LATENT PROGRAMMING HORIZONS IN CODING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

A coding agent solving a software-engineering task spends dozens of steps reasoning, editing code, and running tests, yet little is known about what the underlying language model internally represents about the program it is working on. We show that the residual streams of language models under coding agents linearly encode properties of the evolving program: a logistic-regression probe on hidden states is able to decode whether the current code parses, passes its test suite, reduces the number of failing tests, and introduces regressions, reaching AUC up to 0.83 for correctness across two models and two benchmarks. Our second finding is more surprising: these representations run ahead of the agent's own edits. Probes trained to predict the outcome of future edits (before they are materialized and written on disk) achieve performance above chance up to roughly 25 steps in advance. We call this the agent's *latent programming horizon*. As a proof of external validity, we show that the probes transfer across benchmarks without retraining. Our positive results open calls for more research in mechanistic interpretability of coding agents.

1 INTRODUCTION

Coding agents are increasingly used to solve complex software engineering tasks. Over many steps, they read files, reason about the problem, edit code, run tests, and revise their changes, iterating until the task is complete. Despite intense interest in building such agents, strikingly little is known about how the underlying language model represents the code it is working on. In this paper, we study the following question: *what does a coding agent internally represent about the program it is editing?*

Prior work in other domains gives reason to expect a rich answer. Transformers trained on board games develop linearly decodable internal representations of the game state (Li et al., 2023; Nanda et al., 2023). Models trained on synthetic grid-world programs encode both the current and future semantic state of those programs (Jin & Rinard, 2024).

In the software domain, prior work (Ribeiro et al., 2025; Bui et al., 2025; Vu et al., 2025) has established that the correctness of a single generated function is linearly decodable from a model's hidden states. Yet, these papers assume single-step generation where the full program sits in context and never changes. None of them is about coding agents iteratively editing an only partially observed real codebase, over dozens of steps, where the model's hidden state shifts with every edit. This is the problem we address in this paper.

To study this arguably difficult question, we proceed as follows. We systematically collect agentic trajectories, extract the residual stream of the model at each agentic step, and train linear probes to decode properties of the program being edited. We consider two open-weight models running `mini-swe-agent` (Yang et al., 2024) on two benchmarks (`SWE-Bench-Verified` (Jimenez et al., 2024) and `SWE-Bench-Pro` (Deng et al., 2025)). Two main findings emerge.

Coding agents encode the current program. Linear probes recover hard program properties. We show that the probes capture whether the current program parses, passes its test suite, reduces the number of failing tests, and introduces regressions, all well above a sound shuffled-label control in the unit of token. The signal persists throughout the trajectory and is strongest in the model's intermediate layers.

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

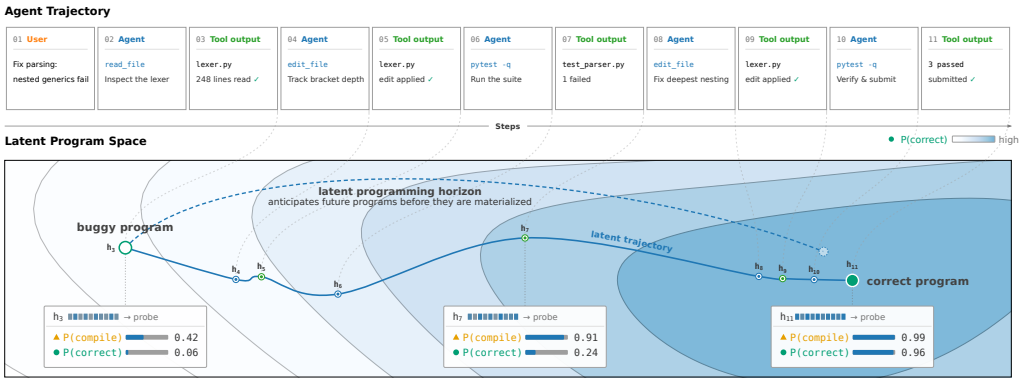

Figure 1: A coding agent iteratively edits code and runs tests (top); the hidden states at each step trace a corresponding path through the latent program space (bottom). The space is shaded by probe-estimated $P(\text{correct})$, with a warm basin marking the final region where the final program is fully correct and the coding task is successful. Our experiments show that linear probes trained on hidden states decode program properties such as full correctness well above chance throughout the agentic trajectories. Beyond current state, the latent program representation captures future programs' properties: this demonstrates that coding agents have a *latent programming horizon*.

Coding agents anticipate future programs. Beyond tracking the present, the agents represent programs that have not yet materialized. Probes trained to predict the outcome of a future edit succeed up to roughly 25 steps in advance. This is the coding analogue of learned look-ahead in game-playing networks (Jenner et al., 2024; Taufeeque et al., 2024). Coding agents have an idea of the shape of the future programs they will write, well before those changes are materialized with edits.

To the best of our knowledge, we are the first to show that coding agents maintain and continually update latent representations of programs and their semantic properties (Figure 1). Furthermore, we are the first to report on a long-term programming horizon in the latent space, with strong evidence from ahead-of-chance prediction capabilities.

Contributions.

- We propose a novel experimental protocol to study how coding agents maintain a representation of programs in the latent space.
- We show that linear probes on the hidden states of coding agents are able to decode four hard properties of programs. Our experimental results are based on real-world codebases and multi-step trajectories (median = 52 steps). We demonstrate that these probes transfer across benchmarks without retraining.
- We show that the latent space captures the programming horizon. The latent program representation grounds future edit prediction: probes can predict the outcome of an edit k steps ahead, and this holds for up to $k = 25$ steps.

2 Latent Program Representations

Consider a coding agent solving a programming task by iteratively searching and reasoning about the codebase, making edits, running tests, and using the feedback to guide its next steps. Beyond the tokens the underlying model emits and observes, such an agent may encode a latent program representation, an internal account of the program that it updates and revises. This representation, invisible to the harness and the user at the token level, is the object we study in this paper.

This representation, if it exists, lives somewhere in the internal calculation of the model. Indeed, transformer models maintain a running hidden-state vector, called the residual stream, which each layer reads from and writes an additive update back to, carrying information from the input through

to the output. We write $\mathbf{h}_t^\ell \in \mathbb{R}^d$ for the residual stream at token position t after layer ℓ. Because every layer communicates through it, the residual stream is the standard target for probing what a model internally represents (Elhage et al., 2021). In this paper, we study whether the residual stream contains latent program representations.

Definition 1 (Latent program representation)**.** *A latent program representation is what is encoded about a program in the residual stream $\mathbf{h}_t^\ell$. It is captured independently from the surface tokens the agent has emitted or will emit.*

The residual stream does not only contain the latent program representation. It also carries information about the task at hand and the past conversation, so the latent program representation is tangled with other signals in the hidden state (recovering it is the job of our probes, see Section 3.4).

Definition 2 (Latent program space)**.** *The latent program space is the region of the hidden-state space $\mathbb{R}^d$ along which latent program representations are encoded. As the agent edits code and runs tests, the hidden state moves along this region, forming a latent program representation trajectory.*

At each step, the hidden state is the point on this region, it captures the current program. A capable agent, however, might not only reason about the present. If the model is planning ahead, its current hidden state should also encode properties of the future programs, the ones it might write several steps ahead. We call the reach of this planning in the latent space the "latent programming horizon".

Definition 3 (Latent programming horizon)**.** *The latent programming horizon is the extent to which future edits are already encoded in the current residual stream. We define the latent programming horizon as the capability of predicting program properties of the program at $t + k$ steps in advance, only based on $\mathbf{h}_t^\ell$. We refer to it as the programming horizon at k.*

The horizon k captures the depth of future planning. Any prediction about the program at $k = 0$ means that the properties of the current program are represented in the latent space For small k (a short-term horizon), prediction success means the current hidden state already carries properties of edits that are a few steps away, for example, a sequence of edits in the same file. For large k (a long-term horizon), prediction success means the agent anticipates programs that it will write only much later in the trajectory.

While language models are not explicitly trained to encode program representations in their hidden states, several training signals may push them in that direction. Next-token prediction implicitly rewards learning program syntax and semantics, as predicting the next line of a function is easier if those are understood. Beyond next-token training objectives, modern code models are also post-trained with signals based on execution traces (Copet et al., 2025) or feedback from compilers and test suites (Ye et al., 2022). Basically, both pre-training and post-training should contribute to having semantically rich latent representations of the program a coding agent is working on.

We are the first to study whether such representations emerge in coding models and how coding agents maintain them during software engineering tasks.

3 METHODS

3.1 OVERVIEW

We study what coding agents do during multi-step agentic editing of real codebases, in the latent space. We need a way to capture program representations in the hidden state. For this, we follow the probing literature (Alain & Bengio, 2017; Zhang et al., 2025) and train linear probes to classify program properties from hidden states. Figure 2 shows an overview of our setup. If such probes predict program properties above chance, it shows the model actually encodes those properties in its residual stream.

3.2 FORMAL SETUP

Let $\mathcal{M}$ be a language model with L transformer layers. An agentic trajectory $\mathcal{T}$ for a coding task consists of S interaction steps. At each step s, the agent generates a response A_s given the accumulated context; the tool then produces an output O_s.

