
EnvTrace: Simulation-Based Semantic Evaluation of LLM Code via Execution Trace Alignment—Demonstrated at Synchrotron Beamlines

Noah van der Vleuten

Center for Functional Nanomaterials
Brookhaven National Laboratory
Upton, NY 11973, USA

Anthony Flores

Center for Functional Nanomaterials
Brookhaven National Laboratory
Upton, NY 11973, USA

Shray Mathur

Center for Functional Nanomaterials
Brookhaven National Laboratory
Upton, NY 11973, USA

Max Rakitin

National Synchrotron Light Source II
Brookhaven National Laboratory
Upton, NY 11973, USA

Thomas Hopkins

National Synchrotron Light Source II
Brookhaven National Laboratory
Upton, NY 11973, USA

Kevin G. Yager

Center for Functional Nanomaterials
Brookhaven National Laboratory
Upton, NY 11973, USA

Esther Tsai

Center for Functional Nanomaterials
Brookhaven National Laboratory
Upton, NY 11973, USA
etsai@bnl.gov

ABSTRACT

Evaluating large language models (LLMs) for instrument control requires methods that go beyond standard, stateless algorithmic benchmarks, since the behavior of physical systems cannot be fully captured by unit tests alone. Here we introduce EnvTrace, a simulation-based method that evaluates execution traces to assess semantic code equivalence. EnvTrace is demonstrated with a beamline control-logic digital twin to facilitate the evaluation of instrument control code, with the digital twin itself also enabling the pre-execution validation of live experiments. Over 30 LLMs were evaluated using trace alignment to generate a multi-faceted score for functional correctness across key behavioral dimensions, showing that many top-tier models can approach human-level performance in rapid control-code generation. This is a first step toward a broader vision where LLMs and digital twins work symbiotically: LLMs providing intuitive control and agentic orchestration, and digital twins offering safe and high-fidelity environments, paving the way towards autonomous embodied AI.

1 Introduction

The advent of large language models (LLMs) has catalyzed a paradigm shift in artificial intelligence, moving beyond task-specific applications to powering autonomous agents capable of interacting with complex digital and physical environments. This new generation of "physical AI" holds immense promise for advancing robotics, industrial control, and laboratory automation by translating high-level human intent into executable code. However, the transition from

virtual tasks to real-world action exposes a critical challenge: ensuring that the generated code is not merely syntactically valid, but functionally correct and safe. An error in a chatbot is an inconvenience; an error in controlling a costly and scarce scientific instrument can cause severe equipment damage and result in the loss of invaluable experimental time.

While the use of LLMs for code generation [1, 2] is growing, evaluation metrics are continually evolving in an effort to more accurately measure correctness. When at least one ground-truth example is available, reference-based evaluation metrics, such as BLEU [3] or variants CrystalBLEU [4], often fail to represent functional correctness and can thus be misleading. CodeBLEU [5] leverages the n-gram matching capability of BLEU while also incorporating code syntax through abstract syntax trees (AST) and capturing code semantics via data-flow analysis. CodeBERTScore is a metric for evaluating code generation by leveraging pretrained models to independently encode both the generated and reference code, along with optional natural language context [6]. However, these reference-based approaches, whether with tokens or embeddings, can still be misleading when the goal is to assess true functional correctness [2, 7]. Without the need for unit tests and reference code, metrics that use LLMs as judges can provide improved correlations with functional correctness and human preferences [8, 9]. Functional correctness is assessed by executing generated code and checking if its outputs match expectations, ensuring alignment with human judgment and real-world relevance. This is most commonly done through unit tests, which compare the program’s output against predefined results for given inputs. Such approaches are standard in coding competitions, where human-written code is validated using a combination of public and hidden test cases. Pass@ k gives the expected fraction of problems for which at least one of k independently sampled completions passes all unit tests [10]. Even execution-based benchmarks like APPS [11, 12], which verify generated code through hidden test cases, are limited to stateless, algorithmic tasks and cannot capture more complex dynamics or continuous interaction with a changing environment. SWE-Bench [13] evaluates the LLM ability to reason and modify the codebase for realistic GitHub issues, while AgentBench [14] benchmarks LLMs on their reasoning and decision-making abilities as agents in multi-turn, open-ended real-world environments. These evaluations reflect a growing shift toward execution- and interaction-based methods for assessing LLMs.

Providing a sufficiently realistic execution environment for code evaluation requires the development and use of practical mechanisms such as mocking frameworks or digital twins. Digital twins have become increasingly central to scientific and industrial applications, serving as high-fidelity virtual representation of physical systems used for simulation, monitoring, and optimization [15, 16, 17]. Developing digital twins is complex and time-intensive, requiring expert-crafted models while facing challenges such as data sparsity and trade-offs between accuracy, efficiency, and interpretability in physics-based and data-driven approaches. Evaluating digital twins remains challenging owing to their dynamic behavior and the multi-faceted nature of assessing their fidelity and performance, prompting studies to investigate various strategies. Ershenko et al. [18] conducted a comparative study of error-based metrics on a digital twin of a railway braking system, showing that the appropriate metric is task specific: aggregate percentage-based measures such as mean absolute percentage error (MAPE) are well suited for judging overall fidelity, while instantaneous error measures like absolute percentage error (APE) are critical for triggering timely interventions. Complementing this metric-based perspective, Muñoz et al. [19] introduced a trace-alignment approach to assess behavioral similarity between digital and physical twins using offline execution traces. Muñoz et al. [20] also demonstrated runtime fidelity monitoring through continuous trace comparison. Trace-alignment methods have also been widely explored in process mining, where techniques derived from sequence-alignment are used to measure deviations between observed and expected behavior [21, 22]. Together, these studies underscore the importance of dynamic, behavior-based evaluation for digital twins. Our work extends such behavioral evaluation to the domain of code generation, assessing the functional correctness of LLM-generated control logic through its interaction with a simulated environment.

While digital twins provide realistic, physics-based environments for evaluation, synergistically LLMs can also enable efficient control and design of digital twins, progressing toward agentic orchestration of experiments. Advances in LLMs enable automated code generation for simulations, control, and data pipelines, reducing manual effort and enhancing the adaptability and scalability of digital twins [23, 24, 25, 26]. LLM-driven digital twins or experimentation show strong potential for complex system reasoning, but deploying them safely in critical domains requires strict safety measures, expert oversight, and tailored evaluation methods. Recently, Hellert et al. [27, 28] presented an agentic system for a multi-stage physics experiment at a synchrotron light source storage ring to significantly reduce the preparation time. Chen et al. [29] demonstrated an agentic AI workflow capable of conducting experiments, including planning, execution, analysis, and iterative refinement to achieve a scientific objective in simulation (with calculated diffraction intensity) and also experimentally. The work further emphasizes the need for simulated environments to safely develop and deploy agentic systems. Other works have also explored the incorporation of language models for physical experiments [30, 31, 32, 33, 34], highlighting the increasing interest in agentic experimental workflows in science.

We have previously demonstrated a modular AI assistant VISION [31] that can voice-control synchrotron [35] beamlines by generating Python code from natural language queries. Although we explored quantitative evaluation based on existing code similarity metrics, including exact string matching, Levenshtein distance, and CodeBLEU [5], these

syntactic metrics are limited in capturing functional correctness, as they often penalize correct code with differing structure or style. For example, a `for` loop may be penalized when written in place of an equivalent `while` loop; semantic bugs can go undetected when the structure is correct but the parameters are wrong, such as moving a motor to $+10$ rather than -10 . This gap between syntactic similarity and functional correctness is the primary barrier to the safe and reliable deployment of LLM agents in high-stakes physical systems. To address these limitations, here we present EnvTrace¹, a semantic code evaluation framework that focuses on assessing functional correctness and runtime performance based on a program’s environmental effects, rather than its syntax. Leveraging a control-logic digital twin, the framework evaluates code behavior, with outcomes cross-validated against human judgment to ensure alignment. The digital twin is deployed within a secure, sandboxed environment, providing a controlled and risk-free setting.

We detail the mock simulation environment and quantitative multi-faceted evaluation in Section 2, evaluation of coding outcomes for multiple open- and closed-source LLMs in Section 3, discussion and summary in Sections 4 and 5, respectively.

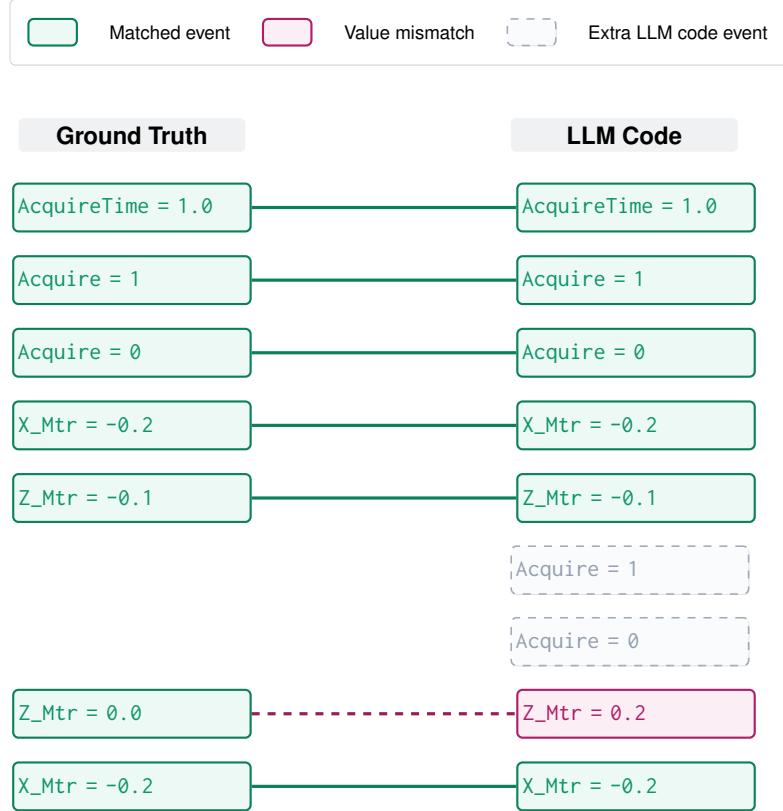


Figure 1: Illustration of the EnvTrace alignment process and score calculation. The ground-truth and LLM code (predicted code) execution traces are aligned based on their Process Variable (PV) changes. Matched events are shown by connected solid lines, value mismatches by dashed lines, and extra predicted events are shown in dashed box. This alignment is then used for computing scores to evaluate the LLMs.

2 Methodology

EnvTrace overcomes the limitations of syntactic code evaluation by assessing the semantic and functional correctness of LLM-generated code in a simulated physical environment. It involves the execution of both the ground-truth (reference) code and the LLM-generated code within the mock environment or simulator. An execution trace, which is a time-ordered log of all state changes to the environment’s parameters, is captured for each code. By systematically aligning and comparing these traces, as shown in Fig. 1, we can determine if the two pieces of code produced the same functional outcome, executed in the correct sequence, and with correct timing. While Chen et al. [29] utilizes calculated

¹<https://github.com/CFN-softbio/EnvTrace>

diffraction intensity for instrument feedback, our work emphasizes tracking simulated physical instruments, offering a valuable complementary perspective. Here we demonstrate and validate EnvTrace at synchrotron beamlines [35], where extremely intense X-ray enables studies in e.g. microelectronics [36, 37, 38], energy and material science [39, 40], and biomedical science [41, 42]. The beamline environment is a quintessential cyber-physical system: it is highly stateful, where actions have persistent effects; it is real-time, where the timing and pacing of operations are critical; and its control involves a complex interplay of discrete actions (e.g., motor movements, temperature targets, shutter triggers) and continuous processes (e.g., temperature ramps). The high-stakes nature of synchrotron experimentation, due to the limited experiment time and expensive instrumentation, makes it a powerful and representative domain for developing evaluation methods that can be generalized to other physical AI systems.

The simulation-based evaluation paradigm offers several key advantages over static code analysis. First and foremost, it enables a true functional equivalence comparison. Instead of a binary pass/fail score from an exact string match or a similarity score from CodeBLEU, this approach provides a granular, multi-dimensional evaluation that compares code on an action-by-action basis. Second, the framework provides scientists and developers with a valuable resource for debugging code, exploring alternative implementations, and validating ground-truth examples, thereby strengthening the testbed. Third, the simulator enables "pre-flight check" of experimental plans, allowing researchers to detect errors in complex sequences before using limited and costly experiment time, e.g. beamtime at X-ray or neutron facilities. This highlights the value of robust digital twins for developing safe and efficient agent-based systems in scientific research.

2.1 Simulation Environment

To enable execution-based evaluation without risking physical hardware or consuming scarce beamtime, we developed a control-logic beamline simulator, practical for realistic instrument-code evaluation. The simulator provides the actual beamline environment through scripts defined in the Python-based Bluesky data acquisition framework [43]. These scripts not only specify the basic Bluesky configurations, but also capture beamline-specific devices and measurement protocols, enabling faithful reproduction of experimental workflows in a simulated setting. As Bluesky continues to be adopted across synchrotron facilities, such a simulator may gain in broad applicability and suitability for authentic assessment of instrument-code.

