbolic interpreter over those code contexts.

Proof. For every input, the transformer induces a finite hidden-state trajectory

$$H^{(0)} \to H^{(1)} \to \cdots \to H^{(L)}.$$

At each target position, this yields a finite logit trajectory

$$\mathcal{T}_t(x) = (z_t^{(0)}, \ldots, z_t^{(L)}).$$

Probe masses over syntax-, binding-, and procedure-compatible continuation classes define the interpreted trajectory

$$\widetilde{\mathcal{T}}_{\mathrm{code},t}.$$

By the support claim, successful code continuation requires support for the relevant symbolic continuation classes. By the divergence claim, contexts requiring different binding or procedural continuations require different interpreted trajectories. By the latent-state claim, this implies that the model's finite trajectory encodes context-sensitive code state. Because the model has finite context, finite depth, and finite-dimensional hidden states, the resulting interpretation is bounded. Therefore the transformer functions as a bounded symbolic interpreter over code contexts.

A.8 Bounded Automata-Like Behavior

The bounded-interpreter view is compatible with automata-like behavior over finite horizons. Syntax tracking may resemble finite-state recognition; delimiter or indentation tracking may resemble bounded stack-like behavior; and binding/procedure tracking may resemble bounded symbolic execution. However, the transformer does not thereby become an unbounded automaton or Turing machine. Its context, depth, and hidden dimensionality are finite, and its computation provides no general guarantee of exact execution or arbitrary-length generalization.

B Synthetic Code Language and Training Details

B.1 Grammar and Program Generation

The empirical study uses a synthetic code-like language designed to isolate syntax, binding, procedure, and result. Programs contain declarations, branch conditions, and branch-dependent print statements. A representative valid program is:

```
let x = 1; let y = 4; if T then print x
else print y
```

The language contains variable tokens, integer values, assignment syntax, Boolean branch conditions, and print-style result expressions. Programs are generated so that the active branch determines which variable binding is selected, and the selected variable determines the final result.

Invalid examples are generated by perturbing structural components of otherwise well-formed programs. These invalid programs are used for syntax classification, but do not define unique binding, procedure, or result labels.

B.2 Label Construction

Each example receives four labels:

- **Syntax**: whether the program is structurally valid.
- **Binding**: which variable is selected by the active branch.
- **Procedure**: which branch condition or procedural path is active.
- **Result**: which value should be produced by the selected binding.

Binding, procedure, and result labels are evaluated only on valid examples, because invalid programs do not define a unique executable state.

B.3 Model Architecture

The model is a small decoder-only transformer trained from scratch on the synthetic code language. The configuration is shown in Table 5.

Parameter	Value
Maximum sequence length	24
Hidden dimension	96
Transformer layers	4
Attention heads	4
Training epochs	6
Test examples	1,800
Valid test examples	1,401

Table 5: Synthetic code transformer configuration.

B.4 Training History

The model converges rapidly on the synthetic language. The epoch-level training loss is shown in Table 6.

Epoch	Loss
1	2.696
2	1.523
3	1.470
4	1.386
5	0.669
6	0.063

Table 6: Training loss over six epochs.

C Additional Empirical Results

C.1 Structured Accuracy Results

Table 7 reports the full structured readout results for the synthetic code task. The model recovers all four target coordinates: syntactic validity, branch-dependent binding, procedural condition, and final result.

Metric	Value
Syntax accuracy	1.000
Binding accuracy, valid examples	1.000
Procedure accuracy, valid examples	1.000
Result accuracy, valid examples	1.000
Number of test examples	1,800
Number of valid examples	1,401

Table 7: Structured accuracy results for syntax, binding, procedure, and result.

These results show that a finite transformer can learn a bounded code state in which syntactic validity, active binding, procedural condition, and final result are jointly recoverable.

C.2 Local Jacobian Sensitivity

To test whether the result readout depends selectively on semantically relevant context, we compute gradient sensitivity of the correct result logit with respect to input token embeddings. Relevant tokens include branch conditions and branch-selected variables. Irrelevant tokens include structural filler such as `let`, `=`, `;`, `then`, and `else`.

For each example, we compute the mean gradient norm over relevant tokens and the mean gradient norm over irrelevant tokens. The sensitivity ratio is:

$$r = \frac{\text{mean}\|\nabla_{\text{rel}}\|}{\text{mean}\|\nabla_{\text{irrel}}\|}.$$

The roughly 5.5-fold sensitivity increase indicates that the model's result prediction is concentrated on the tokens that determine binding and procedure, rather than uniformly on all code-like tokens.