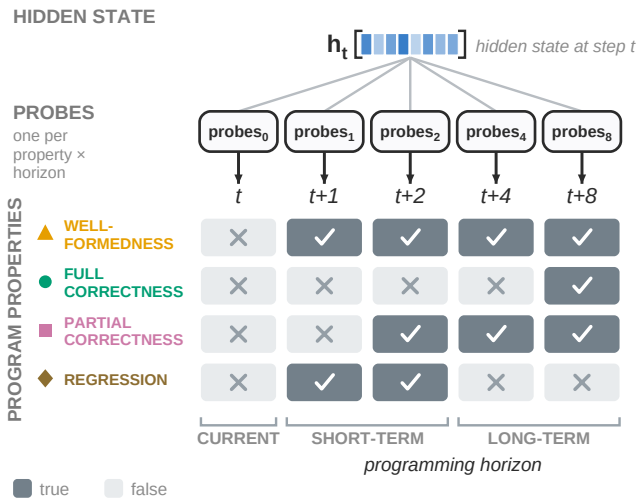

Figure 2: Our probing setup. Linear probes read the residual-stream vector $\mathbf{h}_t$ at step t of an agentic trajectory and predict one of four binary properties (▲ WELL-FORMEDNESS, ● FULL CORRECTNESS, ■ PARTIAL CORRECTNESS, ◆ REGRESSION) of the program k steps ahead. Probes at k=0 test whether the current program is represented; larger k measure the latent programming horizon. Labels (filled: true, light: false) come from checking out and evaluating each program version.

The full trajectory is the linearised concatenation:

$$\mathcal{T} = C_1 \oplus A_1 \oplus O_1 \oplus \cdots \oplus A_S \oplus O_S$$

where C_1 is the initial context (typically system and user prompts), and $\oplus$ denotes token-sequence concatenation.

For each position t in the generated portions $\bigcup_{s=1}^{S} A_s$, let $\mathbf{h}_t^\ell \in \mathbb{R}^d$ denote the residual stream after layer ℓ. We index layers from 1, so that $\mathbf{h}_t^1$ is the residual stream after the first transformer block and $\mathbf{h}_t^L$ after the final block.

An *edit event* occurs at step s when A_s includes a tool call that modifies the codebase. Let $e_1 < e_2 < \cdots$ denote the token indices of successive edit events within $\mathcal{T}$. The edits capture the trial-and-error process of the model to perform the task at hand.

In this paper, a *program property* ϕ is a function that assigns to each edit event e_i a binary label $y_i^\phi \in \{0, 1\}$ reflecting a property of the resulting program version. For example, the property 'compilability' captures whether a program at a given edit compiles, see Section 3.3 for the list of properties considered in this paper.

Labels are computed externally only at each event, and then kept until the next edit: every generated token position t with $e_i \leq t < e_{i+1}$ receives the label of edit e_i.

We also evaluate ϕ on the initial program version and treat it as a phantom edit e_0 preceding the trajectory, so every position t has a well-defined label even before the first agent edit.

It is convenient to index labels by step rather than by token: let $\sigma(t)$ be the step in which token t is generated (so $t \in A_{\sigma(t)}$), and let y_s^ϕ denote the label in effect at step s, that is, the label of the most recent edit at or before step s. The token-level label is then $y_t^\phi = y_{\sigma(t)}^\phi$.

3.3 PROGRAM PROPERTIES

We define four program properties to study through probing. We will train probes to predict both current and future values of these properties from the model's hidden states.

▲ **WELL-FORMEDNESS** Well-formedness is a necessary precondition for code to execute. We operationally define it by parsing or compiling the program at a given version. The label is 1 if it succeeds and 0 otherwise.

● **FULL CORRECTNESS** Full correctness tracks the functional behavior of the program, which is the ultimate goal of code generation and repair. The label is 1 if the program passes the benchmark's test suite oracle, and 0 otherwise.

■ **PARTIAL CORRECTNESS** Rather than asking whether the code is fully correct, this probe captures partial correctness, whether the program is moving in the right direction. The label is 1 if the number of failing tests is strictly less than at the beginning of the task, and 0 otherwise.

◆ **REGRESSION** A capable agent must attend not only to the tests it is trying to fix but also to those it must not break. This probe tests whether hidden states carry a signal about side effects of an edit. The label is 1 if any test that passed at the beginning of the task now fails, and 0 otherwise.

Together, these four properties span what a coding agent must track to resolve a task: the well-formedness of an edit (▲ WELL-FORMEDNESS), its functional correctness (● FULL CORRECTNESS), its directional progress per the test suite outcomes (■ PARTIAL CORRECTNESS), and its collateral effect on the rest of the repository (◆ REGRESSION).

3.4 PROBING PROGRAM POPERTIES

For each layer $\ell \in \{1, 11, 21, 31, 40\}$, we collect all hidden states $\mathbf{h}_t^\ell$ together with their labels y_t^ϕ. To test whether hidden states at layer ℓ carry information about ϕ, we train a logistic regression classifier, $\mathbf{w} \in \mathbb{R}^d$, to predict y_t^ϕ from the residual stream $\mathbf{h}_t^\ell$. We split trajectories into training, validation, and test sets (70/15/15% of tasks) based on task identifier, ensuring no trajectories of the same task appear in more than one split. As a sanity check, to verify that the probes decode a signal already present in the hidden states rather than fit it themselves (Hewitt & Liang, 2019), we also retrain each probe on randomly permuted labels, where labels are permuted at the level of individual hidden states.

The classifiers trained on one benchmark are further used to probe the hidden states of another to check if the signal they extract transfers across benchmarks. Probe hyperparameters are selected via random search sweeps based on mean validation AUC.

3.5 PROGRAMMING HORIZON

To study whether coding agents have a programming horizon, we study whether one can predict the outcome of a future edit before that particular edit appears in their output. Concretely, we train probes to predict program properties at varying lookahead horizons.

For each property ϕ and horizon $k \geq 0$ (in steps), we train a probe on the hidden state $\mathbf{h}_t^\ell$ to predict $y_{\sigma(t)+k}^\phi$, the label of the program k steps later. We fit a separate probe at each k. We sweep k from 0 (equivalent to the current-state probe) to $k_{\max} = 50$, excluding the final $k_{\max}$ steps of each trajectory to hold the token population constant across horizons. If the probe discriminative performance is higher than chance for a given k, it means that the coding agent has a programming horizon of length k.

3.6 DATASET

We consider two coding agents on two benchmarks. For agents, we run `mini-swe-agent` v2.2.8 (Yang et al., 2024) with two state-of-the-art medium-sized models, `Qwen3.6-35B-A3B` (Qwen Team, 2026) and `Laguna-XS.2` (Abadji et al., 2026). Both models have a hidden-state dimension of $d = 2048$. Regarding benchmarks, we use `SWE-Bench-Verified` (Jimenez et al., 2024) (all 500 tasks) and `SWE-Bench-Pro` (Deng et al., 2025) (all 731 tasks).

For each task, we attempt to generate up to $n = 10$ trajectories on a single node with 8 A100 40GB GPUs. Agent hyperparameters, including limits, are listed in Table 2. For each trajectory, we record

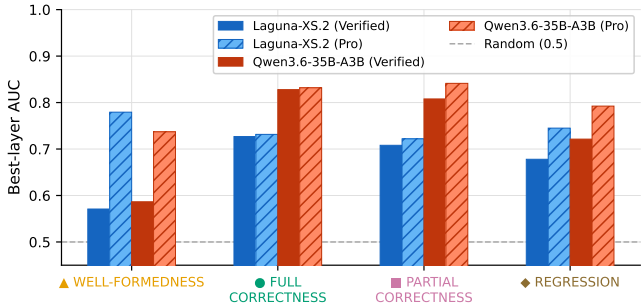

Figure 3: Best-layer AUC for each probe, model, and dataset. Each bar shows the peak AUC across layers. Probes trained on randomly shuffled labels in the unit of token collapse to 0.50, confirming that the signal is in the representation rather than an artefact of probe capacity. Full per-layer results are in Table 5 in the appendix.

the full program version after every tool call that modifies the codebase, and the results of evaluating the considered program properties.

For each trajectory, we record the hidden states every 5 tokens, for the sake of space. In total, we collect 22,714 trajectories with 79,480 edits and 22.4M hidden states (Table 1 in the appendix); trajectory length distributions (a median of 36k tokens per trajectory) and label prevalence per probe are shown in the Appendix (Figure 7). Table 1 clearly shows that our trajectories are multistep, with a median number of two and a mean of 3.5 edits per trajectory.

4 Results

4.1 Coding agents encode the current program

Figure 3 reports our first main finding. Every probe, on both models, is able to decode its corresponding program property well above chance. For example, considering ▲ Well-formedness, the best-trained probe is for `Laguna-XS.2`, with an AUC of 0.78 well above the random baseline of 0.5. In other words, the hidden state encodes properties of the program being edited.