An overview of the EnvTrace implementation for beamline code evaluation is provided in Fig. 2. Two sets of code, ground-truth/reference and human- or LLM-generated, are executed at the simulated beamline. During execution, the monitoring system logs the states of beamline components to produce a multi-faceted score that quantifies the functional similarity between the two. This control-logic digital twin replicates the control infrastructure of the beamline by modeling relevant components with configurable physical accuracy, including motors, thermal stage, and detectors, while ignoring non-essential commands by mocking them through a black hole IOC (Input-Output Controller). As shown in Fig. 2, the simulator is composed of two subsystems: the interactive control session and the simulated Experimental Physics and Industrial Control System (EPICS) [44] services. Our simulator replicates this using a set of Docker containers and Python scripts, each running an EPICS IOC. The fidelity of our control-logic digital twin is specifically tailored for validating control logic rather than performing a full physics-based simulation. The IOCs achieve this by providing two key levels of abstraction: (1) Interface Replication: They expose the exact same EPICS Process Variables (PVs) as the physical hardware. (2) State Tracking: They maintain a persistent internal state for critical components. For example, the simulated motor IOCs track and update their position values after a move command, ensuring that subsequent commands operate on the correct state. The simulation does not, however, reproduce more complex realistic behaviors such as generating synthetic detector images or modeling the precise thermodynamics of a temperature ramp. This level of functional, state-aware abstraction is sufficient for verifying the correctness of command sequences, parameters, and timing. The mock framework executes code within a sandboxed IPython session, communicating with simulated devices via the standard EPICS Channel Access network protocol. This ensures that, from the control software's perspective, interactions with the simulated environment are functionally indistinguishable from those with real beamline hardware, allowing the same code to run in both simulated and live instrument control environments. The detailed EnvTrace architecture is provided in Fig. S1, with the implementation for the beamline control-logic simulator available at: <https://github.com/CFN-softbio/VISION>.

In this work, we demonstrate EnvTrace within the beamline environment at the National Synchrotron Light Source II (NSLS-II) 11-BM Complex Materials Scattering (CMS) beamline. The simulator can run the same Python code used in actual beamline operations without modification, making it easily adaptable for deployment at other beamlines. This architecture generates realistic sequences of state changes that form the basis of our evaluation.

²The black hole image is courtesy of the Event Horizon Telescope Collaboration [45] (CC BY 4.0).

2.2 Multi-Faceted Evaluation

Built upon this simulation environment, EnvTrace provides a methodology for assessing the semantic equivalence of code snippets by comparing their execution traces. An execution trace is a time-ordered sequence of state changes observed in the environment. Each entry in the trace is a tuple containing the Process Variable's (PV) name, its new value, and a high-precision timestamp: (pv_name, value, timestamp). The process begins when a code snippet, either ground-truth or LLM-generated, is passed to the evaluation framework, as illustrated in Fig. 2. The code is injected into the simulated IPython session, where its execution modifies the state of the simulated IOCs. An EPICS monitor, implemented via the camonitor utility, captures all PV updates and records them to construct the execution trace. Specific function calls can be selected or omitted for tracking.

Comparing two execution traces requires a more nuanced approach than a simple one-to-one check. EnvTrace computes a holistic `full_score`, given in Eq.(1), by analyzing three distinct aspects of equivalence: **(1) State Sequence Alignment:** the first and most critical aspect is whether the correct sequence of operations occurred. We compare the ground-truth and the generated code traces using a sequence alignment algorithm from `difflib.SequenceMatcher`. This algorithm identifies matching subsequences and highlights insertions, deletions, and substitutions. Using this alignment, we compute a `pv_match_rate`, which represents the fraction of operations that are both of the expected PV type and contain the correct values. **(2) Temporal Dynamics Analysis:** for operations executed in the correct sequence, it is equally important that they occur with the proper timing and pace. Using the set of correctly matched PV changes from the alignment step, corresponding timestamps were extracted. A linear regression is performed between timestamps of the ground-truth and predicted code. A high coefficient of determination (R^2) indicates consistent pacing, while a slope close to 1.0 indicates a correct overall duration. These factors, along with the Mean Absolute Percentage Error (MAPE) of the time intervals, are combined into a `timing_score`. **(3) Continuous Process Fidelity:** Some beamline operations, like temperature ramping, are continuous processes where the evolution of a PV state over time is critical, not just its final state. To evaluate these, EnvTrace employs "process fidelity metrics" designed to compare the overall trajectory of these values. For instance, when evaluating a temperature ramp, we calculate the Mean Absolute Error (MAE) across the entire profile, as well as the absolute difference in the final temperature. These are combined into an exponentially decaying `temp_score`, which rewards close tracking of the target profile. The specific thresholds and detailed logic for each component are provided in Section S2.1.

The motor PVs and temperature are monitored in this work, however the metric can be adapted to accommodate other instrument status, e.g. in-situ sample humidity or pressure. Here three individual scores are combined into a single, weighted simulation `full_score` that provides a comprehensive measure of functional equivalence,

$$\text{full_score} = \begin{cases} 0.6 \cdot \text{pv_match_rate} + 0.2 \cdot \text{timing_score} + 0.2 \cdot \text{temp_score} & \text{if temperature} \\ 0.8 \cdot \text{pv_match_rate} + 0.2 \cdot \text{timing_score} & \text{if no temperature} \end{cases} \quad (1)$$

This weighting is chosen to reflect a hierarchy of importance for physical systems. The state sequence, captured by the `pv_match_rate`, receives the highest weight as it represents the *primary* functional correctness, i.e., whether the

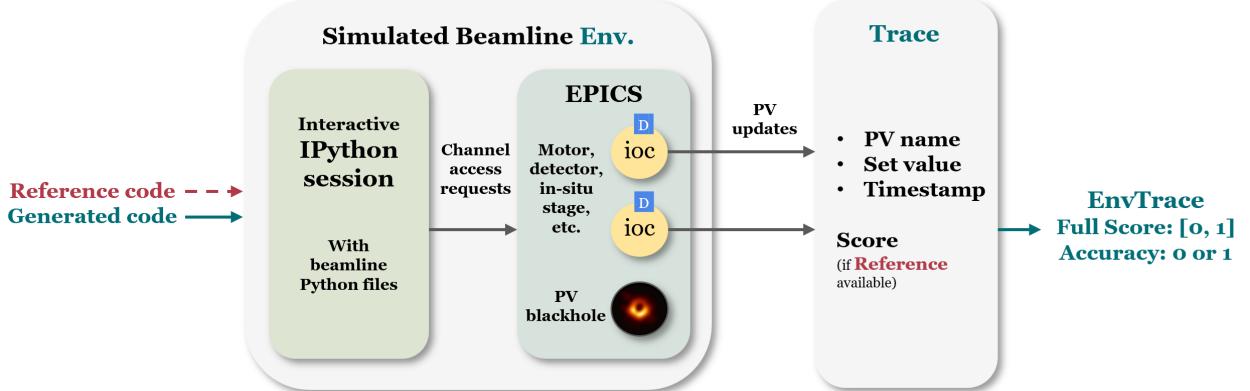


Figure 2: An overview of the EnvTrace architecture. LLM-generated code and ground-truth code are executed within an interactive IPython session. This session communicates with a simulated beamline environment, which runs EPICS IOCs in Docker containers. An EPICS monitor captures all state changes and generates an execution trace. The ground-truth and predicted code traces are then aligned to compute a multi-faceted EnvTrace full score based on state, temporal, and behavioral equivalence².

code produce the correct operations with the correct targeted values. A failure in this aspect constitutes a fundamental error. In contrast, the `timing_score` and `temp_score` represent *secondary* performance characteristics. This is because multiple, equally valid interpretation of intent and coding strategies can exist to achieve the same high-level experimental goal, each producing a different low-level execution trace. For instance, a task to “measure every 20°C during a ramp to 200°C” could be implemented in at least two functionally correct ways: (a) By setting a single final temperature target of 200°C and programming a loop that waits for the temperature to cross multiples of 20°C before triggering a measurement. (b) By programming a loop that explicitly sets a series of discrete 20°C heating steps, e.g., set to 20°C, wait, measure; set to 40°C, wait, measure; and so on. While both strategies achieve the desired instrument and measurement outcomes, different PV traces are generated for the temperature setpoint and result in varying temporal profiles. Consequently, the scoring metric can be defined to prioritize the successful execution of the core commands, such as the sequence of detector triggers that takes measurements, which are evaluated by the highly-weighted `pv_match_rate`. The specific implementation path, reflected in the temperature profile and timing, is evaluated by the lower-weighted `temp_score` and `timing_score`. This approach ensures that implementations which should be considered functionally correct yet logically or stylistically different still achieve a high overall score, recognizing them as valid solutions. This EnvTrace full score, ranging from 0.0 to 1.0, forms the basis of our semantic evaluation. When there are multiple reference answers, the one that gives the highest full score is used to assess the model performance.

As a strict and conservative metric, we also employed a measure referred to as ‘EnvTrace accuracy’, which is a more stringent, composite criterion with only binary results (success or failure). This also provides a comparison with the binary string-based exact match approach as used in Mathur et al. [31]. Instead of applying a threshold to each score component in Eq. (1) as in the full score, the generated code snippet is considered a functional match with a score of 1 only if it passes a series of independent strict checks across all relevant aspects of its execution trace, otherwise it is considered a mismatch with a score of 0. EnvTrace accuracy serves as a simulator informed lower-bound due to its strictness. This metric for evaluating functional match requires an exact match in the sequence of state changes, strict adherence to temporal dynamics, and, when relevant, high accuracy in tracking continuous processes like temperature ramps. Any deviation from these criteria results in the code being categorized as a mismatch. Although some may view this metric as overly conservative, it provides an additional means of evaluation, particularly for highly sensitive systems that demand an extremely high degree of functional and semantic correctness. Details for defining accuracy are given in Section S2.2.

3 Results

Our investigation and evaluation are structured around two central questions: (1) Does the proposed semantic, execution-based approach yield a more reliable measure of code correctness than traditional syntactic metrics? and (2) How do state-of-the-art LLMs perform under this more rigorous benchmark for instrument control? The benchmark dataset for evaluating LLM performance includes 116 simple cases comprising single- or few-line commands and 20 complex-flow control examples involving `for` or `while` loops. The ground-truth (GT) solutions were generated through a collaborative process involving two human scientists and Claude-3.7-Sonnet. Each GT entry was executed within our simulator to verify its functional correctness and resolve bugs. Due to the inherent ambiguity in some natural language prompts, most examples have multiple valid GT implementations, therefore a functional or exact string match with any of them is considered correct. Ground-truths here should be viewed as reference implementations, and due to inherent ambiguity in intent, a less-than-perfect full score may reflect an interpretation of intent mismatch rather than an actual error. The resulting scores should be interpreted in this context, while also considering that fewer than five GTs are present per example. Moreover, to better interpret the full score of LLMs, two human beamline scientists also provided their code implementations for the complex-flow prompts. Human submissions were manually debugged using the simulator to reflect typical beamline workflows, and the corrected versions were evaluated with EnvTrace.

Our study benchmarked the code quality and performance across closed- and open-source models, with model details provided in Section S3. For all LLM evaluations, each prompt was run 3 times and the results averaged to strengthen statistical significance. To improve efficiency when evaluating across datasets and models, the framework employs caching to avoid redundant executions. Once a piece of code has been run in the simulator, subsequent runs retrieve the stored PV changes directly, rather than re-executing the simulation. The variability in evaluation results may arise from the stochastic nature of LLM code generation and from fluctuations in simulator execution timing. With the use of caching, the effect of simulator variability will be less visible in the results. The scoring outcomes are thus also discussed qualitatively along with the quantitative results.

Box 1: Example of a simple-flow prompt and LLM code, showing that while closed-source model Claude-Sonnet-4 (Thinking) provided an accurate solution, the open-source model Qwen2.5-coder hallucinated one of the commands.

```

NL INPUT: Move the organic thin film x by 5.8 and align the thin film
-----
#GROUND-TRUTH SNIPPET:
sam.xr(5.8);
sam.align()
=====
#Claude Sonnet 4 (Thinking) PREDICTED SNIPPET:
sam = Sample('organic_thin_film')
sam.xr(5.8);
sam.align()
#-----
#PV match rate: 100.00%
#Timing match: True (score: 0.999)
#EnvTrace Accuracy: True (Full score: 1.000)
=====
#Qwen2.5-coder PREDICTED SNIPPET:
sam.xr(5.8);
sam.aligned()
#-----
#PV match rate: 50.00%
#Timing match: True (score: 1.000)
#EnvTrace Accuracy: False (Full score: 0.600)

```

3.1 Model Performance on Simple-Flow Tasks

EnvTrace was used for LLM evaluation on simple-flow examples, which typically involve single-line commands for actions, e.g. moving a motor or taking a measurement. The performance of 31 LLMs (including reasoning and vendor variants) on the 116 simple flow tasks is detailed in Table 1, with selected model performance plotted in Fig. 3. In each model family, the model performance is sorted by the EnvTrace full score, from highest to lowest. There is a notable difference between string-based ‘exact match’ and ‘EnvTrace accuracy’. This difference arises because LLMs often generate code that is functionally equivalent but syntactically distinct from the ground-truth examples. Common variations include using different variable names, adding comments, explicitly naming default parameters in functions, or using different functions to accomplish the same behavior. While these variations cause syntactic mismatch, EnvTrace can correctly identify them as functionally identical by verifying that they produced the same sequence of state changes in the simulator. As shown in Fig. 3, each LLM’s performance is shown with the blue bar (on the left) representing the EnvTrace full score, while the orange and green stacked bar (on the right) reflects exact match accuracy and the added correctness captured by the simulator, respectively. The green segment reflects improvement in functional correctness missed by syntactic metrics.