Quantity	Value
Samples	400
Mean relevant gradient norm	0.982
Mean irrelevant gradient norm	0.177
Mean sensitivity ratio	5.492
Median sensitivity ratio	5.498

Table 8: Local Jacobian sensitivity summary. The correct result logit is substantially more sensitive to branch and binding tokens than to structural filler.

C.3 Lifted Trajectory Dynamics

We analyze the model's layerwise hidden-state trajectory by projecting hidden states into a 24-dimensional PCA space. We then fit linear transition operators over the resulting lifted trajectories. We compare a single global operator against regime-conditioned operators, where regimes are defined by procedure and syntactic validity.

Quantity	Value
PCA dimension	24
Explained variance	0.982
Global relative dynamics error	0.296
Mean regime relative dynamics error	0.247
Mean improvement over global	0.049
Reduced-state result accuracy	1.000

Table 9: Lifted trajectory dynamics summary.

The 24-dimensional representation captures 98.2% of hidden-state variance while preserving perfect result-readout accuracy. Regime-conditioned operators improve over a single global transition operator, suggesting that code-relevant regimes correspond to different trajectory dynamics.

Regime	Value	N	Error	Gain
Procedure	0	872	0.252	0.045
Procedure	1	928	0.281	0.016
Syntax valid	0	399	0.180	0.116
Syntax valid	1	1,401	0.276	0.021

Table 10: Regime-conditioned lifted operator errors. Gain is improvement over the global relative dynamics error of 0.296.

C.4 Empirical Coverage Matrix

Table 11 maps the theoretical components of the bounded-interpreter account to the empirical evidence used in this paper.

D Extended Discussion

D.1 Why This Is Not Mere Autocomplete

A surface-continuation account predicts that code generation should be dominated by local token reg-

Theory block	Status	Evidence	Interpretation
Code as syntax/binding/procedure interpretation	Main empirical result	Structured readouts over syntax, binding, procedure, and result	Finite transformer states recover bounded symbolic variables.
Local context sensitivity	Main empirical result	Jacobian sensitivity of result logit to relevant versus filler tokens	Result prediction depends more strongly on branch and binding tokens.
Low-dimensional trajectory structure	Main empirical result	PCA and lifted transition-operator analysis	Hidden-state trajectories are compact and regime-sensitive.
Bounded rather than unbounded computation	Theoretical and empirical	Finite synthetic language, finite context, finite model	Evidence supports bounded symbolic interpretation, not exact arbitrary execution.

Table 11: Empirical coverage matrix for the claims evaluated in this paper.

ularities. The synthetic experiments create cases where local code-like form is insufficient: the result depends on branch condition and variable binding. The model's perfect structured readout performance, together with the local Jacobian concentration on relevant tokens, supports the claim that it learns a bounded state over the relevant symbolic variables.

This does not imply that surface statistics play no role. The model is still trained by prediction, and the learned state is induced by distributional supervision. The point is narrower: distributional training can produce state-like internal structure when the prediction problem requires nonlocal symbolic constraints.

D.2 Why This Is Not an Unbounded Turing Machine

The bounded-interpreter account rejects the stronger claim that transformer code competence implies unbounded computation. A transformer has finite context, finite depth, finite hidden dimensionality, and no external tape. It may approximate stack-like or execution-like behavior over bounded horizons, but this is not equivalent to a classical Turing machine or an unbounded pushdown automaton.

The synthetic code task is therefore intentionally finite. It tests whether a transformer can learn bounded symbolic state, not whether it can execute arbitrary programs of unbounded length.

D.3 Relation to Formal-Language Work

Prior formal-language analyses show that transformers have limitations as recognizers of certain languages and do not straightforwardly implement unbounded sequence machines. The present account is compatible with those results. It does not require unbounded recognition or exact computation. Instead, it studies how finite transformer trajectories can approximate symbolic state over bounded contexts.

D.4 Relation to Mechanistic Interpretability

The analysis is trajectory-level rather than purely final-output-level. Probe masses read symbolic support from layerwise logits; local Jacobians measure sensitivity of the result readout to input tokens; and lifted dynamics study the evolution of hidden states across layers. These methods are aligned with mechanistic interpretability in spirit, but the target is a domain-level interpretation state rather than a single circuit.