Predicting semantic properties is generally stronger. The signal is strongest for ● Full Correctness (AUC up to 0.83) and ■ Partial Correctness (AUC up to 0.84) for `Qwen3.6-35B-A3B`. Similar results can be observed for `Laguna-XS.2`, where ● Full Correctness is the strongest on `SWE-Bench-Verified` (AUC up to 0.73) and where ◆ Regression reaches an AUC up to 0.75 on `SWE-Bench-Pro`. This means that the model's hidden states capture the semantic changes implied by every edit. This holds consistently across both benchmarks: the similar scores on `SWE-Bench-Verified` and `SWE-Bench-Pro` indicate the signal is not specific to a single task distribution.

▲ Well-formedness on `SWE-Bench-Verified` is a clear exception, with probes collapsing to near chance (AUC always below 0.60). This happens because trajectories in `SWE-Bench-Verified` contain mostly compilable Python programs (positive label rate above 0.92) – the considered models excel in this language. In the case of `SWE-Bench-Pro`, which includes three additional programming languages in its tasks (e.g. C++, TypeScript), and a far richer mix of compilation failures (positive rate between 0.52 and 0.57) in our training datasets, we observe higher training data balance and higher signal (AUC up to 0.78).

To sum up, as a coding agent edits a real codebase, its residual stream carries a linearly decodable representation of current program properties: whether it compiles, whether it passes its tests, and how its edits move the test suite relative to the initial repository. Those results have strong external validity, based on 22714 trajectories collected over 1231 different coding tasks.

Layer localization of program properties. Figure 4 shows a consistent inverted-U across layer depth for all probes except ▲ Well-formedness. Performance on program property prediction is

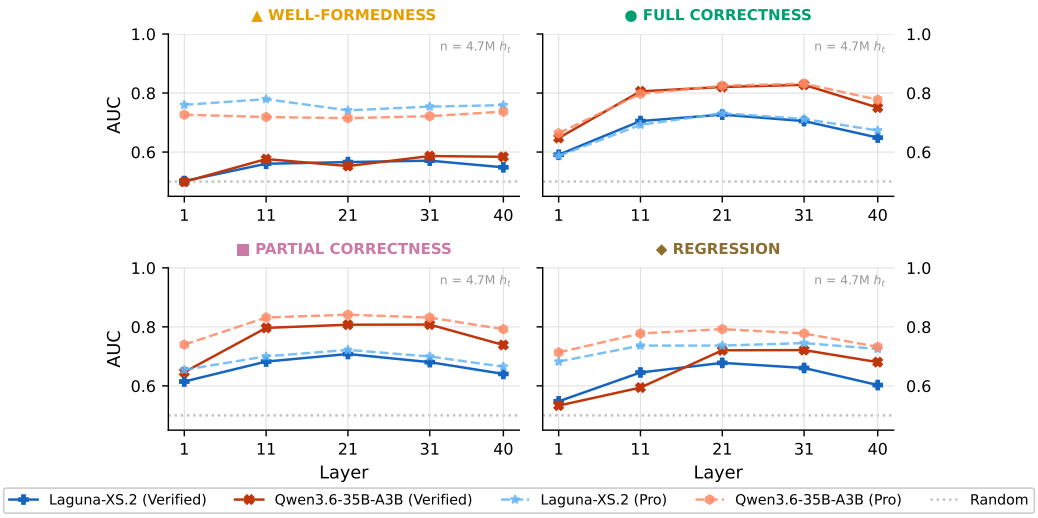

Figure 4: AUC-ROC by transformer layer at $k = 0$: ▲ WELL-FORMEDNESS (top left), ● FULL CORRECTNESS (top right), ■ PARTIAL CORRECTNESS (bottom left), ◆ REGRESSION (bottom right), for both models across both datasets. The inverted-U pattern, weakest at layer 1 and peaking mid-depth, holds for all probes and both models except ▲ WELL-FORMEDNESS.

lowest at the shallowest layer, rises to a peak at intermediate layers, and falls again slightly at the final layer. This holds across models and datasets, indicating that the program signal is most concentrated in the middle layers.

At layer 1, the probe is almost always the weakest of any depth, in some cases near or at the random baseline. The mid-layer peaks match previous results on natural language tasks, showing that intermediate layers often encode richer representations (Skean et al., 2025). While better than the first layer, the final layer shows a slight drop in signal, suggesting that final layers trade abstract features for features closer to the next token prediction task. Full per-layer results are in Table 5 in the appendix.

Model comparison. Per Figure 3, `Qwen3.6-35B-A3B` consistently encodes program properties more strongly than `Laguna-XS.2`. The gap is roughly 0.10 AUC on ● FULL CORRECTNESS and ■ PARTIAL CORRECTNESS across both benchmarks (0.83 vs. 0.73 and 0.81/0.84 vs. 0.71/0.72), and roughly 0.04–0.05 on ◆ REGRESSION. Crucially, the layer finding holds, with both models exhibiting an inverted-U (except for ▲ WELL-FORMEDNESS) peaking in the middle layers, and the final layer slightly below the peak. The two models therefore differ in how linearly decodable the program representation is but not in where it is encoded. This suggests that mechanistic interpretability tasks might be more appropriate on `Qwen3.6-35B-A3B` than with `Laguna-XS.2`.

Probes transfer across datasets. Figure 5 tests whether the probes trained on one benchmark can, without retraining, retrieve program properties from trajectories on another dataset. For both models, all semantic probes transfer substantially. For example, ● FULL CORRECTNESS and ■ PARTIAL CORRECTNESS retain AUC of 0.63–0.78 at the best layer under cross-dataset evaluation, compared to 0.71–0.84 in-distribution, a drop of only 0.04–0.09 units. These results confirm that our probes extract real signal related to program properties from hidden states, adding further evidence to the hypothesis that coding agents maintain latent program representations.

4.2 CODING AGENTS HAVE A LONG-TERM PROGRAMMING HORIZON

We have shown that the hidden state encodes the current properties of the program. Now, we demonstrate that it also encodes the properties of the future programs to come, that one can predict the outcome of future edits before those edits have materialized. We train each probe to predict a program property k steps in the future, $\hat{y}^{\phi}_{\sigma(t)+k} = \text{probe}(\mathbf{h}^{\ell}_{t})$, and sweep the horizon k from 0 (at the state to

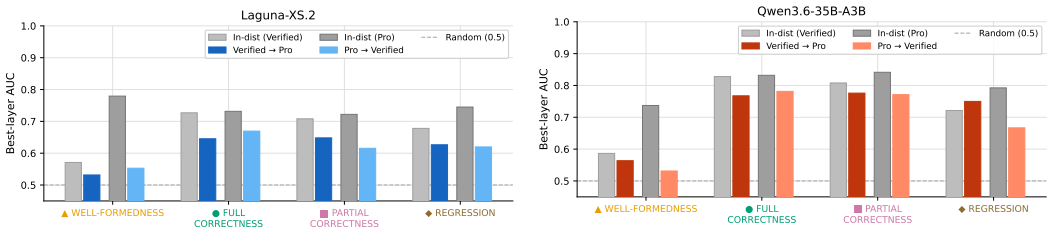

Figure 5: Cross-dataset transfer AUC at the best layer for each probe and model. Gray bars show in-distribution performance; colored bars show transfer performance without retraining. ● FULL CORRECTNESS and ■ PARTIAL CORRECTNESS transfer with small drops; ▲ WELL-FORMEDNESS collapses near chance due to the syntactic-failure-rate mismatch between benchmarks.

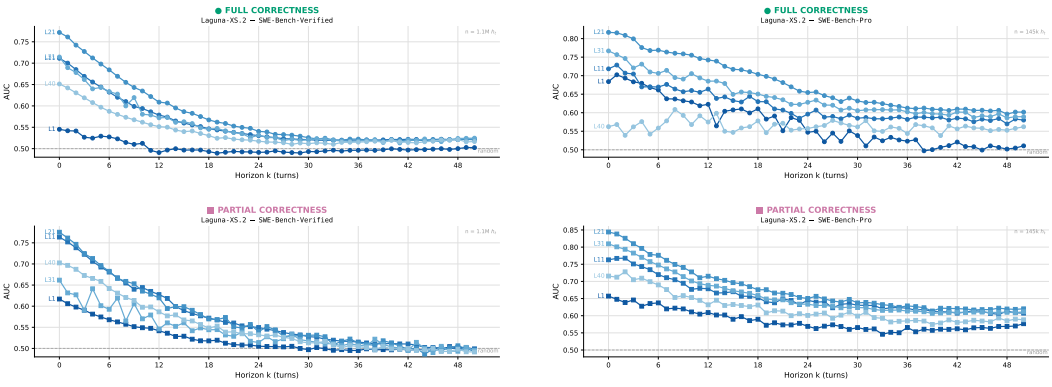

Figure 6: Lookahead AUC for ● FULL CORRECTNESS (top) and ■ PARTIAL CORRECTNESS (bottom) on `Laguna-XS.2`; `SWE-Bench-Verified` (left) and `SWE-Bench-Pro` (right). A probe at token position t predicts the property k steps later, $k \in [0, 50]$; each line is a layer, the dashed line is the 0.50 baseline. Full per-probe results are in Figures 8 and 10 in the appendix.

predict) out to 50 steps ahead. Figure 6 plots the two strongest properties (● FULL CORRECTNESS and ■ PARTIAL CORRECTNESS) on `Laguna-XS.2` across both benchmarks; each curve is a layer.