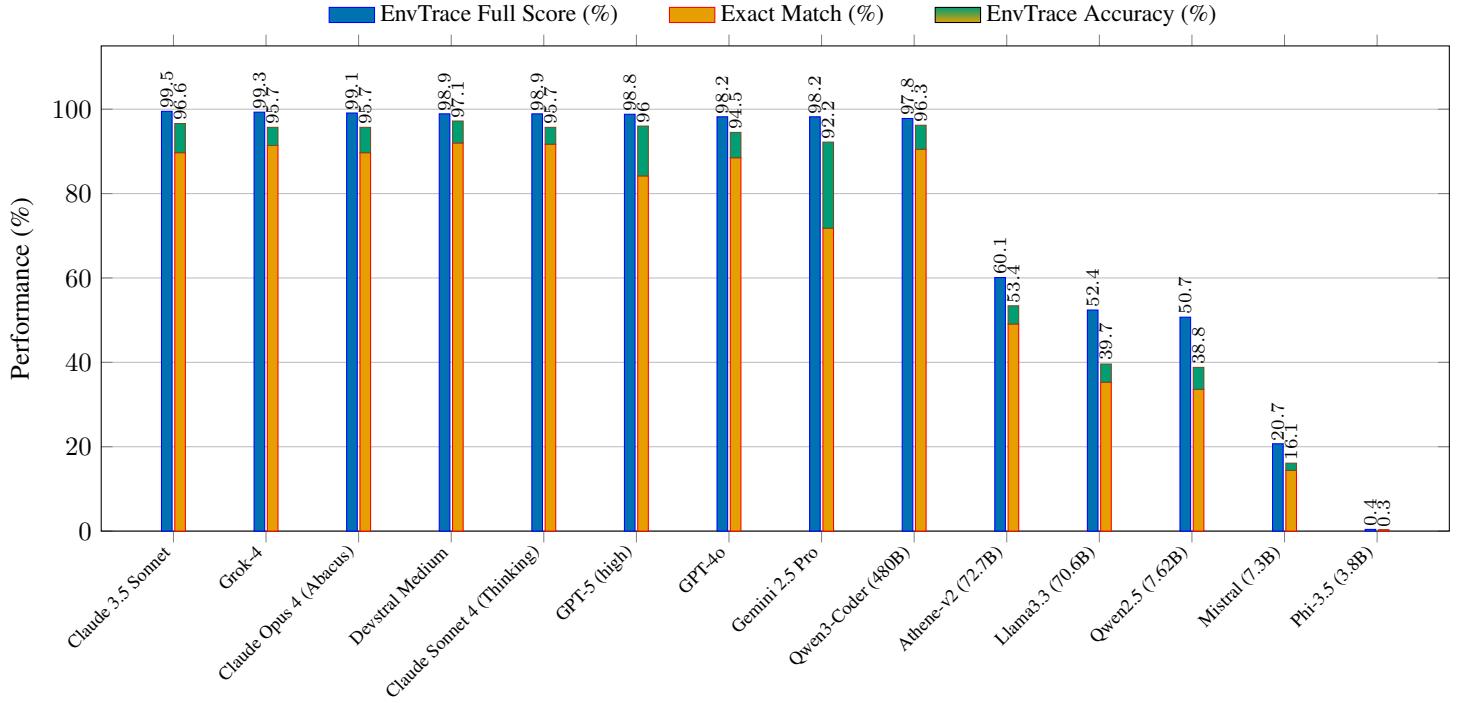
An example of a simple-flow task, with corresponding LLM results and evolution, is given in Box 1. The natural language query “Move the organic thin film x by 5.8 and align the thin film” corresponds to moving the sample in x direction with `sam.xr(5.8)` and perform the sample alignment routine with `sam.align()`. If the generated code performs the correct actions but additionally specifies a sample name as by Claude-Sonnet-4-(Thinking), string-based exact match would fail, while EnvTrace would return a perfect PV match, achieved EnvTrace accuracy, and a full score ideally 100%. In the case where the generated code is partially correct, as in the case of Qwen2.5-Coder where the alignment function is written incorrectly, EnvTrace can indicate a partial success by giving a score of 60%. EnvTrace accuracy and full score, both stemmed from the same trace-based semantic analysis, provide improved and meaningful indicators of LLM coding performance on simple-flow tasks. By performing a deeper trace analysis, the full score serves as a stronger proxy for performance.

Table 1 presents the performance of both closed- and open-source models, showing the EnvTrace full score and accuracy, alongside string-based exact match, normalized Levenshtein distance, and inference time for comparison. Closed-source models from vendors such as Anthropic, xAI, and OpenAI form the top tier of performance, with all achieving greater than 90% on EnvTrace accuracy and more than 95% on full score. The less than perfect scores are generally due to, for example, the ground truth assuming that the thermal stage is already powered on when setting the temperature, but some LLMs turn it on again (redundant but harmless), resulting in an extra PV change and a slight mismatch. This illustrates the idea that an imperfect score can imply ‘mismatch’ rather than ‘error’. For some models, stochasticity can make a

noticeable difference. For example, in a simple command to move the sample stage, GPT-5-(minimal) occasionally ‘over-thought’ the task and, instead of simply moving the motor, it asked for clarification on the exposure time and failed to generate any code. This results in a lower performance score compared to other models in the OpenAI family. Meanwhile, GPT-5-(high) utilized try-except for robustness, although unnecessary in this case, but indicative of useful deep thinking for more complex scenarios. While open-source model Qwen3-Coder achieved performance comparable to closed-source ones, other open-source models performed significantly worse, with most achieving under 60% in both accuracy and full score. Athene-v2 and its agent-tuned variant, achieve full score around 60% (with accuracies around 53%). Other popular models like Llama3.3 and the Qwen series fall into a 40-50% full score range (with 30-40% accuracy range). This highlights that generating a simple/small piece of code that is syntactically and semantically correct remains a challenging task that is not yet mastered by most open-source models. These models often make small but critical errors: they might understand the task but hallucinate the command or its format, as illustrated by the Qwen2.5-coder example in Box 1. Smaller open-source models like Qwen2.5 and Mistral-(7.3B) offer fast inference (<1s), although at a considerable cost to accuracy. Closed-source model GPT-4o provides both fast inference (0.8s) and high accuracy (98%).

To assess the impact of prompt length, we analyzed LLM performance with shorter prompts in Section S4. When system prompts include more details and cover more functionalities, we observe that the performance of smaller LLMs decline significantly. For example, Qwen2.5-Coder performance dropped from a full score of over 98% with the 2000 word prompts (~3800 tokens with GPT-4o tokenizer), as shown in Table S2, to 38% with the use of the 5000 word prompt (~8500 tokens) in Table 1. This suggests that open-source models perform reasonably well when only limited knowledge is included in the system prompt; as the demand for broader knowledge or functionality increases, closed-source models lead in both accuracy and inference speed.

Figure 3: Selected model performance on simple-flow tasks in Table 1. Left blue bars show the EnvTrace full score, and right stacked bars give the string-based exact match (bottom, orange) and improvement (top, green) with EnvTrace accuracy.



3.2 Model Performance on Complex-Flow Tasks

Tasks involving complex control flows, such as nested loops (for or while) and conditional statements (if), present greater challenges for both LLM reasoning and evaluation due to the increased ambiguity in natural language interpretation and the variability of possible implementations. The performance of 31 LLMs on the 20 complex-flow examples is provided in Table 2 and plotted in Fig. 4. The full score and accuracy metrics offer different perspectives on how to interpret the evaluation. For example, Claude-Sonnet-4-(Thinking) achieves a full score of 91% but only 48% in accuracy. This accuracy means that around half of the time the generated code was not perfect, regardless of the

Table 1: Performance of LLMs on Simple-Flow Code Generation Tasks (N=116)

Model	EnvTrace Full Score (%), \uparrow ^a	EnvTrace Accuracy (%), \uparrow ^b	Exact Match (%, \uparrow) ^c	Norm. Lev. Dist. (%), \downarrow ^d	Inference Time (s, \downarrow)
Closed Source Models					
<i>Anthropic</i>					
Claude 3.5 Sonnet	99.5	96.6	89.7	2.4	1.6
Claude Sonnet 4 (Bedrock)	99.2	97.4	94.0	1.6	28.8
Claude Sonnet 4 (Abacus)	99.2	97.4	94.0	1.6	2.6 \pm 0.1
Claude Opus 4 (Abacus)	99.1	95.7	89.7	2.3	2.9 \pm 0.1
Claude Sonnet 4.5 (Thinking) ^t	99.1	95.7 \pm 0.9	94.0 \pm 2.3	2.2 \pm 1.1	7.1 \pm 0.8
Claude Sonnet 4 (Thinking) ^t	98.9 \pm 0.8	95.7 \pm 1.7	91.7 \pm 1.8	1.9 \pm 0.7	5.0 \pm 0.2
Claude Opus 4 (Thinking) ^t	98.4 \pm 1.0	95.4 \pm 1.0	88.2 \pm 1.3	2.6 \pm 0.5	5.7 \pm 0.1
<i>xAI</i>					
Grok-4 ^t	99.3	95.7	91.4 \pm 1.5	2.1 \pm 0.1	16.0 \pm 0.8
Grok-4 Fast	98.8 \pm 0.6	95.7 \pm 2.6	89.7 \pm 0.9	2.5 \pm 0.2	3.4 \pm 1.2
Grok Code Fast 1	97.5 \pm 0.9	91.4 \pm 2.6	81.6 \pm 3.5	5.4 \pm 1.1	2.9 \pm 0.1
<i>OpenAI / Microsoft</i>					
o3 (high) ^t	99.1 \pm 0.5	96.0 \pm 0.5	89.7 \pm 1.7	2.5 \pm 0.5	4.9 \pm 0.4
GPT-5 (high) ^t	98.8 \pm 0.6	96.0 \pm 0.5	84.2 \pm 2.2	6.3 \pm 1.4	31.8 \pm 2.1
GPT-4o	98.2	94.5 \pm 0.5	88.5 \pm 0.5	3.5 \pm 0.1	0.8
GPT-4o (Abacus)	97.6	92.8 \pm 0.5	80.7 \pm 0.5	4.7 \pm 0.1	2.3 \pm 0.6
GPT-5 (minimal) ^t	95.7 \pm 2.2	92.2 \pm 2.3	84.2 \pm 2.2	6.5 \pm 1.8	1.3 \pm 0.1
<i>Mistral</i>					
Devstral Medium ^t	98.9 \pm 0.4	97.1 \pm 0.5	92.0 \pm 0.5	2.9 \pm 0.5	2.0 \pm 1.4
<i>Google</i>					
Gemini 2.5 Pro ^t	98.2 \pm 0.4	92.2 \pm 1.5	71.8 \pm 0.5	7.0 \pm 0.1	5.1
Open Source Models					
Qwen3-Coder (480B)	97.8 \pm 0.7	96.3 \pm 1.0	90.5 \pm 0.9	3.2 \pm 0.7	2.0 \pm 0.5
Athene-v2-Agent (72.7B)	62.9	52.6	46.6	23.0	5.3
Athene-v2 (72.7B)	60.1	53.4	49.1	21.0	5.2
Qwen3-Coder (30.5B)	54.7 \pm 0.5	48.0 \pm 0.5	39.7	23.7	1.9
Llama3.3 (70.6B)	52.4	39.7	35.3	25.6	5.1
Qwen2.5 (7.62B)	50.7	38.8	33.6	23.4	0.8
Qwen2 (7.62B)	46.8	37.1	31.9	28.2	0.8
Qwen2.5-Coder (32.8B)	37.6	31.9	28.4	32.0	2.7
GPT-oss (117B)	33.2	27.6	25.9	59.8	240.8 \pm 0.3
Mistral-NeMo (12.2B)	25.9	17.2	11.2	38.6	1.1
Mistral (7.3B)	20.7 \pm 0.5	16.1 \pm 0.5	14.4 \pm 0.5	38.5 \pm 0.2	0.9 \pm 0.1
Phi-3.5 (3.8B, fp16)	0.6 \pm 1.0	0.3 \pm 0.5	0.3 \pm 0.5	99.1 \pm 1.5	1.4 \pm 1.4
Phi-3.5 (3.8B)	0.4 \pm 0.8	0.3 \pm 0.5	0.3 \pm 0.5	99.0 \pm 1.8	1.0 \pm 0.6
Qwen3 (32B)	0.0	0.0	0.0	96.4	6.6

^a **EnvTrace Full Score (%)**: The continuous score from 0 to 100, reflecting the weighted average of state and temporal fidelity.

^b **EnvTrace Accuracy (%)**: The percentage of test cases passing the strict binary criteria for semantic equivalence.

^c **Exact Match (%)**: The percentage of test cases where the generated code string is identical to a ground-truth solution [31].

^d **Normalized Levenshtein Distance (%)**: A value of 0% indicates identical strings.

^e All results are the mean and standard deviation over three runs.

^f Models are grouped and then sorted by descending Average Full Score.

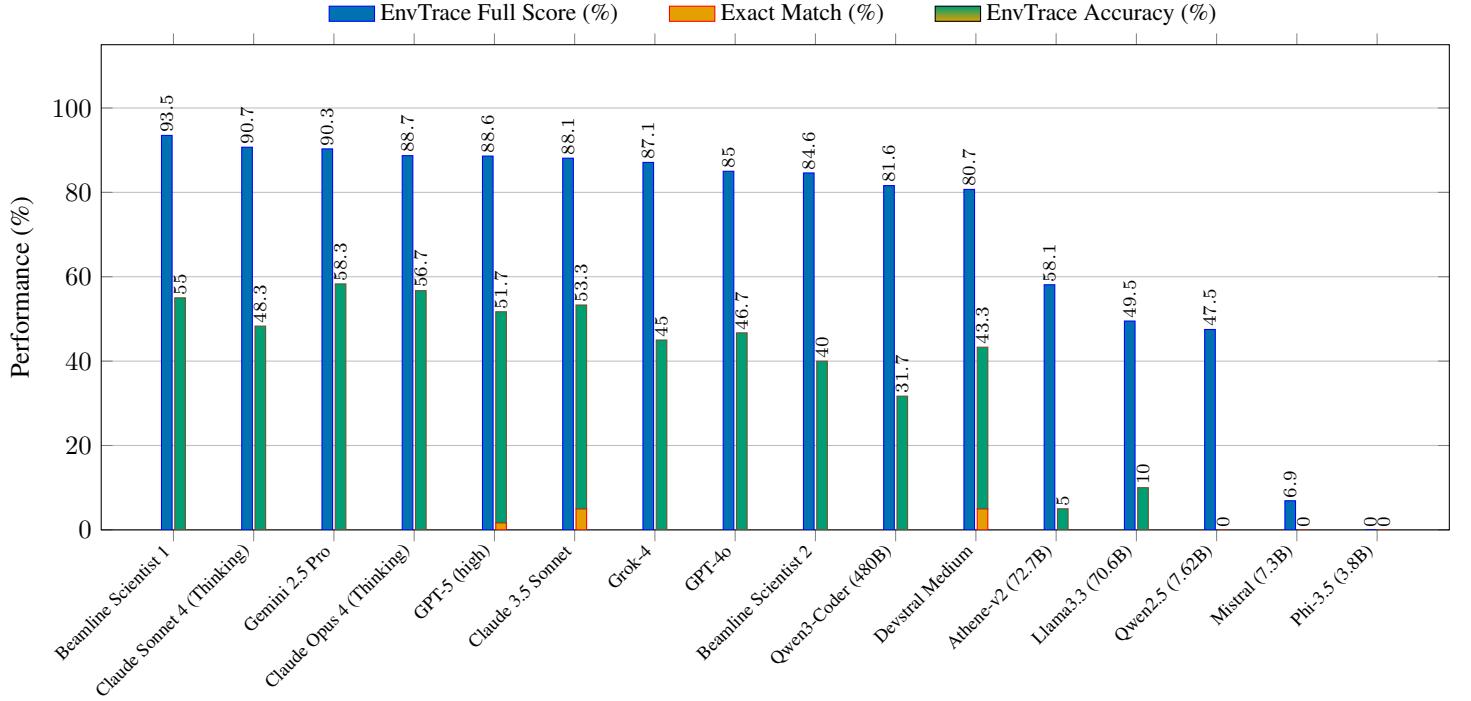
^g **No CodeBLEU on simple flows**: snippets are too short/non-self-contained for stable AST/data-flow scoring

^t Models run with reasoning enabled. “(high)” and “(minimal)” denote different reasoning levels.

magnitude of the error/mismatch. The 91% full score indicated that these mismatches were minor. The breakdown of the full score into PV match, timing score, and temperature score is shown in Fig. S2 in Section S5.2. Accuracy and the degree of mismatch can both be valuable aspects of the evaluation. The choice of metric can be adjusted depending on the specific application to better align with user needs.

The same top-tier closed-source models continue to lead for complex-flow tasks, with full score mostly above 85% and accuracy around 35-55%. These scores are comparable to those of human scientists, suggesting that LLMs can generate useful code drafts at human level, while final validation still requires human oversight or simulation-based evaluation. The imperfect scoring largely reflects ambiguities in the natural language prompts rather than actual errors. For example, after a series of measurements, one may choose to stay at the last measured position, return to the starting original position, or predict and move to a fresh sample spot to be ready for the next measurement. The order of the motor scanning can also introduce mismatch but do not result in actual error in the experiment. These varying interpretations are all valid, but differences in implementation lead to variations in the PV changes. If scattering data were simulated, one could envision adjusting the metric to take the scattering data or features into account rather than solely relying on the PVs. Different models, influenced by their stochastic characteristics, exhibit varying behaviors. For example, when asked to perform a map scan along sample x and y axes, most closed-source models would do a simple grid scan, while GPT-5-minimal and high (reasoning levels) would sometimes suggest a zigzag scan to minimize motor movement for efficient scanning, as shown in Box 2 with PV traces in Box S1. While all Grok models perform grid scan, Grok-Code-Fast-1 sometimes uses redundant motor movement: instead of moving sample x from 0.15 directly to 0.3, it moves from 0.15 back to 0 and then to 0.3. The averaged full score reflects the overall model performance and reliability, whereas the individual code comparison (per query) reflect specific errors or mismatches for easy bug detection or code improvement. Most open-source models struggled with this map scan task, frequently producing hallucinated functions, confusing relative and absolute motor movements, or generating incomplete code. Among them, only the Athene models consistently produced correct executions for the grid scan case, although their overall performance remained limited. In contrast, Qwen3-Coder-(480B) performed significantly better across most scenarios, but failed once (out of three runs) in the grid scan due to an incorrect Python function call, resulting in a `TypeError` and thus a score of 0. Aside from Qwen3-coder-(480B) that delivered comparable performance to closed-source models, in general the performance of open-source models did not exceed 60% in full score and failed to achieve accuracy above 10%, demonstrating that open-source models generally struggle with logic-based tasks. When provided with a much shorter prompt with limited beamline functionalities, performance of open-source models improves, as shown in Table S3.

Figure 4: Model performance on complex-flow tasks. Left blue bars give the EnvTrace full score, while right stacked bars provide the string-based exact match (bottom, orange) and improvement (top, green) with EnvTrace accuracy.



Box 2: Example of a complex-flow task and LLM code, showing that while closed-source model GPT-5-high provided efficient zigzag scan, open-source model use motor movement commands incorrectly.

```

NL INPUT: Do a map scan, x range from 0 to 0.3mm, y from 0 to 0.6mm, step size is 0.15
           horizontally and 0.2 vertically. (Exposure time 1s.)
-----
#GROUND-TRUTH SNIPPET:
for x in np.arange(0, 0.3 + 0.15/2, 0.15):
    for y in np.arange(0, 0.6 + 0.2/2, 0.2):
        sam.xabs(x)
        sam.yabs(y)
        sam.measure(1)
=====
#Gpt-5-high PREDICTED SNIPPET:
exposure_time = 1.0
step_x = 0.15
step_y = 0.2
x_end = 0.3
y_end = 0.6

nx = int(round(x_end / step_x)) + 1
ny = int(round(y_end / step_y)) + 1

for j in range(ny):
    for i in range(nx):
        sam.measure(exposure_time)
        if i < nx - 1:
            sam.xr(step_x if j % 2 == 0 else -step_x)
        if j < ny - 1:
            sam.yr(step_y)
#
#PV match rate (non-temp): 60.00%
#Timing match: True (score: 0.871)
#EnvTrace Accuracy: False (Full score: 0.654)
=====
#Athene-v2 PREDICTED SNIPPET:
for x in np.arange(0, 0.3 + 0.15/2, 0.15):
    for y in np.arange(0, 0.6 + 0.2/2, 0.2):
        sam.xr(x)
        sam.yr(y)
        sam.measure(1)
#
#PV match rate (non-temp): 46.94%
#Timing match: False (score: 0.780)
#EnvTrace Accuracy: False (Full score: 0.531)

```

Table 2: Performance of LLMs on Complex-Flow Code Generation Tasks (N=20)

Model	EnvTrace Full Score (%), ↑) ^a	EnvTrace Accuracy (%), ↑) ^b	CodeBLEU (comb-7, %, ↑) ^c	Norm. Lev. Dist. (%), ↓) ^d	Inference Time (s)
Closed Source Models					
<i>Anthropic</i>					
Claude Sonnet 4 (Thinking) ^t	90.7 ± 1.3	48.3 ± 2.9	61.4 ± 1.1	40.8 ± 2.1	9.1 ± 0.4
Claude Opus 4 (Thinking) ^t	88.7 ± 2.4	56.7 ± 7.6	57.8 ± 1.0	43.3 ± 0.6	12.2 ± 1.2
Claude Sonnet 4.5 (Thinking) ^t	88.4 ± 4.7	48.3 ± 5.8	60.2 ± 2.1	40.2 ± 1.7	15.4 ± 0.7
Claude 3.5 Sonnet	88.1 ± 0.5	53.3 ± 2.9	57.2 ± 0.8	39.1 ± 1.1	3.1 ± 0.1
Claude Opus 4 (Abacus)	82.8 ± 0.4	35.0	54.5 ± 0.2	46.8 ± 0.2	7.2 ± 0.6
Claude Sonnet 4 (Abacus)	80.4	35.0	57.0	42.0	4.2 ± 0.4
Claude Sonnet 4 (Bedrock)	80.0 ± 0.2	33.3 ± 2.9	57.7 ± 0.2	41.9 ± 0.5	11.2 ± 2.1
<i>Google</i>					
Gemini 2.5 Pro ^t	90.3 ± 0.6	58.3 ± 2.9	60.3 ± 2.0	39.6 ± 0.9	16.2 ± 0.9
<i>xAI</i>					
Grok-4 Fast	90.3 ± 0.7	60.0	57.4 ± 1.8	47.4 ± 1.0	10.2 ± 0.7
Grok-4 ^t	87.1 ± 1.1	45.0	55.9 ± 2.1	44.3 ± 3.8	74.8 ± 7.2
Grok Code Fast 1	85.2 ± 2.3	46.7 ± 2.9	58.4 ± 2.4	44.5 ± 1.3	7.4 ± 1.5
<i>OpenAI / Microsoft</i>					
GPT-5 (high) ^t	88.6 ± 2.1	51.7 ± 7.6	53.0 ± 2.8	50.4 ± 2.7	108.2 ± 12.9
GPT-4o	85.0 ± 1.2	46.7 ± 5.8	54.3 ± 0.4	46.1 ± 1.2	1.8
o3 (high) ^t	84.7 ± 1.6	45.0 ± 8.7	56.7 ± 1.1	42.9 ± 1.5	15.7 ± 0.4
GPT-5 (minimal) ^t	83.9 ± 3.8	33.3 ± 7.6	49.8 ± 1.3	51.7 ± 1.2	4.1 ± 0.4
GPT-4o (Abacus)	83.7 ± 1.1	48.3 ± 5.8	54.4 ± 0.6	44.2 ± 0.9	3.5 ± 0.3
<i>Mistral</i>					
Devstral Medium ^t	80.7 ± 0.6	43.3 ± 5.8	56.9 ± 0.4	40.9 ± 1.4	5.0 ± 4.3
Open Source Models					
Qwen3-Coder (480B)	81.6 ± 2.5	31.7 ± 2.9	49.9 ± 0.8	46.3 ± 1.4	4.8 ± 0.3
Athene-v2 (72.7B)	58.1	5.0	57.2	44.5	8.8 ± 0.7
Qwen3-Coder (30.5B)	57.4 ± 1.1	5.0	46.5 ± 0.1	51.1 ± 0.4	3.0 ± 0.2
Athene-v2-Agent (72.7B)	53.3 ± 0.2	0.0	55.8 ± 0.2	43.9 ± 0.7	8.0 ± 0.6
Llama3.3 (70.6B)	49.5 ± 1.1	10.0	49.9 ± 0.4	49.4 ± 0.2	9.3 ± 0.7
Qwen2.5 (7.62B)	47.5 ± 0.1	0.0	47.2 ± 2.0	48.5 ± 1.0	1.8 ± 0.5
Qwen2.5-Coder (32.8B)	42.7 ± 0.6	5.0	57.0	40.8 ± 0.1	4.7 ± 0.2
Mistral-NeMo (12.2B)	33.6 ± 1.6	5.0	52.2 ± 1.2	46.3 ± 0.5	2.2 ± 0.4
GPT-oss (117B)	32.4	5.0	48.5	72.2	213.0 ± 0.6
Qwen2 (7.62B)	25.9 ± 0.5	5.0	45.8 ± 0.3	59.9 ± 0.5	1.6 ± 0.2
Mistral (7.3B)	6.9	0.0	39.6	63.0	1.7
Phi-3.5 (3.8B)	0.0	0.0	40.0	100.0	0.7
Qwen3 (32B)	0.0	0.0	38.4	89.2	14.0 ± 0.1
Phi-3.5 (3.8B, fp16)	0.0	0.0	40.0	100.0	0.6
Humans					
Beamline Scientist 1	93.5	55.0	54.1	44.8	—
Beamline Scientist 2	84.6	40.0	48.6	42.8	—

^a EnvTrace Full Score (%): The continuous score from 0 to 100, reflecting the weighted average of state and temporal fidelity.^b EnvTrace Accuracy (%): The percentage of test cases passing the strict binary criteria for semantic equivalence.^c CodeBLEU (comb-7, %): syntactic similarity metric. We report the “comb-7” variant, which weights data-flow and syntax more heavily [5].^d Normalized Levenshtein Distance (%): A value of 0% indicates identical strings.^e All results are the mean and standard deviation over three runs.^f Models are grouped and then sorted by descending Average Full Score.^g Human Inference Time in the order of minutes (>100s)^t Models run with reasoning enabled. “(high)” and “(minimal)” denote different reasoning levels.