First, we see that the predictive signal decays smoothly with horizon (and drops most sharply after the initial edit which captures the present state of the program). At the edit ($k = 0$) the best-layer probe recovers ● FULL CORRECTNESS at AUC $\sim$0.77 on `SWE-Bench-Verified` and $\sim$0.82 on `SWE-Bench-Pro` (a slightly different figure than in Section 4.1, because this is a different test set). For almost all layers, prediction remains above 0.5 for at least 25 steps. As the horizon lengthens this falls, reaching $\sim$0.55 and $\sim$0.65 respectively around $k = 25$, before flattening into a plateau that persists out to $k = 50$. The plateau actually sits above the 0.50 control on both benchmarks ($\sim$0.52 on Verified, $\sim$0.60 on Pro): even 50 steps ahead, an edit's eventual outcome is weakly predictable. To our knowledge, this is the first evidence of long-term horizon by coding agents.

5 Related Work

Probing internal representations. Probing trains supervised classifiers on frozen hidden states to test for a target property (Alain & Bengio, 2017; Tenney et al., 2019; Belinkov, 2022). Linear probes recover the board state of an Othello-playing model (Li et al., 2023; Nanda et al., 2023) and the belief-state geometry of a data-generating process (Shai et al., 2024); Park et al. (2024) and Elhage et al. (2022) explain why many features remain linearly accessible. We follow best practice (Hewitt & Liang, 2019) and use limited-capacity linear probes compared against a shuffled-label control in the unit of token.

Predicting future states. Hidden states carry information about tokens not yet generated (Pal et al., 2023; Belrose et al., 2023; Lindsey et al., 2025; Pfau et al., 2024), future optimal moves in chess

(Jenner et al., 2024), and plans up to roughly 50 steps ahead in Sokoban (Taufeeque et al., 2024). During chain-of-thought, they encode whether the final answer will be correct (Zhang et al., 2025; Afzal et al., 2025; David, 2025; Cencerrado et al., 2025). We show the coding analogue, where the decoded outcome is an edit made many tool-use steps later.

Code representations in LLMs. Code models encode syntax, from constituency trees and full ASTs (Wan et al., 2022; Hernández López et al., 2022) to data and control-flow graphs (Ma et al., 2024), and probing suites cover their syntactic, structural, and semantic properties (Karmakar & Robbes, 2021; 2023; Troshin & Chirkova, 2022). Jin & Rinard (2024) recover current and future execution states of Karel programs with linear probes.

Correctness probing. Closest to us, recent work decodes whether a single generated function is correct from hidden states (Ribeiro et al., 2025; Bui et al., 2025; Vu et al., 2025), even before it is generated (Di Cicco, 2026), or studies correctness through attribution graphs (He et al., 2026), localized calibration (Gros & Devanbu, 2025), and sparse autoencoders (Tahimic & Cheng, 2025). These works study isolated snippets or single-step generation. We probe real-world programs as agents rewrite them over many steps, and are the first to predict the outcomes of future edits.

Latent program spaces. Other work explicitly builds latent program spaces, from neural program induction (Reed & De Freitas, 2015; Neelakantan et al., 2015) to continuous spaces searched for synthesis (Hong et al., 2021; Balog et al., 2020; Macfarlane & Bonnet, 2026; Macfarlane et al., 2026) and repair (Silva et al., 2025). We show that off-the-shelf LLMs acquire such representations as a byproduct of training, without any latent-space objective.

Coding agents. Coding agents such as SWE-agent (Yang et al., 2024), OpenHands (Wang et al., 2025), CodeAct (Wang et al., 2024), and AutoCodeRover (Zhang et al., 2024) let a language model edit and execute code, and are evaluated on SWE-bench (Jimenez et al., 2024), SWE-bench Pro (Deng et al., 2025), and Multi-SWE-bench (Zan et al., 2026). Failed trajectories are distinguishable from successful ones in token space (Bouzenia & Pradel, 2025), and Sui et al. (2026) probe agents' residual streams for early failure detection. Neither studies program properties in the latent space.

6 Limitations

Decodability is not causality. Our probes establish that program properties are linearly decodable from hidden states, not that the agent uses this information when acting. Showing causality requires steering along a probe direction and measuring the effect on the resulting edit, which we leave to future work.

Effect of label imbalance. ● Full Correctness and ■ Partial Correctness achieve the highest AUC, while ▲ Well-formedness achieves the lowest, because ▲ Well-formedness is overwhelmingly positive in our trajectories as today's models rarely emit invalid code. The hardest, most interesting properties to predict are the ones that flip often and take longest to resolve.

Scope. We study two open-weight models under a single agent scaffold (`mini-swe-agent`) on two benchmarks. We expect the same representations and horizons to appear in frontier models, other scaffolds, and other tasks, but this remains to be shown empirically.

7 Conclusion

Coding agents encode program properties in their residual streams. Linear probes decode whether code compiles, passes its tests, reduces failing tests, and introduces regressions, reaching AUC up to 0.83 for full correctness. This representation also grounds predictions about edits the agent has not yet written, remaining above chance up to 25 steps ahead. These results hold across two models and two benchmarks and transfer across datasets without retraining, opening new directions for monitoring and steering coding agents from within the latent space.

AI USE STATEMENT

We have used generative AI tools in the following ways. First, we have used coding agents for parts of the execution of our project. Second, we have used AI to adversarially reason about our experiments and the interpretation of their results. Third, we have used AI to polish writing of the paper, specifically to guarantee consistency between writing and data, in improving figures and tables, and in correcting grammatical mistakes.

REFERENCES

Julien Abadji, Marah Abdin, Connor Adams, Eric Alcaide, Mustafa Altun, Michele Artoni, Junze Bao, Uday Barar, Vassilis Bekiaris, Arkadii Bessonov, et al. Laguna m. 1/xs. 2 technical report. *arXiv preprint arXiv:2605.27605*, 2026.

Anum Afzal, Florian Matthes, Gal Chechik, and Yftah Ziser. Knowing before saying: Llm representations encode information about chain-of-thought success before completion. In *Findings of the Association for Computational Linguistics: ACL 2025*, pp. 12791–12806, 2025.

Guillaume Alain and Yoshua Bengio. Understanding intermediate layers using linear classifier probes. In *5th International Conference on Learning Representations, ICLR 2017, Workshop Track Proceedings*, 2017. URL https://openreview.net/forum?id=HJ4-rAVtl.

Matej Balog, Rishabh Singh, Petros Maniatis, and Charles Sutton. Neural program synthesis with a differentiable fixer. *arXiv preprint arXiv:2006.10924*, 2020.

Yonatan Belinkov. Probing classifiers: Promises, shortcomings, and advances. *Computational Linguistics*, 48(1):207–219, 2022.

Nora Belrose, Igor Ostrovsky, Lev McKinney, Zach Furman, Logan Smith, Danny Halawi, Stella Biderman, and Jacob Steinhardt. Eliciting latent predictions from transformers with the tuned lens. *arXiv preprint arXiv:2303.08112*, 2023.

Islem Bouzenia and Michael Pradel. Understanding software engineering agents: A study of thought-action-result trajectories. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 2846–2857. IEEE, 2025.

Tuan-Dung Bui, Thanh Trong Vu, Thu-Trang Nguyen, Son Nguyen, and Hieu Dinh Vo. Correctness assessment of code generated by large language models using internal representations. *Journal of Systems and Software*, pp. 112570, 2025.

Iván Vicente Moreno Cencerrado, Arnau Padrés Masdemont, Anton Gonzalvez Hawthorne, David Demitri Africa, and Lorenzo Pacchiardi. No answer needed: Predicting llm answer accuracy from question-only linear probes. *arXiv preprint arXiv:2509.10625*, 2025.

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.

Joey David. Temporal predictors of outcome in reasoning language models. *arXiv preprint arXiv:2511.14773*, 2025. URL https://arxiv.org/abs/2511.14773.

Xiang Deng, Jeff Da, Edwin Pan, Yannis Yiming He, Charles Ide, Kanak Garg, Niklas Lauffer, Andrew Park, Nitin Pasari, Chetan Rane, et al. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

Carlo Di Cicco. Code correctness signals in llm hidden states: Pre-generation probing and repair geometry. *arXiv preprint arXiv:2606.14530*, 2026.

Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, et al. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 1(1):12, 2021.

Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, et al. Toy models of superposition. *arXiv preprint arXiv:2209.10652*, 2022.

David Gros and Prem Devanbu. Localized calibrated uncertainty in code language models. *arXiv preprint arXiv:2512.24560*, 2025.

Yicheng He, Zheng Zhao, Zhou Kaiyu, Bryan Dai, Jie Fu, and Yonghui Yang. Codecircuit: Toward inferring llm-generated code correctness via attribution graphs. *arXiv preprint arXiv:2602.07080*, 2026.

José Antonio Hernández López, Martin Weyssow, Jesús Sánchez Cuadrado, and Houari Sahraoui. Ast-probe: Recovering abstract syntax trees from hidden representations of pre-trained language models. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–11, 2022.

John Hewitt and Percy Liang. Designing and interpreting probes with control tasks. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (emnlp-ijcnlp)*, pp. 2733–2743, 2019.

Joey Hong, David Dohan, Rishabh Singh, Charles Sutton, and Manzil Zaheer. Latent programmer: Discrete latent codes for program synthesis. In *International Conference on Machine Learning*, pp. 4308–4318. PMLR, 2021.

Erik Jenner, Shreyas Kapur, Vasil Georgiev, Cameron Allen, Scott Emmons, and Stuart Russell. Evidence of learned look-ahead in a chess-playing neural network. *Advances in Neural Information Processing Systems*, 37:31410–31437, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pp. 54107–54157, 2024.

Charles Jin and Martin Rinard. Emergent representations of program semantics in language models trained on programs. In *International Conference on Machine Learning*, pp. 22160–22184. PMLR, 2024.

Anjan Karmakar and Romain Robbes. What do pre-trained code models know about code? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1332–1336. IEEE, 2021.

Anjan Karmakar and Romain Robbes. Inspect: Intrinsic and systematic probing evaluation for code transformers. *IEEE Transactions on Software Engineering*, 50(2):220–238, 2023.

Kenneth Li, Aspen K Hopkins, David Bau, Fernanda Viégas, Hanspeter Pfister, and Martin Wattenberg. Emergent world representations: Exploring a sequence model trained on a synthetic task. *ICLR*, 2023.

Jack Lindsey, Wes Gurnee, Emmanuel Ameisen, Brian Chen, Adam Pearce, Nicholas L. Turner, Craig Citro, David Abrahams, Shan Carter, Basil Hosmer, Jonathan Marcus, Michael Sklar, Adly Templeton, Trenton Bricken, Callum McDougall, Hoagy Cunningham, Thomas Henighan, Adam Jermyn, Andy Jones, Andrew Persic, Zhenyi Qi, T. Ben Thompson, Sam Zimmerman, Kelley Rivoire, Thomas Conerly, Chris Olah, and Joshua Batson. On the biology of a large language model. *Transformer Circuits Thread*, 2025. URL https://transformer-circuits.pub/2025/attribution-graphs/biology.html.

Wei Ma, Shangqing Liu, Mengjie Zhao, Xiaofei Xie, Wenhang Wang, Qiang Hu, Jie Zhang, and Yang Liu. Unveiling code pre-trained models: Investigating syntax and semantics capacities. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–29, 2024.

Matthew Macfarlane and Clem Bonnet. Searching latent program spaces. *Advances in Neural Information Processing Systems*, 38:103463–103520, 2026.

Matthew Macfarlane, Clément Bonnet, Herke van Hoof, and Levi Lelis. Gradient-based program synthesis with neurally interpreted languages. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=NAORIWBaoO.

Neel Nanda, Andrew Lee, and Martin Wattenberg. Emergent linear representations in world models of self-supervised sequence models. In *Proceedings of the 6th BlackboxNLP Workshop: Analyzing and Interpreting Neural Networks for NLP*, pp. 16–30, 2023.

Arvind Neelakantan, Quoc V Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. *arXiv preprint arXiv:1511.04834*, 2015.

Koyena Pal, Jiuding Sun, Andrew Yuan, Byron C Wallace, and David Bau. Future lens: Anticipating subsequent tokens from a single hidden state. In *Proceedings of the 27th Conference on Computational Natural Language Learning (CoNLL)*, pp. 548–560, 2023.

Kiho Park, Yo Joong Choe, and Victor Veitch. The linear representation hypothesis and the geometry of large language models. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 39643–39666, 2024.

Jacob Pfau, William Merrill, and Samuel R. Bowman. Let's think dot by dot: Hidden computation in transformer language models. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=NikbrdtYvG.

Qwen Team. Qwen3.6-Plus: Towards real world agents, April 2026. URL https://qwen.ai/blog?id=qwen3.6.

Scott Reed and Nando De Freitas. Neural programmer-interpreters. *arXiv preprint arXiv:1511.06279*, 2015.

Francisco Ribeiro, Claudio Spiess, Prem Devanbu, and Sarah Nadi. On llms' internal representation of code correctness. *arXiv preprint arXiv:2512.07404*, 2025.

Adam S Shai, Sarah E Marzen, Lucas Teixeira, Alexander G Oldenziel, and Paul M Riechers. Transformers represent belief state geometry in their residual stream. *Advances in Neural Information Processing Systems*, 37:75012–75034, 2024.

André Silva, Gustav Thorén, and Martin Monperrus. Gradient-based program repair: Fixing bugs in continuous program spaces. *arXiv preprint arXiv:2505.17703*, 2025.

Oscar Skean, Md Rifat Arefin, Dan Zhao, Niket Nikul Patel, Jalal Naghiyev, Yann Lecun, and Ravid Shwartz-Ziv. Layer by layer: Uncovering hidden representations in language models. In *International Conference on Machine Learning*, pp. 55854–55875. PMLR, 2025.

Yuan Sui, Yulin Chen, Yibo Li, Xue Jiang, Yufei He, Yihong Dong, Xiaoxin He, Tianyu Gao, and Bryan Hooi. Tact: Mitigating overthinking and overacting in coding agents via activation steering. *arXiv preprint arXiv:2605.05980*, 2026.

Kriz Tahimic and Charibeth Cheng. Mechanistic interpretability of code correctness in llms via sparse autoencoders. *arXiv preprint arXiv:2510.02917*, 2025.

Mohammad Taufeeque, Philip Quirke, Maximilian Li, Chris Cundy, Aaron David Tucker, Adam Gleave, and Adrià Garriga-Alonso. Planning in a recurrent neural network that plays sokoban. *arXiv preprint arXiv:2407.15421*, 2024.

Ian Tenney, Dipanjan Das, and Ellie Pavlick. Bert rediscovers the classical nlp pipeline. In *Proceedings of the 57th annual meeting of the association for computational linguistics*, pp. 4593–4601, 2019.

Sergey Troshin and Nadezhda Chirkova. Probing pretrained models of source codes. In *Proceedings of the Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, pp. 371–383, 2022.

Thanh Trong Vu, Tuan-Dung Bui, Thu-Trang Nguyen, Son Nguyen, and Hieu Dinh Vo. Model-agnostic correctness assessment for llm-generated code via dynamic internal representation selection. *arXiv preprint arXiv:2510.02934*, 2025.

Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th international conference on software engineering*, pp. 2377–2388, 2022.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *Proceedings of the 41st International Conference on Machine Learning*, pp. 50208–50232, 2024.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. Openhands: An open platform for ai software developers as generalist agents. In *International Conference on Learning Representations*, volume 2025, pp. 65882–65919, 2025.

John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024. URL https://openreview.net/forum?id=mXpq6ut8J3.

He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–13, 2022.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Shulin Xin, Linhao Zhang, Qi Liu, Li Aoyan, Lu Chen, Xiaojian Zhong, et al. Multi-swe-bench: A multilingual benchmark for issue resolving. *Advances in Neural Information Processing Systems*, 38, 2026.

Anqi Zhang, Yulin Chen, Jane Pan, Chen Zhao, Aurojit Panda, Jinyang Li, and He He. Reasoning models know when they're right: Probing hidden states for self-verification. In *Second Conference on Language Modeling*, 2025. URL https://openreview.net/forum?id=O6I0Av7683.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604, 2024.

A Appendix

A.1 Dataset Statistics

Table 1: Dataset statistics per model and benchmark. #Traj. counts all agent runs; ≥50 counts trajectories reaching that step threshold (the length filter used in the lookahead experiments). Med. steps is the median number of steps per trajectory. #Edits is the total number of code edits across all trajectories. ≥2 edits counts trajectories with at least two edits. Med. edits is the median number of edits per trajectory. $\#h_t$ is the total number of collected hidden-state vectors.

Model	Benchmark	#Traj.	≥50 steps	Med. steps	#Edits	≥2 edits	Med. edits	$\#h_t$
`Laguna`	Verified	4,991	2,837	55	14,545	2,974	2	7.5M
	Pro	6,921	4,159	56	36,648	5,273	4	7.6M
`Qwen3.6`	Verified	4,998	2,001	39	9,730	2,161	1	3.8M
	Pro	5,804	3,186	54	18,557	3,682	2	3.5M
Total		22,714	12,183	52	79,480	14,090	2	22.4M

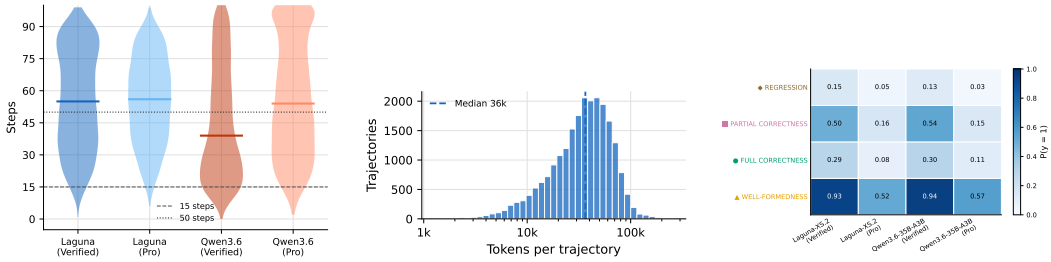

Figure 7: Left: distribution of trajectory lengths (in agent steps) per (model, dataset) combination. Dashed vertical lines mark the $k_{\max} = 15$ and $k_{\max} = 50$ lookahead thresholds; the former is nearly lossless (90–99% of trajectories), while the latter retains 40–60%. Middle: distribution of tokens per trajectory (median 36k). Right: positive label rate $P(y = 1)$ per probe and (model, dataset) combination. ▲ Well-formedness is nearly always positive, which suppresses its AUC despite a real signal being present (see Limitations); ● Full Correctness and ■ Partial Correctness are more balanced, consistent with their higher AUC.