Box 3: Example of LLM generating a functionally equivalent Bluesky code

```

NL INPUT: Do a map scan over smx and smy , x from 0 to 5 over 6 steps , y from 0 to 10
          with 5 steps
-----
#GROUND-TRUTH SNIPPET:
RE(outer_product_scan(cms.detector, smx , 0, 5, 6, smy , 0, 10, 5))
=====
#GPT-4o PREDICTED SNIPPET:
# Perform a grid scan over smx and smy , with smx from 0 to 5 in 6 steps and smy from 0
          to 10 in 5 steps
RE(grid_scan(cms.detector, smx, 0, 5, 6, smy, 0, 10, 5))
#
#PV match rate (non-temp): 100.00%
#Timing match: True (score: 0.872)
#Full match: True (score: 0.974)

```

3.3 Discover Alternative Plans

The EnvTrace method can be readily adapted to any codebase or beamline for evaluating code and evaluating functional equivalence. We demonstrate this using native Bluesky plans [43], which are a set of several dozen pre-defined experimental procedures that serve as fundamental building blocks for constructing custom beamline operations. As shown in Table S5 and Fig. S3, even with minimal documentation available online, LLMs can effectively generate instrument-control code, and when paired with EnvTrace, their outputs can be meaningfully evaluated. For example, with the natural language query “Do a map scan over smx and smy, x from 0 to 5 over 6 steps, y from 0 to 10 with 5 steps”, GPT-4o used a different function with comments, as shown in Box 3. Exact string or syntactic metrics would fail entirely in this case. However, EnvTrace returned a perfect PV match for all 94 PV changes, with slight mismatch on timing due to simulator timing variation, producing a full score of 97%. Open-source models like Llama 3.3 and Athene tend to make minor but fatal edits to the commands, e.g., incorrect input, variable type, or name. Bluesky plans match simple-flow cases in brevity but differ in that they encapsulate a more intricate sequence of actions. Each Bluesky plan triggers a series of actions with several PV changes, showing that EnvTrace with a simulator is especially important to track simple code with internal procedures.

4 Discussion

We have demonstrated three categories of LLM-generated code evaluated with EnvTrace: (1) simple-flow code involving minimal instrument changes, (2) complex-flow incorporating control logic that allows creative solutions, and (3) simple code with internal procedures where LLMs can produce valid but different implementations. LLMs can generate codes with stylistic variation, creative logic, or alternative implementations; EnvTrace provides a robust and generalizable framework for evaluating LLMs in instrument-control settings by focusing on semantic, execution-based assessment rather than brittle syntactic metrics. This approach supports safer deployment, ensuring that only functionally correct code is executed on costly and sensitive instruments. It enables cross-beamline benchmarking, offering a unified evaluation method that can be applied consistently across different beamlines and facilities, thereby making performance results transferable and comparable. Below, we summarize the key findings and discuss future directions.

4.1 Semantic over Syntactic

A central claim of our work is that static syntactic-based code metrics do not reliably reflect the functional correctness essential for physical systems. Evaluating with functional correctness allows for the identification and correction of two critical failure modes of syntactic metrics, as discussed below.

Identify False Negatives: High-performing models like Claude-3.5-Sonnet achieve excellent EnvTrace full scores (over 80%) but are heavily penalized by syntactic metrics (CodeBLEU score around 50%). This occurs when the model generates a valid, alternative implementation that is stylistically different from the ground truth. EnvTrace correctly identifies this code as successful, whereas a purely syntactic evaluation would incorrectly flag it as low-quality.

Identify False Positives: Conversely, models such as Athene-v2 produce code that appears syntactically plausible (CodeBLEU score approx. 60%) but contains critical semantic errors, resulting in a lower functional score. These

subtle but critical bugs, such as incorrect function name (as shown in Box 1) or incorrect parameter in a function call, can cause minor variations in syntactic metrics but are immediately caught by our execution-based framework.

To further investigate these, we examined the relationship between our semantic EnvTrace full score and two widely-used syntactic metrics, CodeBLEU and normalized Levenshtein distance (nLD), with details provided in Section S8. Figure S4 shows the comparison between full score and normalized Levenshtein distance (nLD) for simple-flows. As longer Levenshtein distance indicates lower performance, here the higher EnvTrace full score roughly correlates with higher performance determined by nLD. While there is a correlation for simple-flows, many LLMs with an nLD between 20–40% exhibit significant variation in full score, ranging from around 20–80%. This shows that, despite the overall correlation, the full score provides a metric that more accurately reflects the actual performance. As shown by the Qwen2.5-coder example in Box 1, when part of the generated code provides correct execution, full score is able to capture the partial success. For complex-flows, comparison of model performance with full score versus CodeBLEU is provided in Fig. S6. For most LLMs, CodeBLEU scores are clustered around 40–60%, while the full score varies from 0–90%, rendering the CodeBLEU a weak indicator for performance. As shown by the full score versus nLD in Fig. S5, except for very weak models that give higher nLD, the nLD metric places most LLMs around 40–60%, while again the full scores give a full range of around 0–90%. The lack of correlation between full score and syntactic metrics is expected as any slight variation in the variable name or the use of different but equivalent logic would be heavily penalized by syntactic methods. Semantic and syntactic differences are most significant in complex-flow problems or sequence of actions, while syntactic similarity is a better, yet imperfect, indicator for simple tasks. This growing divergence with task complexity underscores the necessity of semantic execution-based evaluation for developing robust and reliable real-world AI agents. The choice of metric can be tailored to the system by adjusting parameters based on the simulator’s capabilities and experiment complexity, such as choosing between full score or accuracy and tuning the sensitivity of each component to suit the system.

4.2 Interpreting Functional Performance

Our results clearly demonstrate that a semantic, execution-based evaluation framework like EnvTrace, provides a meaningful and insightful assessment of code-generating agents than traditional syntactic metrics. The continuous full score provides a practical metric for evaluating the quality of generated code, but its numerical value must be interpreted in the context of real-world beamline operations. Based on our human expert review of the execution traces, we have developed a heuristic for translating EnvTrace scores into practical assessments. A score above **90%** typically indicates a robust and reliable solution, where any minor deviations are functionally inconsequential. Scores between **70-90%** often represent code that is largely correct but may have minor flaws in timing or non-critical parameters. Scores below **60%**, however, almost always indicate a significant functional failure where the primary goal of the command was not achieved. This interpretive framework, validated by beamline scientists, allows the full score to serve as a reliable guide for both model development and operational decision-making. Small variations in the score, typically a few percents, can arise from the inherent stochasticity of the model or the simulated physical system, such as minor overheads in motor or detector responses that affect the timing component. This reflects the reality of working with physical hardware and underscores the value of a metric that is robust to such real-world fluctuations.

A key challenge in this domain is the nature of the ‘ground truth’ itself. Many natural language commands are inherently ambiguous and can have multiple valid implementations. For this reason, our ground-truth dataset should be viewed as a collection of ‘expected answers’ rather than an exclusive, rigid truth. A deviation from a ground-truth solution is not necessarily an error, but rather a ‘mismatch’ that may represent a valid alternative implementation. This nuance is clearly reflected in our human benchmark results. Even expert-written code did not achieve a perfect score right away, due to a combination of minor typos or bugs, different interpretations of the ambiguous natural language queries, or valid procedural differences. For instance, after a scan, one scientist might program the sample to return to its starting position while another might leave it at the end position; one person may interpret measurement interval as inclusive or exclusive of measurement time itself. If provided with a comprehensive list of possible GTs and carefully debugged human-generated code, the human performance presented in Section 3.2 would achieve perfect results. However, instead of generating this idealized performance, the human performance we presented here is a realistic benchmark based on quick responses from human scientists and non-exclusive GTs, i.e. multiple valid answers may exist beyond those represented in our evaluation. Therefore, the competitiveness of top-tier LLMs with the human benchmark strongly demonstrates their capacity to accelerate code generation.

For LLMs to be effectively integrated into workflows, it is essential to first establish basic infrastructure, such as clear documentation, well-structured code, and organized data and metadata. These steps are relatively straightforward and not technically demanding, yet they are crucial for making instruments and datasets ‘AI-ready’. Once in place, this foundation enables LLMs to automate tasks easily and reliably, paving the way for intelligent agentic workflows.

4.3 Symbiotic LLM and Digital Twin Systems

The simulator serves not only as part of AI evaluation but also as an integral component of the experimental workflow for human users. In initial attempts, human-written code often contains minor syntax errors or typos, particularly during time-sensitive stressful beamtime sessions. The coding capabilities of LLMs provide rapid code drafts for human review, while the simulator environment offers runtime feedback that enables fast, iterative debugging. This process can occur in real time during experiments or as a ‘pre-flight check’ for new experimental procedures, eliminating the need for direct beamline access and beamtime consumption. We have integrated the simulator into VISION [31] as part of the Operator for real-time operation, as shown in Fig. 5. This integration supports infrastructure-aware generation by loading the beamline-specific configuration files, ensuring that an LLM will generate code for the specific features and capabilities of the beamline. Users can interact with VISION using natural language, after which an LLM generates the corresponding code; users can then review, edit, and simulate the code as needed, providing a flexible and intuitive interface for controlling and testing instrument code. With simulation, PV traces from successful execution and errors (if any) from the IPython session will be displayed. Once the simulation is complete, the user can choose to get an AI evaluation, as described in Section S9. This AI evaluation takes in the natural language query, generated (or edited) code, and the PV changes stemming from the simulation, as shown by examples in Fig. 5 and Section S7. To provide physical grounding of LLM-generated code, the beamline EPICS archiver is used to load the most recent instrument state and settings (PV values) into the simulator, enabling accurate and realistic predictions. For improved beamline integration, developing a digital twin within the Bluesky ecosystem, e.g. Bluesky Adaptive, QueueServer, and Tiled, can facilitate more efficient and streamlined workflows.

Here, we used EnvTrace with a control-logic digital twin to track instrument changes. If an advanced simulator or digital twin is available, one could envision defining a metric based on the simulated scattering data [29] or features rather than solely relying on the instrument changes. Digital twins [23, 24] are important for advancing physical systems by providing real-time, high-fidelity simulations that mirror scientific instruments or complex environments in scientific facilities [46, 47]. Different methodologies, instrument configurations, and experiments can be explored without constraints on instrument availability and physical location. LLMs and digital twins are mutually beneficial: incorporating LLMs into digital twins allows AI agents to interpret natural-language instructions and manage simulated and physical interactions, while digital twins, in turn, provide realistic environments for evaluating LLM performance. Together, digital twins powered by LLMs within multi-agent ecosystems form the basis of embodied AI for enabling autonomous agents to perceive, reason, and act intelligently in real-world environments through complex, context-aware decision-making [48].

5 Conclusion

Deploying LLM coding agents in physical systems is challenging due to the high risks involved with hardware interaction and the lack of reliable evaluation metrics. EnvTrace addresses this by executing ground-truth and LLM-generated code in a simulator, comparing their execution traces to produce a multi-faceted score that captures correctness, timing, and continuous variable progression. We have demonstrated three types of examples for LLM code generation evaluated with EnvTrace: (1) simple-flow code involving minimal instrument changes, (2) complex-flow incorporating control logic that allows creative solutions, and (3) simple code with internal sequences where LLMs can produce valid but different implementations. LLMs can generate code with stylistic variation, creative logic, or alternative implementations. EnvTrace provides a robust and generalizable framework for evaluating LLMs in instrument-control settings by focusing on interpretable semantic and execution-based assessment, rather than brittle syntactic metrics.

We have also demonstrated the integration of the EnvTrace simulator into a real-time assistant to support LLM-embedded beamline operation. While our demonstrations are already deployable at the beamline, achieving broader impact and wider applicability will require further integration of LLMs within the Bluesky ecosystem, together with continued progress toward a full digital twin with e.g. simulated optics and X-ray data. Effective evaluation of LLM coding agents require digital twins, while LLM-augmented digital twins enable more intelligent, adaptive control, highlighting the mutual benefits of their integration. The integration of LLMs with digital twins not only advances the capabilities of physical AI but also establishes a foundation for embodied AI, enabling autonomous agents to learn, adapt, and operate reliably within complex real-world environments.

Acknowledgments

The work was supported by a DOE Early Career Research Program. This research also used beamline 11-BM (CMS) of the National Synchrotron Light Source II (NSLS-II) and utilized the X-ray scattering partner user program at the Center for Functional Nanomaterials (CFN), both of which are U.S. Department of Energy (DOE) Office of Science

User Facilities operated for the DOE Office of Science by Brookhaven National Laboratory under Contract No. DE-SC0012704. We thank beamline scientists Ruipeng Li and Siyu Wu and data scientists Jennefer Maldonado, Muzamil Hussain Syed, and technology architect Kunal Shroff for their assistance on the project.