Table 2: Agent hyperparameters used during trajectory generation, following the recommended settings provided by each model vendor. Temperature and top-p control sampling; context is the maximum sequence length; max steps is the step limit per trajectory; and step timeout is the wall-clock limit per tool call.

Model	Temp.	Top-p	Context	Max steps	Step timeout
`Laguna-XS.2`	0.7	0.95	256k	100	180 s
`Qwen3.6-35B-A3B`	0.6	0.95	256k	100	180 s

A.2 Probe Hyperparameters

A.3 Full Probe Results

Hyperparameter	**Distribution**	**Values**
Learning rate	Log-uniform	$[10^{-4},\ 10^{-1}]$
Weight decay	Log-uniform	$[10^{-5},\ 10^{-1}]$
Batch size	Categorical	$\{256,\ 512,\ 1024\}$
Patience	Categorical	$\{10,\ 25,\ 50\}$

Table 3: Hyperparameter search space used in the random search sweeps with 20 trials each. The best configurations, maximising validation AUC per probe per layer per model per dataset are used for training the final probes.

Table 4: Chosen hyperparameters for each model and benchmark, selected by 20-trial random search maximising mean validation AUC. Sweeps were run independently per probe and per layer; table shows the middle probed layer as representative.

Dataset	Model	Probe	Learning rate	Weight decay	Batch size	Patience
VERIFIED	Laguna-XS.2	▲ WELL-FORMEDNESS	8.58×10^{-3}	1.18×10^{-3}	512	50
		● FULL CORRECTNESS	5.19×10^{-4}	3.81×10^{-3}	512	10
		◆ REGRESSION	2.95×10^{-2}	5.19×10^{-5}	256	50
		■ PARTIAL CORRECTNESS	2.66×10^{-2}	2.13×10^{-3}	1024	50
	Qwen3.6-35B-A3B	▲ WELL-FORMEDNESS	3.33×10^{-3}	4.20×10^{-2}	512	25
		● FULL CORRECTNESS	1.17×10^{-2}	4.10×10^{-4}	1024	25
		◆ REGRESSION	3.11×10^{-4}	1.91×10^{-4}	512	50
		■ PARTIAL CORRECTNESS	3.39×10^{-3}	1.75×10^{-4}	1024	10
PRO	Laguna-XS.2	▲ WELL-FORMEDNESS	1.40×10^{-3}	3.93×10^{-4}	512	25
		● FULL CORRECTNESS	2.09×10^{-4}	6.05×10^{-4}	256	25
		■ PARTIAL CORRECTNESS	1.52×10^{-3}	1.64×10^{-3}	1024	50
		◆ REGRESSION	1.13×10^{-4}	1.49×10^{-2}	1024	10
	Qwen3.6-35B-A3B	▲ WELL-FORMEDNESS	2.30×10^{-4}	1.46×10^{-5}	512	10
		● FULL CORRECTNESS	4.15×10^{-4}	2.27×10^{-4}	1024	10
		◆ REGRESSION	2.74×10^{-3}	2.00×10^{-4}	512	50
		■ PARTIAL CORRECTNESS	1.26×10^{-3}	8.97×10^{-5}	1024	50

Table 5: AUC per probe, model, and benchmark across transformer layers. **Bold** marks the best layer per row. *Shuffled* uses label-permuted data as a sanity baseline.

	AUC ↑					Shuffled
Probe	1	11	21	31	40	(best layer)
SWE-bench Verified						
Laguna-XS.2						
▲ Well-formedness	0.502	0.560	0.566	**0.571**	0.548	0.500
● Full Correctness	0.591	0.705	**0.727**	0.705	0.649	0.501
■ Partial Correctness	0.615	0.682	**0.708**	0.680	0.641	0.500
♦ Regression	0.548	0.645	**0.678**	0.661	0.603	0.500
Qwen3.6-35B-A3B						
▲ Well-formedness	0.499	0.576	0.553	**0.587**	0.584	0.501
● Full Correctness	0.648	0.806	0.820	**0.828**	0.751	0.501
■ Partial Correctness	0.646	0.796	0.807	**0.808**	0.739	0.501
♦ Regression	0.533	0.594	0.721	**0.721**	0.681	0.501
SWE-bench Pro						
Laguna-XS.2						
▲ Well-formedness	0.760	**0.779**	0.741	0.754	0.759	0.501
● Full Correctness	0.587	0.692	**0.732**	0.712	0.673	0.500
■ Partial Correctness	0.655	0.700	**0.722**	0.700	0.665	0.500
♦ Regression	0.682	0.736	0.737	**0.745**	0.725	0.501
Qwen3.6-35B-A3B						
▲ Well-formedness	0.727	0.719	0.715	0.722	**0.737**	0.502
● Full Correctness	0.664	0.797	0.825	**0.832**	0.777	0.502
■ Partial Correctness	0.740	0.832	**0.841**	0.831	0.792	0.502
♦ Regression	0.713	0.778	**0.792**	0.778	0.732	0.507

Table 6: Cross-dataset transfer AUC. Gray rows show in-distribution reference performance. Transfer rows show AUC when probe weights trained on one dataset are evaluated on the other; the subscript shows the delta relative to the in-distribution baseline on the *same* evaluation set.

	Layer 1	Layer 11	Layer 21	Layer 31	Layer 40
▲ WELL-FORMEDNESS					
Laguna-XS.2					
In-dist (Verified)	0.502	0.560	0.566	0.571	0.548
In-dist (Pro)	0.760	0.779	0.741	0.754	0.759
Verified → Pro	0.508 −0.251	0.527 −0.252	0.532 −0.210	0.523 −0.231	0.508 −0.251
Pro → Verified	0.484 −0.018	0.521 −0.040	0.553 −0.013	0.528 −0.043	0.498 −0.050
Qwen3.6-35B-A3B					
In-dist (Verified)	0.499	0.576	0.553	0.587	0.584
In-dist (Pro)	0.727	0.719	0.715	0.722	0.737
Verified → Pro	0.564 −0.163	0.516 −0.202	0.516 −0.199	0.516 −0.206	0.480 −0.257
Pro → Verified	0.509 +0.010	0.527 −0.049	0.531 −0.021	0.503 −0.083	0.447 −0.137
● FULL CORRECTNESS					
Laguna-XS.2					
In-dist (Verified)	0.591	0.705	0.727	0.705	0.649
In-dist (Pro)	0.587	0.692	0.732	0.712	0.673
Verified → Pro	0.502 −0.085	0.630 −0.062	0.645 −0.086	0.643 −0.069	0.542 −0.132
Pro → Verified	0.531 −0.060	0.634 −0.071	0.669 −0.057	0.636 −0.069	0.594 −0.055
Qwen3.6-35B-A3B					
In-dist (Verified)	0.648	0.806	0.820	0.828	0.751
In-dist (Pro)	0.664	0.797	0.825	0.832	0.777
Verified → Pro	0.615 −0.048	0.730 −0.068	0.748 −0.076	0.767 −0.064	0.702 −0.076
Pro → Verified	0.587 −0.061	0.764 −0.042	0.781 −0.039	0.778 −0.049	0.699 −0.052
■ PARTIAL CORRECTNESS					
Laguna-XS.2					
In-dist (Verified)	0.615	0.682	0.708	0.680	0.641
In-dist (Pro)	0.655	0.700	0.722	0.700	0.665
Verified → Pro	0.519 −0.136	0.622 −0.078	0.648 −0.074	0.625 −0.075	0.556 −0.109
Pro → Verified	0.517 −0.097	0.574 −0.109	0.615 −0.093	0.577 −0.104	0.551 −0.090
Qwen3.6-35B-A3B					
In-dist (Verified)	0.646	0.796	0.807	0.808	0.739
In-dist (Pro)	0.740	0.832	0.841	0.831	0.792
Verified → Pro	0.623 −0.117	0.774 −0.058	0.773 −0.068	0.776 −0.055	0.720 −0.072
Pro → Verified	0.605 −0.041	0.754 −0.042	0.766 −0.041	0.771 −0.037	0.699 −0.040
♦ REGRESSION					
Laguna-XS.2					
In-dist (Verified)	0.548	0.645	0.678	0.661	0.603
In-dist (Pro)	0.682	0.736	0.737	0.745	0.725
Verified → Pro	0.459 −0.223	0.500 −0.237	0.626 −0.110	0.618 −0.127	0.540 −0.185
Pro → Verified	0.560 +0.012	0.612 −0.033	0.620 −0.058	0.601 −0.059	0.560 −0.043
Qwen3.6-35B-A3B					
In-dist (Verified)	0.533	0.594	0.721	0.721	0.681
In-dist (Pro)	0.713	0.778	0.792	0.778	0.732
Verified → Pro	0.575 −0.139	0.593 −0.185	0.750 −0.042	0.724 −0.053	0.641 −0.091
Pro → Verified	0.557 +0.024	0.664 +0.070	0.667 −0.054	0.661 −0.060	0.613 −0.067

A.4 LOOKAHEAD ANALYSIS

The main lookahead experiment uses a maximum k of 50, which excludes tokens within 50 steps of the end of a trajectory, including all trajectories shorter than 50 steps. Figure 9 replicates the analysis at $k \leq 15$, which imposes a smaller constraint. The monotonic rise in AUC as $k \to 0$ and

the layer-ordering are consistent with the main results, confirming that the hockey-stick pattern is not an artifact of the reduced sample.

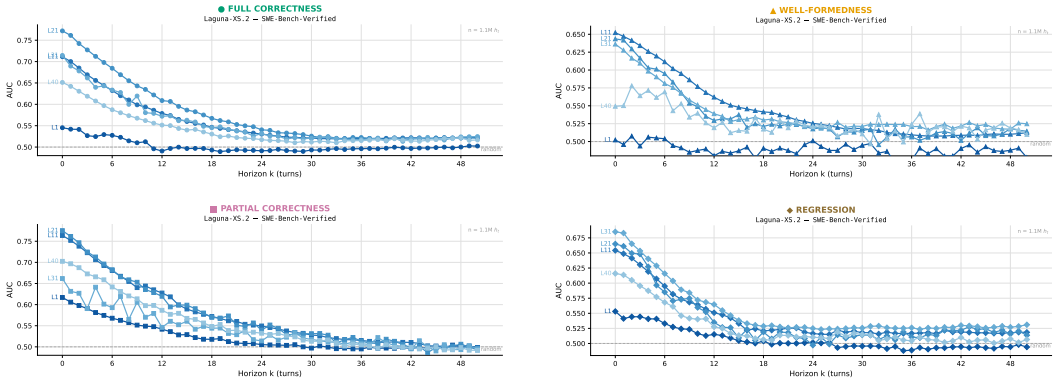

Figure 8: Lookahead AUC for all four probes on `Laguna-XS.2` (`SWE-Bench-Verified`), $k \leq 50$. Panels show ● FULL CORRECTNESS (top left), ▲ WELL-FORMEDNESS (top right), ■ PARTIAL CORRECTNESS (bottom left), and ◆ REGRESSION (bottom right). A probe at token position t predicts the program property k steps later; each line is a transformer layer; the dashed line marks the 0.50 random baseline. AUC rises as $k \to 0$ across all probes.

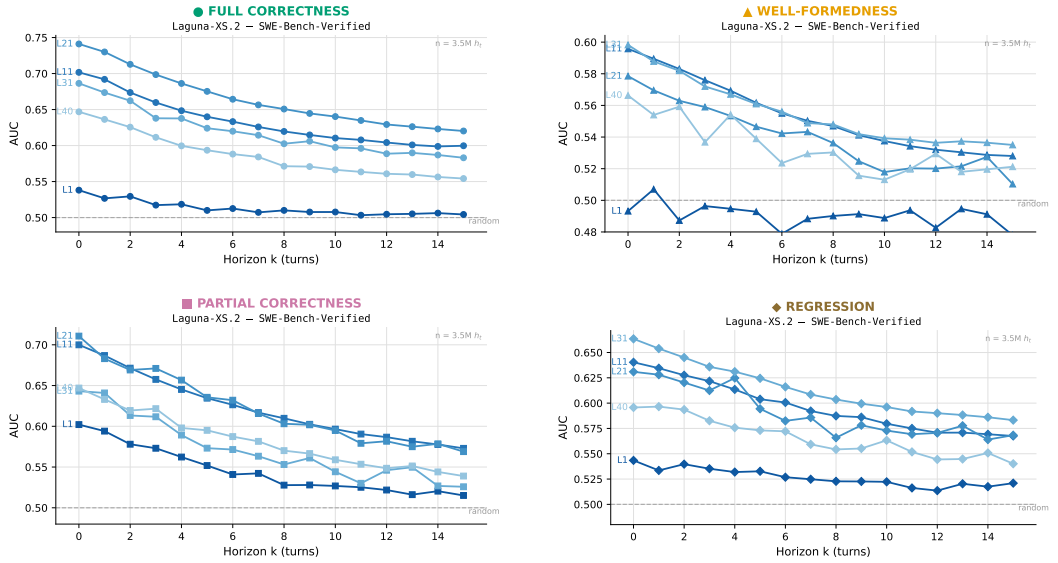

Figure 9: Lookahead AUC for all four probes on `Laguna-XS.2` (`SWE-Bench-Verified`), $k \leq 15$, using the full trajectory sample without the length filter required for $k \leq 50$. Panel layout matches Figure 8. The monotonic rise in AUC as $k \to 0$ and the layer ordering replicate the $k \leq 50$ results, confirming the hockey-stick pattern is not an artifact of the trajectory length filter.

A.5 EXAMPLE LATENT TRAJECTORIES

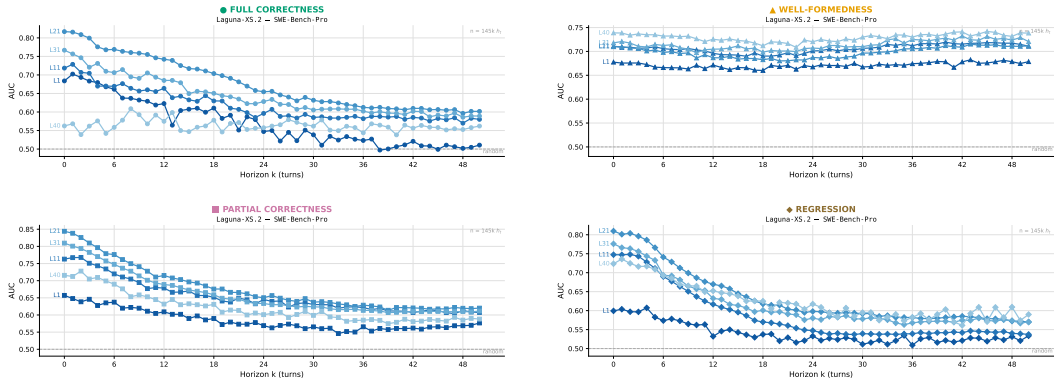

Figure 10: Lookahead AUC for all four probes on `Laguna-XS.2` (`SWE-Bench-Pro`), $k \leq 50$. Panel layout matches Figure 8. The hockey-stick pattern replicates across this harder, multi-language benchmark.

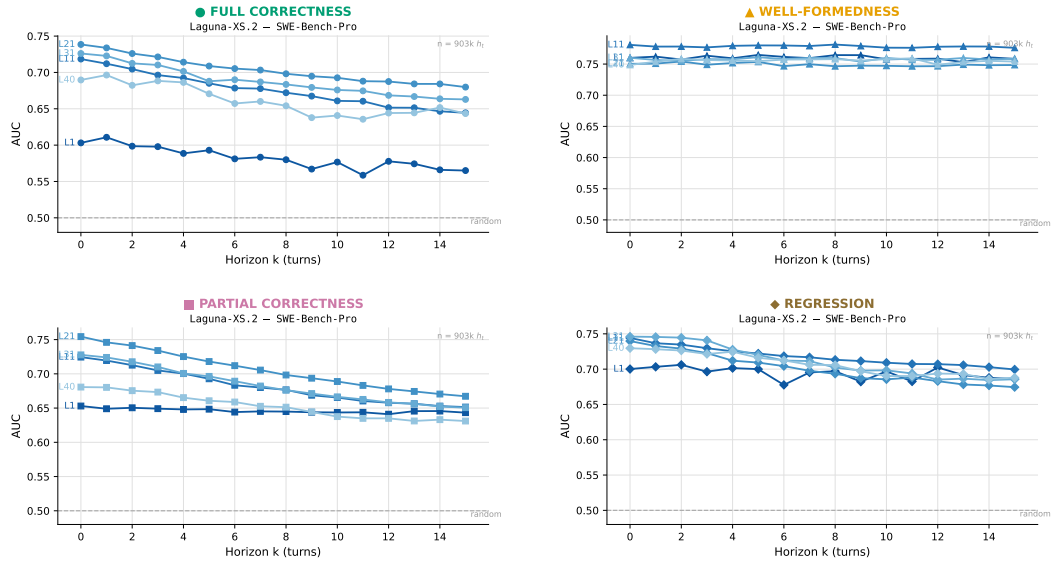

Figure 11: Lookahead AUC for all four probes on `Laguna-XS.2` (`SWE-Bench-Pro`), $k \leq 15$, using the full trajectory sample without the length filter. Panel layout matches Figure 8. Results are consistent with the $k \leq 50$ analysis on this benchmark.