References

- [1] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.
- [2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- [3] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, pages 311–318, 2002.
- [4] Aryaz Eghbali and Michael Pradel. Crystalbleu: precisely and efficiently measuring the similarity of code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.
- [5] Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. Codebleu: a method for automatic evaluation of code synthesis. *arXiv preprint arXiv:2009.10297*, 2020.
- [6] Shuyan Zhou, Uri Alon, Sumit Agarwal, and Graham Neubig. Codebertscore: Evaluating code generation with pretrained models of code. *arXiv preprint arXiv:2302.05527*, 2023.
- [7] Atharva Naik. On the limitations of embedding based methods for measuring functional correctness for code generation. *arXiv preprint arXiv:2405.01580*, 2024.
- [8] Terry Yue Zhuo. Ice-score: Instructing large language models to evaluate code. *arXiv preprint arXiv:2304.14317*, 2023.
- [9] Weixi Tong and Tianyi Zhang. Codejudge: Evaluating code generation with large language models. *arXiv preprint arXiv:2410.02184*, 2024.
- [10] Sumith Kulal, Panupong Pasupat, Kartik Chandra, Mina Lee, Oded Padon, Alex Aiken, and Percy S Liang. Spoc: Search-based pseudocode to code. *Advances in Neural Information Processing Systems*, 32, 2019.
- [11] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps. *NeurIPS*, 2021.
- [12] Noah van der Vleuten. Dr. boot: Bootstrapping program synthesis language models to perform repairing, 2023. URL <https://arxiv.org/abs/2507.15889>.
- [13] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.
- [14] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.
- [15] Adil Rasheed, Omer San, and Trond Kvamsdal. Digital twin: Values, challenges and enablers from a modeling perspective. *IEEE access*, 8:21980–22012, 2020.
- [16] Florian Stadtmann, Erik Rugaard Furevik, Adil Rasheed, and Trond Kvamsdal. Physics-guided federated learning as an enabler for digital twins. *Expert Systems with Applications*, 258:125169, 2024.
- [17] Herman Van der Auweraer and Dirk Hartmann. The executable digital twin: merging the digital and the physics worlds. *arXiv preprint arXiv:2210.17402*, 2022.
- [18] Dmitrii Ershenko, Glafira Derbysheva, Andreas Panayi, and Clement Fortin. Quantitative metrics for validation and decision-making in digital twins: a comparative study on a railway braking system. *Proceedings of the Design Society*, 5:2671–2680, 2025.
- [19] Paula Muñoz, Manuel Wimmer, Javier Troya, and Antonio Vallecillo. Using trace alignments for measuring the similarity between a physical and its digital twin. In *Proceedings of the 25th international conference on model driven engineering languages and systems: Companion proceedings*, pages 503–510, 2022.

- [20] Paula Muñoz, Javier Troya, and Antonio Vallecillo. Towards Measuring Digital Twins Fidelity at Runtime. In *Proceedings of the ACM/IEEE 27th International Conference on Model Driven Engineering Languages and Systems*, MODELS Companion ’24, pages 507–512, New York, NY, USA, October 2024. Association for Computing Machinery. ISBN 979-8-4007-0622-6. doi: 10.1145/3652620.3688267.
- [21] R. P. Jagadeesh Chandra Bose and Wil M. P. van der Aalst. Process diagnostics using trace alignment: Opportunities, issues, and challenges. *Information Systems*, 37(2):117–141, April 2012. ISSN 0306-4379. doi: 10.1016/j.is.2011.08.003.
- [22] Shuhong Chen, Sen Yang, Moliang Zhou, Randall Burd, and Ivan Marsic. Process-Oriented Iterative Multiple Alignment for Medical Process Mining. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*, pages 438–445, November 2017. doi: 10.1109/ICDMW.2017.63.
- [23] Linyao Yang, Shi Luo, Xi Cheng, and Lei Yu. Leveraging large language models for enhanced digital twin modeling: Trends, methods, and challenges. *arXiv preprint arXiv:2503.02167*, 2025.
- [24] Nan Zhang, Christian Vergara-Marcillo, Georgios Diamantopoulos, Jingran Shen, Nikos Tziritas, Rami Bahsoon, and Georgios Theodoropoulos. Large language models for explainable decisions in dynamic digital twins. In *International Conference on Dynamic Data Driven Applications Systems*, pages 81–89. Springer, 2024.
- [25] Feixiang Wang, Xiaojun Liu, Feng Lv, Chongxin Wang, Jin Shi, Xiaotian Zheng, and Chao Li. An llm-guided sd-lmd digital twin construction strategy (lsdt) for multi-industrial scenarios: Enhancing adaptability and efficiency. *Journal of Manufacturing Systems*, 80:995–1012, 2025.
- [26] Yu-Zheng Lin, Qinxuan Shi, Zhanglong Yang, Banafsheh Saber Latibari, Shalaka Satam, Sicong Shao, Soheil Salehi, and Pratik Satam. Ddd-gendt: Dynamic data-driven generative digital twin framework. *arXiv preprint arXiv:2501.00051*, 2024.
- [27] Thorsten Hellert, Drew Bertwistle, Simon C Leemann, Antonin Sulc, and Marco Venturini. Agentic ai for multi-stage physics experiments at a large-scale user facility particle accelerator. *arXiv preprint arXiv:2509.17255*, 2025.
- [28] Thorsten Hellert, João Montenegro, and Antonin Sulc. Alpha berkeley: A scalable framework for the orchestration of agentic systems. *arXiv preprint arXiv:2508.15066*, 2025.
- [29] Zhantao Chen, Alexander Petsch, Aidan Israelski, Rajan Plumley, Lingjia Shen, Cong Wang, Cheng Peng, Yuan Ni, Arun Bansil, Sugata Chowdhury, et al. An agentic artificially intelligent x-ray scientist. 2025.
- [30] Lance Yao, Suman Samantray, Ayana Ghosh, Kevin Roccapriore, Libor Kovarik, Sarah Allec, and Maxim Ziatdinov. Operationalizing serendipity: Multi-agent ai workflows for enhanced materials characterization with theory-in-the-loop. *arXiv preprint arXiv:2508.06569*, 2025.
- [31] Shray Mathur, Noah van der Vleuten, Kevin G Yager, and Esther Tsai. Vision: A modular ai assistant for natural human-instrument interaction at scientific user facilities. *Machine Learning: Science and Technology*, 2025.
- [32] Antonin Sulc, Thorsten Hellert, Raimund Kammering, Hayden Hoschouer, and Jason St John. Towards agentic ai on particle accelerators. *arXiv preprint arXiv:2409.06336*, 2024.
- [33] Michael H Prince, Henry Chan, Aikaterini Vriza, Tao Zhou, Varuni K Sastry, Yanqi Luo, Matthew T Dearing, Ross J Harder, Rama K Vasudevan, and Mathew J Cherukara. Opportunities for retrieval and tool augmented large language models in scientific facilities. *npj Computational Materials*, 10(1):251, 2024.
- [34] Daniel Potemkin, Carlos Soto, Ruipeng Li, Kevin Yager, and Esther Tsai. Virtual scientific companion for synchrotron beamlines: A prototype. *arxiv*, 2023. doi: 10.48550/arXiv.2312.17180. URL <https://doi.org/10.48550/arXiv.2312.17180>.
- [35] Philip Willmott. *An introduction to synchrotron radiation: techniques and applications*. John Wiley & Sons, 2019.
- [36] Tomas Aidukas, Nicholas W Phillips, Ana Diaz, Emiliya Poghosyan, Elisabeth Müller, Anthony FJ Levi, Gabriel Aeppli, Manuel Guizar-Sicairos, and Mirko Holler. High-performance 4-nm-resolution x-ray tomography using burst ptychography. *Nature*, 632(8023):81–88, 2024.
- [37] Mirko Holler, Michal Odstrcil, Manuel Guizar-Sicairos, Maxime Lebugle, Elisabeth Müller, Simone Finizio, Gemma Tinti, Christian David, Joshua Zusman, Walter Unglaub, et al. Three-dimensional imaging of integrated circuits with macro-to nanoscale zoom. *Nature Electronics*, 2(10):464–470, 2019.
- [38] Mirko Holler, Manuel Guizar-Sicairos, Esther HR Tsai, Roberto Dinapoli, Elisabeth Müller, Oliver Bunk, Jörg Raabe, and Gabriel Aeppli. High-resolution non-destructive three-dimensional imaging of integrated circuits. *Nature*, 543(7645):402–406, 2017.

- [39] Siraj Sidhik, Isaac Metcalf, Wenbin Li, Tim Kodalle, Connor J Dolan, Mohammad Khalili, Jin Hou, Faiz Mandani, Andrew Torma, Hao Zhang, et al. Two-dimensional perovskite templates for durable, efficient formamidinium perovskite solar cells. *Science*, 384(6701):1227–1235, 2024.
- [40] Yuanze Xu, Juno Kim, Shripathi Ramakrishnan, Taiyi Chen, Xiaoyu Zhang, Yugang Zhang, Andrew J Musser, and Qiuming Yu. Unraveling the formation mechanisms of highly oriented tin perovskite with a 3d-over-2d heterostructure. *ACS Energy Letters*, 9(9):4734–4745, 2024.
- [41] Chao Xiao, Jinde Zhang, Yang Li, Mingyuan Xie, and Dongbai Sun. Application of synchrotron radiation in fundamental research and clinical medicine. *Biomedicines*, 13(6):1419, 2025.
- [42] Sarah H. Shahmoradian, Esther H. R. Tsai, Ana Diaz, Manuel Guizar-Sicairos, Jorg Raabe, L Spycher, M Britschgi, A Ruf, H Stahlberg, and Mirko Holler. Three-dimensional imaging of biological tissue by cryo x-ray ptychography. *Scientific reports*, 7(1):1–12, 2017.
- [43] Daniel Allan, Thomas Caswell, Stuart Campbell, and Maksim Rakitin. Bluesky’s Ahead: A Multi-Facility Collaboration for an a la Carte Software Project for Data Acquisition and Management. *Synchrotron Radiation News*, 32(3):19–22, 2019. doi: 10.1080/08940886.2019.1608121. URL <https://doi.org/10.1080/08940886.2019.1608121>.
- [44] Leo Dalesio, Andrew Johnson, Kay-Uwe Kasemir, et al. The epics collaboration turns 30. In *17th International Conference on Accelerator and Large Experimental Physics Control Systems (ICALEPCS’19)*, New York, NY, USA, 05-11 October 2019, pages 101–105. JACOW Publishing, Geneva, Switzerland, 2020.
- [45] The Event Horizon Telescope Collaboration, Kazunori Akiyama, Antxon Alberdi, Walter Alef, Keiichi Asada, Rebecca Azulay, et al. First M87 Event Horizon Telescope Results. I. The Shadow of the Supermassive Black Hole. *The Astrophysical Journal Letters*, 875(1):L1, April 2019. ISSN 2041-8205, 2041-8213. doi: 10.3847/2041-8213/ab0ec7.
- [46] Mohammad Sadegh Es-haghi, Cosmin Anitescu, and Timon Rabczuk. Methods for enabling real-time analysis in digital twins: A literature review. *Computers & Structures*, 297:107342, July 2024. ISSN 0045-7949. doi: 10.1016/j.compstruc.2024.107342.
- [47] Kamran Iranshahi, Joshua Brun, Tim Arnold, Thomas Sergi, and Ulf Christian Müller. Digital twins: Recent advances and future directions in engineering fields. *Intelligent Systems with Applications*, 26:200516, June 2025. ISSN 2667-3053. doi: 10.1016/j.iswa.2025.200516.
- [48] Kevin G. Yager. Towards a Science Exocortex. *Digital Discovery*, 3:1933–1957, 2024. doi: 10.1039/D4DD00178H. URL <http://dx.doi.org/10.1039/D4DD00178H>.

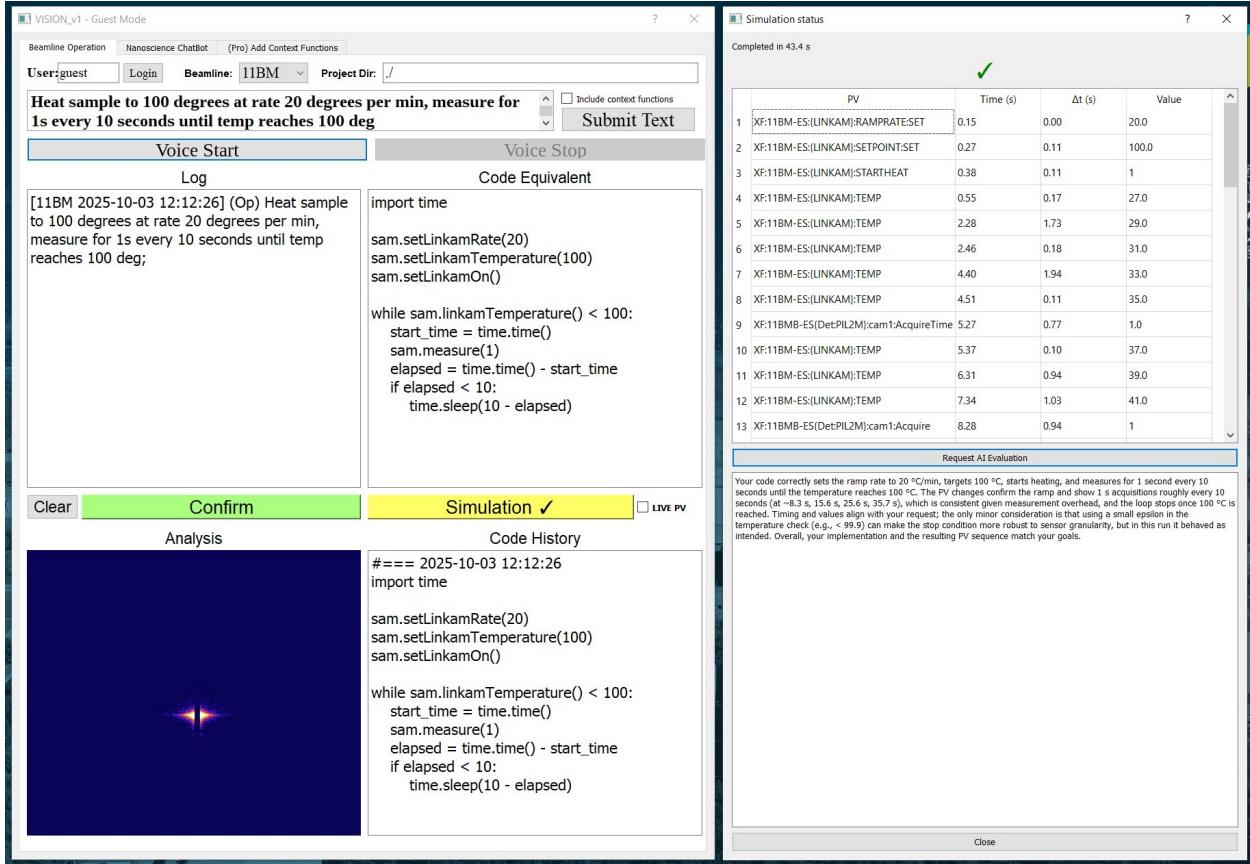


Figure 5: Illustration of the VISION GUI showing an example natural language query for performing in-situ thermal measurements, along with the corresponding LLM-generated code, partial PV traces, and an AI evaluation based on these inputs (query, code, and PV traces).

S1 EnvTrace Framework with Beamline Control-logic Digital Twin

A detailed diagram of the EnvTrace architecture is provided in Fig. S1. LLM-generated code and ground-truth code are executed within a sandboxed, interactive IPython session. The session, managed via the pexpect library for programmatic CLI interaction, communicates with a simulated beamline environment running EPICS IOCs in Docker containers. The environment is configured with the same set of Python functions used during live experiments at the beamline, providing access to the exact same Python API for instrument control. In other words, the same Python scripts used for beamline experiments can be run in the simulation without any modifications. An EPICS monitor captures all state changes (PV updates), generating an execution trace. The EnvTrace framework then aligns the ground-truth (reference) and predicted traces to compute a multi-faceted EnvTrace accuracy and full score based on state, temporal, and behavioral equivalence.

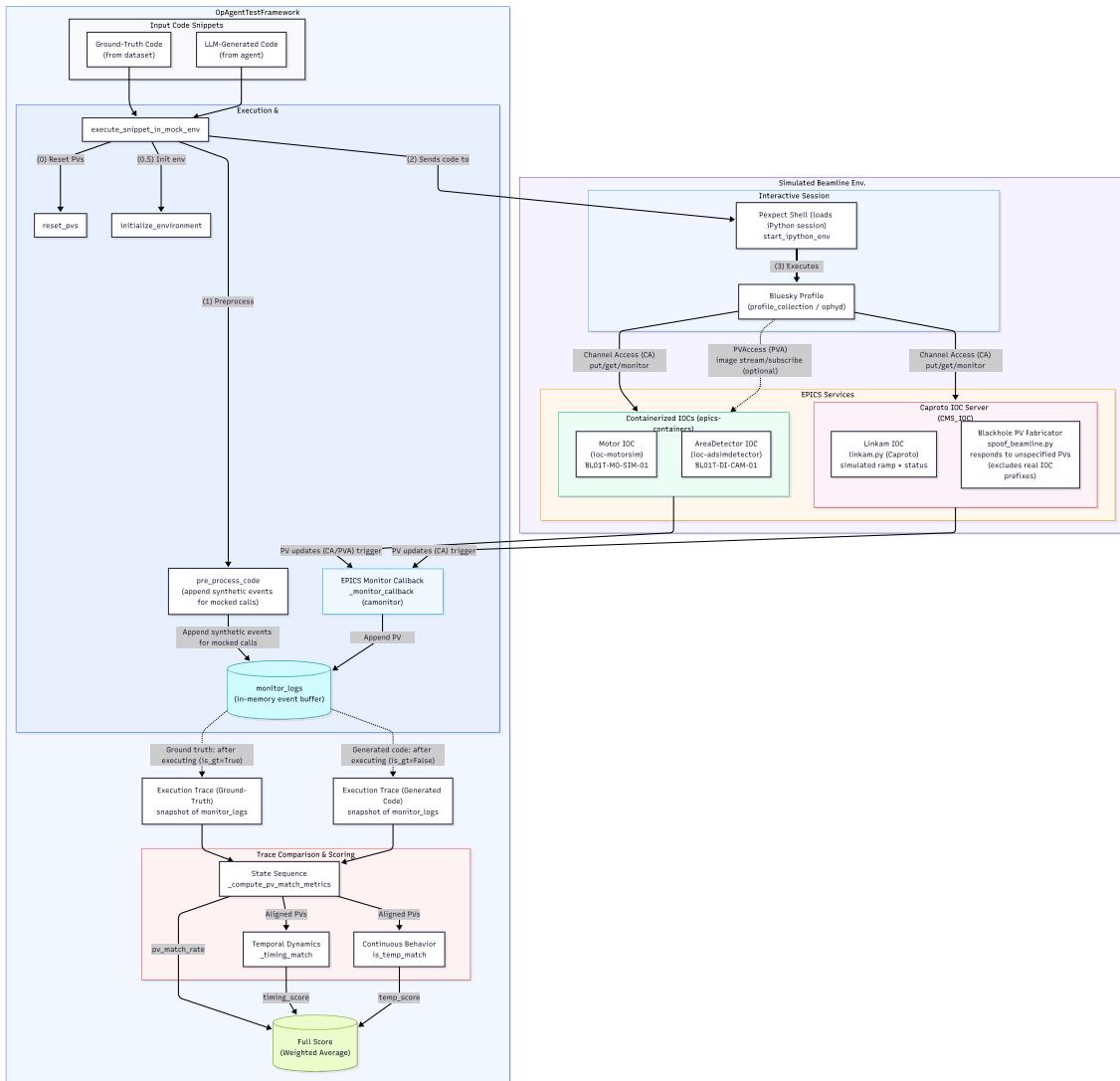


Figure S1: A diagram of the evaluation architecture showing how LLM-generated and ground-truth code run in an IPython session connected to a simulated beamline. PV updates are recorded as execution traces, based on which the EnvTrace framework calculates a comprehensive score reflecting state, timing, and behavioral alignment.

S2 EnvTrace Criteria

In this work, we monitor the PVs associated with the sample stages, a Linkam thermal stage, and an X-ray detector, which are all major components used at the beamline. Depending on the task and the physical instrument's status or the outcomes, different components can be tracked in EnvTrace to provide the full score and accuracy. Each metric's configuration can be tailored to fit specific use cases.

S2.1 EnvTrace Full Score

EnvTrace calculates a continuous, granular `full_score` between 0.0 and 1.0. This score provides a more nuanced measure of similarity than a strict pass/fail metric. It is composed of three component scores: `pv_match_rate`, `timing_score`, and `temp_score`.

State Sequence Score (`pv_match_rate`) The `pv_match_rate`, denoted as S_{pv} , is the fraction of correctly matched state changes after sequence alignment. It is calculated as:

$$S_{\text{pv}} = \frac{N_{\text{value_matches}}}{N_{\text{total_pairs}}} \quad (\text{S1})$$

where $N_{\text{value_matches}}$ is the number of aligned pairs with identical PV names and values, and $N_{\text{total_pairs}}$ is the total length of the aligned sequence (including insertions and deletions).

Temporal Dynamics Score (`timing_score`) The `timing_score`, denoted as S_{timing} , is a weighted average of four sub-scores that quantify different aspects of temporal similarity.

$$S_{\text{timing}} = 0.4 \cdot S_{r2} + 0.2 \cdot S_{\text{slope}} + 0.2 \cdot S_{\text{duration}} + 0.2 \cdot S_{\text{mape}} \quad (\text{S2})$$

The component scores are defined as:

- R-squared Score (S_{r2}): Directly uses the coefficient of determination from the linear regression of timestamps.

$$S_{r2} = R^2 \quad (\text{S3})$$

- Slope Score (S_{slope}): A linear penalty for deviations from a perfect slope of 1.0.

$$S_{\text{slope}} = \max(0, 1 - |\text{slope} - 1|) \quad (\text{S4})$$

- Duration Score (S_{duration}): A linear penalty based on the relative difference in total duration, normalized by a tolerance $\tau_{\text{dur}} = 0.25$.

$$S_{\text{duration}} = \max \left(0, 1 - \frac{|\text{dur}_{\text{pred}} - \text{dur}_{\text{gt}}| / \text{dur}_{\text{gt}}}{\tau_{\text{dur}}} \right) \quad (\text{S5})$$

- MAPE Score (S_{mape}): A linear penalty based on the Mean Absolute Percentage Error of the intervals, normalized by a tolerance $\tau_{\text{mape}} = 1.0$.

$$S_{\text{mape}} = \max \left(0, 1 - \frac{\text{MAPE}}{\tau_{\text{mape}}} \right) \quad (\text{S6})$$

Continuous Process Score (`temp_score`) The `temp_score`, denoted as S_{temp} , is a weighted average of two sub-scores based on an exponential decay function. This function penalizes large errors more significantly than small ones, providing a smooth score between 0 and 1.

$$S_{\text{temp}} = 0.7 \cdot S_{\text{mae}} + 0.3 \cdot S_{\text{final_temp}} \quad (\text{S7})$$

The component scores are defined as:

- MAE Score (S_{mae}): An exponential decay score based on the Mean Absolute Error of the temperature profiles.

$$S_{\text{mae}} = \exp \left(-\frac{\text{MAE}}{\lambda} \right) \quad (\text{S8})$$

- Final Temperature Score ($S_{\text{final_temp}}$): An exponential decay score based on the absolute difference in the final temperatures, ΔT_{final} .

$$S_{\text{final_temp}} = \exp \left(-\frac{\Delta T_{\text{final}}}{\lambda} \right) \quad (\text{S9})$$

For both scores, the characteristic scale λ is set to **15.0°C**.

Holistic Full Score The final continuous EnvTrace `full_score`, denoted as S_{full} , is the weighted sum of the component scores, as described also in Eq. (1).

$$S_{\text{full}} = \begin{cases} 0.6 \cdot S_{\text{pv}} + 0.2 \cdot S_{\text{timing}} + 0.2 \cdot S_{\text{temp}} & \text{if temperature involved} \\ 0.8 \cdot S_{\text{pv}} + 0.2 \cdot S_{\text{timing}} & \text{if no temperature involved} \end{cases} \quad (\text{S10})$$

S2.2 EnvTrace Accuracy

The binary “EnvTrace accuracy” metric is a composite metric derived from three binary flags: `exact_pv_match`, `timing_match`, and `temp_match`. A code snippet is only considered a functional match and thus high “accuracy” if all applicable criteria are met. The logic is as follows:

$$\text{accuracy} = \begin{cases} \text{exact_pv_match} \wedge \text{timing_match} \wedge \text{temp_match} & \text{if temperature involved} \\ \text{exact_pv_match} \wedge \text{timing_match} & \text{if no temperature involved} \end{cases} \quad (\text{S11})$$

The specific criteria for each of these binary flags are detailed below.

State Sequence Match (`exact_pv_match`) The `exact_pv_match` criterion evaluates whether the sequence of non-temperature-related state changes is perfectly identical between the ground-truth and predicted execution traces. It is the most stringent component of the evaluation. This flag is set to True if and only if:

- The number of PV changes in the predicted trace is exactly equal to the number of PV changes in the ground-truth trace.
- After sequence alignment, there are no insertions or deletions (i.e., no extra or missing operations).
- For every aligned pair of PV changes, the PV name and its corresponding value are identical (within a floating-point tolerance of 1×10^{-3} for numerical values).

This ensures a perfect one-to-one correspondence in both the actions performed and their outcomes.

Temporal Dynamics Match (`timing_match`) The `timing_match` criterion assesses whether the execution was paced correctly. This is determined by performing a linear regression on the timestamps of the correctly matched PV events. The flag is set to True if all of the following conditions are met:

- Goodness of Fit: The coefficient of determination (R^2) from the linear regression must be greater than or equal to a threshold of **0.90**. This ensures a consistent, linear relationship between the timing of the two executions.
- Pacing Slope: The slope of the regression line must be within the range **[0.8, 1.2]**. A slope of 1.0 indicates identical pacing; this range allows for a minor (20%) deviation.
- Total Duration: The relative difference in the total duration of the event sequence must be less than or equal to a tolerance of **25%**.
- Interval Consistency: The Mean Absolute Percentage Error (MAPE) between the time intervals of consecutive events must be less than or equal to a tolerance of **100%**.

Continuous Process Match (`temp_match`) For experiments involving temperature ramps, the `temp_match` criterion evaluates the fidelity of the temperature profile. The flag is set to True if both of the following conditions are met:

- Mean Absolute Error (MAE): The MAE between the ground-truth and predicted temperature logs must be less than or equal to a threshold of **5.0°C**.
- Final Temperature Difference: The absolute difference between the final recorded temperatures in the two logs must be less than or equal to a threshold of **5.0°C**.

S3 Models: Date, Provider, Quantization, Temperature

For assessing code quality and benchmarking model performance, models from a range of closed-source vendors and open-source communities were included, ensuring that the results encompassed both proprietary systems and publicly available alternatives. A sampling temperature of 0 was used for most models to promote deterministic outputs. For models with ‘thinking’ or reasoning capabilities explicitly enabled (e.g., Claude-Sonnet-4-(Thinking)), a temperature of 1.0 was required.