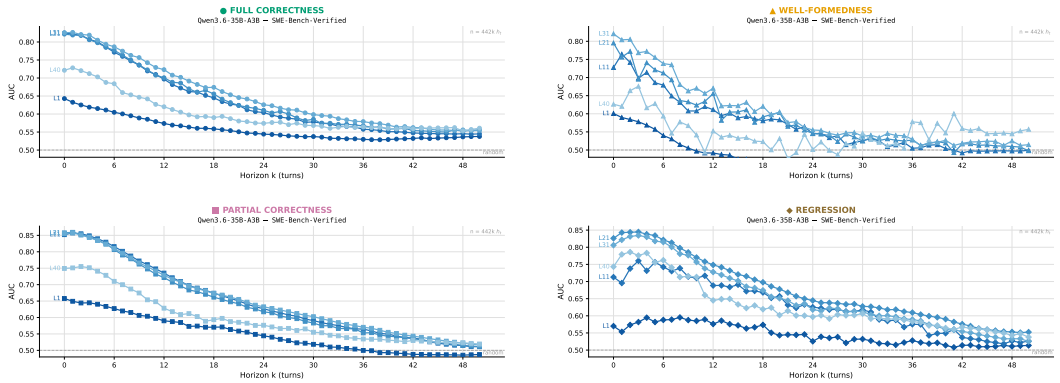

Figure 12: Lookahead AUC for all four probes on `Qwen3.6-35B-A3B` (`SWE-Bench-Verified`), $k \leq 50$. Panel layout matches Figure 8. The hockey-stick pattern and layer ordering replicate at larger model scale.

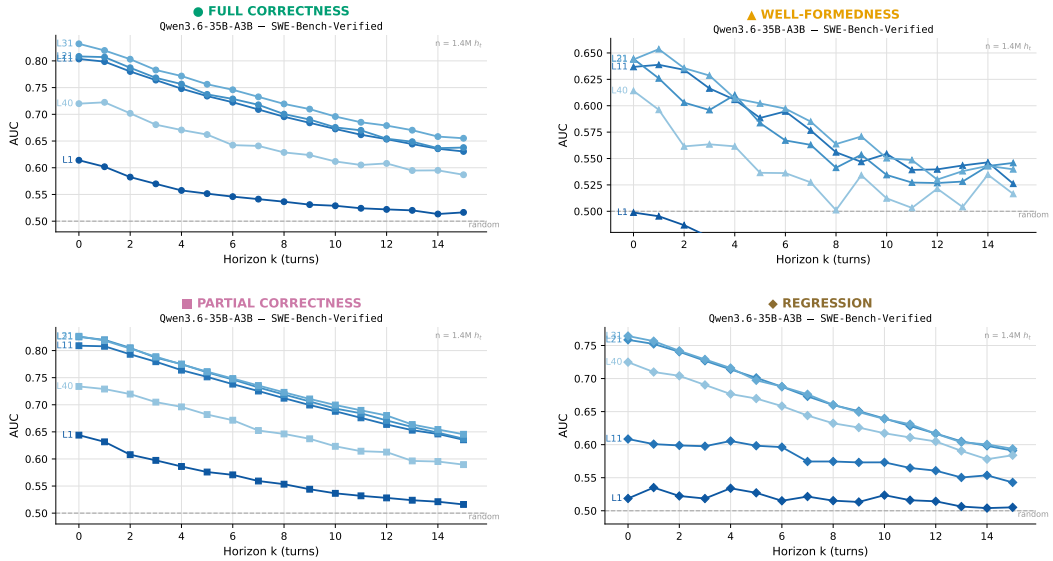

Figure 13: Lookahead AUC for all four probes on `Qwen3.6-35B-A3B` (`SWE-Bench-Verified`), $k \leq 15$, using the full trajectory sample without the length filter. Panel layout matches Figure 8. Results are consistent with the $k \leq 50$ analysis for this model.

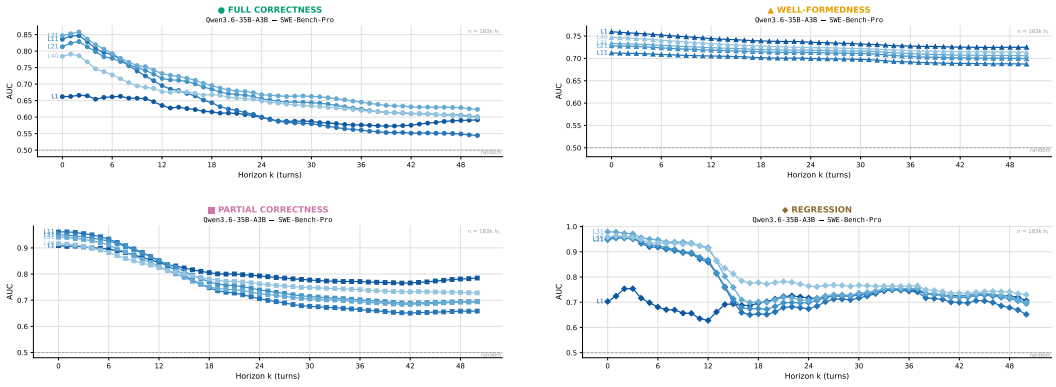

Figure 14: Lookahead AUC for all four probes on `Qwen3.6-35B-A3B` (`SWE-Bench-Pro`), $k \leq 50$. Panel layout matches Figure 8. The hockey-stick pattern and layer ordering replicate at larger model scale on this harder, multi-language benchmark.

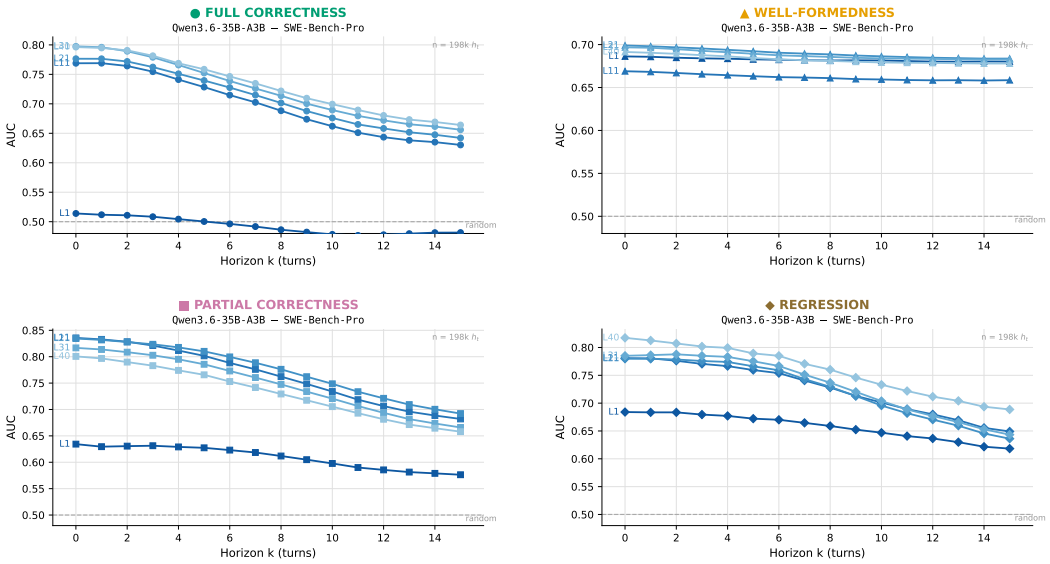

Figure 15: Lookahead AUC for all four probes on `Qwen3.6-35B-A3B` (`SWE-Bench-Pro`), $k \leq 15$, using the full trajectory sample without the length filter. Panel layout matches Figure 8. Results are consistent with the $k \leq 50$ analysis on this benchmark.

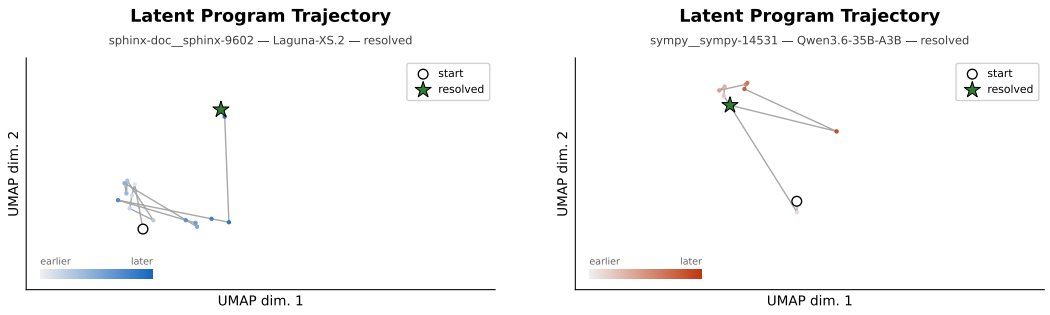

Figure 16: Two example resolved trajectories, hand-selected from a small set of long (many-edit), resolved rollouts for legibility, shown at layer 21 (best or near-best by probe validation AUC for both models). Left: `Laguna-XS.2` on `sphinx-doc/sphinx-9602`. Right: `Qwen3.6-35B-A3B` on `sympy/sympy-14531`. Each trajectory is reduced from its dense, stride-5 token-level activations to one point per code edit plus a start and an end marker, obtained by mean-pooling all captured hidden states within each corresponding span (before the first edit, between consecutive edits, and after the last edit). Color encodes step order (light = earlier, dark = later) in each model's palette color; the white circle marks the start and the star marks the final, resolved state. These examples are illustrative only and are not intended as quantitative evidence.