Table S1: Models: Date, Provider, Quantization, Temperature

Model	Date	Provider	Temperature	Reasoning Token Budget
Closed Source Models				
<i>Anthropic</i>				
Claude Sonnet 4 (Thinking)	2025-05-14	Native API	1	2000
Claude Opus 4 (Thinking)	2025-05-14	Native API	1	2000
Claude 3.5 Sonnet	2024-10-22	Native API	0	—
Claude Opus 4 (Abacus)	(2025-08-06)	Abacus	0	—
Claude Sonnet 4 (Abacus)	(2025-08-06)	Abacus	0	—
Claude Sonnet 4 (Bedrock)	2025-05-14	Bedrock	0	—
<i>Google</i>				
Gemini 2.5 Pro	(2025-09-15)	Native API	0	8192
<i>xAI</i>				
Grok-4 Fast	(2025-09-22)	OpenRouter	0	—
Grok-4	(2025-09-16)	OpenRouter	0	—
Grok Code Fast 1	(2025-09-17)	OpenRouter	0	—
<i>OpenAI / Microsoft</i>				
GPT-5 (high)	2025-08-07	Azure	1	high
GPT-4o	2024-05-13	Azure	0	—
o3 (high)	2025-04-16	Azure	1	high
GPT-5 (minimal)	2025-08-07	Azure	1	minimal
GPT-4o (Abacus)	(2025-08-01)	Abacus	0	—
<i>Mistral</i>				
Devstral Medium	2024-07-10	OpenRouter	0	—

Date in parentheses indicates access date due to unknown model date.

Model	Quantization	Provider	Temperature
Open Source Models			
<i>Qwen3-Coder</i>			
Qwen3-Coder (480B)	fp8/fp4	OpenRouter	0
Athene-v2 (72.7B)	Q4_K_M	Ollama	0
Qwen3-Coder (30.5B)	Q4_K_M	Ollama	0
Athene-v2-Agent (72.7B)	Q4_K_M	Hugging Face ^a	0
Llama3.3 (70.6B)	Q4_K_M	Ollama	0
Qwen2.5 (7.62B)	Q4_K_M	Ollama	0
Qwen2.5-Coder (32.8B)	Q4_K_M	Ollama	0
Mistral-NeMo (12.2B)	Q4_0	Ollama	0
GPT-oss (117B)	MXFP4	Ollama	0
Qwen2 (7.62B)	Q4_0	Ollama	0
Mistral (7.3B)	Q4_0	Ollama	0
Qwen3 (32B)	Q4_K_M	Ollama	0
Phi-3.5 (3.8B, fp16)	fp16	Ollama	0
Phi-3.5 (3.8B)	Q4_0	Ollama	0

^a Available at: <https://huggingface.co/lmstudio-community/Athene-V2-Agent-GGUF>

S4 Short system prompt

To evaluate how prompt length influences model performance, we performed the EnvTrace analysis using a short system prompt that is similar to the one used in Mathur et al. [31]. In contrast to the 5,000-word prompt (~ 8500 tokens with GPT-4o tokenizer) with extensive markdown formatting used in Section 3, this shorter version contains about 2,000 words ($\sim 3,800$ tokens) and consists of only the essential but not all routine beamline functions. Table S2 summarizes the results for the 116 simple-flow tasks using this short prompt. The open-source models perform substantially better under this short prompt than under the longer prompt used Table 1 in Section 3. Simpler, less token-heavy instructions reduce context overhead, which appears to benefit smaller models such as Qwen2.5-Coder, Llama3.3, and Athene-v2. With the short prompt, these models rival the closed-source models Claude-3.5-Sonnet and GPT-4o. Table S3 shows the results for the 20 complex-flow code generation tasks with the short system prompt, for comparison with the results from the long prompt in Table 2. Top-performing reasoning-enabled models, such as Claude-Sonnet-4-(Thinking) and GPT-4o, achieve high full scores (above 85%) and functional accuracies comparable to human benchmarks, while open-source models show mixed performance. Although smaller models remain weaker in logical control and looping structures, with the short prompt they produced syntactically valid and partially functional code much more consistently than with the longer prompt. Overall, these results indicate that simpler, shorter prompts enhance the reliability of open-source LLMs in instrument control tasks.

Table S2: Performance of LLMs on the new dataset (N=116) using the short system prompt.

Model	EnvTrace Full Score (%), ↑ ^a	EnvTrace Accuracy (%), ↑ ^b	Exact Match (%, ↑) ^c	Norm. Lev. Dist. (%), ↓ ^d	Inference Time (s, ↓)
Open Source Models					
Athene-v2-Agent (72.7B)	100.0	100.0	95.7	0.8	0.7 ± 0.1
Athene-v2 (72.7B)	99.7	99.1	94.8	1.0	0.8 ± 0.1
Llama3.3 (70.6B)	99.5 ± 0.1	98.6 ± 0.5	89.1 ± 0.5	2.5 ± 0.2	0.9 ± 0.1
Qwen2.5-Coder (32.8B)	98.5	95.7	89.7	2.8	1.0
Mistral-NeMo (12.2B)	97.5	92.2	77.3 ± 1.0	5.6 ± 0.2	1.2 ± 0.1
Qwen2.5 (7.62B)	96.0	94.0	85.9 ± 0.5	5.1 ± 0.1	1.1 ± 0.1
Qwen2 (7.62B)	95.2	91.4	81.0	6.2	1.1
Mistral (7.3B)	41.3	37.9	19.0	46.3	1.3
Phi-3.5 (3.8B, fp16)	14.0 ± 2.9	0.9 ± 1.5	0.9 ± 1.5	98.8 ± 2.0	1.2 ± 1.1
Phi-3.5 (3.8B)	13.6 ± 3.5	0.9 ± 1.5	0.9 ± 1.5	98.4 ± 2.7	0.8 ± 0.4
Closed Source Models					
<i>Anthropic</i>					
Claude 3.5 Sonnet	99.5	98.3	93.1	1.4	0.9
Claude Sonnet 4 (Thinking) ^t	97.5 ± 0.4	96.3 ± 0.5	92.5 ± 1.3	2.7 ± 0.3	4.5 ± 0.3
<i>OpenAI / Microsoft</i>					
GPT-4o	99.4 ± 0.1	96.0 ± 0.5	89.9 ± 0.5	2.2 ± 0.2	0.7 ± 0.1

^a **EnvTrace Full Score (%)**: The continuous score from 0 to 100, reflecting the weighted average of state and temporal fidelity.

^b **EnvTrace Accuracy (%)**: The percentage of test cases passing the strict binary criteria for semantic equivalence.

^c **Exact Match (%)**: The percentage of test cases where the generated code string is identical to a ground-truth solution [31].

^d **Normalized Levenshtein Distance (%)**: A value of 0% indicates identical strings.

^e All results are the mean and standard deviation over three runs.

^f Models are grouped and then sorted by descending Average Full Score.

S5 Complex-flow results

An example of PV trace alignment is shown to demonstrate the calculation of the PV match rate. We also present a detailed breakdown of LLM performance by PV match, timing, and temperature, offering deeper insight into the model’s coding capabilities.

S5.1 PV match example

For the GPT-5-high example on map scan in Box 2, a comparison on the PV traces is given in Box S1. The ‘check’ and ‘cross’ in the right-most column show whether the PV traces match or not. Although both code versions are semantically

Table S3: Performance of LLMs on Complex-Flow Code Generation Tasks (N=20) with the short prompt

Model	EnvTrace	EnvTrace	CodeBLEU	Norm. Lev.	Inference
	Full Score (% ,↑) ^a	Accuracy (% ,↑) ^b	(comb-7, % ,↑) ^c	Dist. (% ,↓) ^d	Time (s)
Closed Source Models					
<i>Anthropic</i>					
Claude Sonnet 4 (Thinking) ^t	91.9 ± 1.5	55.0 ± 5.0	55.5 ± 1.1	42.0 ± 3.3	9.4 ± 0.7
Claude 3.5 Sonnet	89.8 ± 0.5	61.7 ± 2.9	52.4 ± 0.1	39.4 ± 0.2	2.1 ± 0.1
<i>OpenAI / Microsoft</i>					
GPT-4o	83.0 ± 0.9	43.3 ± 2.9	52.2 ± 0.2	44.9 ± 0.9	1.5
Open Source Models					
Athene-v2-Agent (72.7B)	79.5	40.0	48.2	46.1	7.0 ± 0.5
Athene-v2 (72.7B)	72.2 ± 0.2	41.7 ± 2.9	53.0	45.3 ± 0.2	7.2 ± 0.1
Qwen2.5-Coder (32.8B)	72.0	45.0	51.3	44.7	3.9 ± 0.4
Llama3.3 (70.6B)	71.3 ± 0.1	25.0	50.0	49.3 ± 0.1	6.1 ± 0.5
Qwen2 (7.62B)	62.4 ± 0.9	25.0	47.1 ± 1.1	50.7 ± 0.5	1.2 ± 0.3
Mistral-NeMo (12.2B)	58.0 ± 2.4	26.7 ± 5.8	43.9 ± 1.4	48.3 ± 2.5	1.8 ± 0.3
Qwen2.5 (7.62B)	52.1 ± 1.0	10.0	48.2 ± 0.3	50.6 ± 0.4	1.7 ± 0.5
Mistral (7.3B)	2.8 ± 0.2	0.0	48.2	66.8	1.8
Phi-3.5 (3.8B)	2.0	0.0	40.0	100.0	0.6
Phi-3.5 (3.8B, fp16)	2.0	0.0	40.0	100.0	0.5

^a **EnvTrace Full Score (%)**: The continuous score from 0 to 100, reflecting the weighted average of state and temporal fidelity.

^b **EnvTrace Accuracy (%)**: The percentage of test cases passing the strict binary criteria for semantic equivalence.

^c **CodeBLEU (comb-7, %)**: syntactic similarity metric. We report the “comb-7” variant, which weights data-flow and syntax more heavily [5].

^d **Normalized Levenshtein Distance (%)**: A value of 0% indicates identical strings.

^e All results are the mean and standard deviation over three runs.

^f Models are grouped and then sorted by descending Average Full Score.

^g **Human Inference Time** in the order of minutes (>100s)

^t Models run with reasoning enabled. “(high)” and “(minimal)” denote different reasoning levels.

correct and perform map scans, differences in the scanning order led to mismatched PV traces. This case illustrates a situation where the scoring reflects a mismatch rather than a true error.

S5.2 Complex-flow results with full score breakdown

The continuous full score and its components provide a deeper diagnostic insight into *why* models succeed or fail. Figure S2 breaks down the full score for a selection of models on complex flow tasks into its constituent parts: PV Match Rate, Timing Score, and Temperature Score. The analysis shows that top-performing models like Claude and GPT achieve high, balanced scores across all components, indicating comprehensive success in sequencing, timing, and continuous tracking. In contrast, for lower-performing models, the primary failure mode is a low PV match rate. For example, Llama-3.3 and Qwen-2.5 have PV Match scores below 50%, indicating that they consistently fail to generate the correct sequence of operations. Their timing and temperature scores appear relatively higher only because these metrics are calculated on the small subset of PVs that matched correctly. This component-level breakdown is a practical diagnostic tool, indicating that evaluation should focus on core logical correctness, e.g. here the PV match.

Box S1: Trace alignment and scoring for the GPT-5-high map scan example shown in Box 2.

COMMAND:									
Do a map scan, x range from 0 to 0.3mm, y from 0 to 0.6mm, step size is 0.15 horizontally and 0.2 vertically. (Exposure time 1s.)									
===== LOGS COMPARISON =====									
GROUND TRUTH									PREDICTED
XF:11BMB-ES(Det:PIL2M):cam:AcquireTime 09:24:33.402 1.0	XF:11BMB-ES(Det:PIL2M):cam:AcquireTime 18:07:23.020 1.0	✓							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:24:36.417 1	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:07:23.782 1	✓							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:24:37.419 0	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:07:24.784 0	✓							
XF:11BMB-ES(Ch:Smpl-Ax:Z)Mtr 09:24:38.189 0.2	XF:11BMB-ES(Ch:Smpl-Ax:X)Mtr 18:07:25.335 0.15	✗							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:24:44.894 1	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:07:30.777 1	✓							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:24:45.896 0	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:07:31.788 0	✓							
XF:11BMB-ES(Ch:Smpl-Ax:Z)Mtr 09:24:45.735 0.4	XF:11BMB-ES(Ch:Smpl-Ax:X)Mtr 18:07:32.326 0.30000000000000004	✗							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:24:51.684 1	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:07:37.759 1	✓							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:24:52.686 0	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:07:38.761 0	✓							
XF:11BMB-ES(Ch:Smpl-Ax:Z)Mtr 09:24:53.235 0.5000000000000000	XF:11BMB-ES(Ch:Smpl-Ax:X)Mtr 18:07:39.313 0.2	✗							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:24:59.101 1	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:07:45.189 1	✓							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:25:00.104 0	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:07:46.112 0	✓							
XF:11BMB-ES(Ch:Smpl-Ax:X)Mtr 09:25:00.757 0.15	XF:11BMB-ES(Ch:Smpl-Ax:X)Mtr 18:07:46.678 0.15000000000000002	✓							
XF:11BMB-ES(Ch:Smpl-Ax:Z)Mtr 09:25:01.110 0.0	✗								
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:25:07.342 1	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:07:52.094 1	✓							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:25:08.345 0	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:07:53.097 0	✓							
XF:11BMB-ES(Ch:Smpl-Ax:Z)Mtr 09:25:08.980 0.2	XF:11BMB-ES(Ch:Smpl-Ax:X)Mtr 18:07:53.646 3.688224830031759e-16	✗							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:25:14.870 1	✗								
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:25:15.873 0	✗								
XF:11BMB-ES(Ch:Smpl-Ax:Z)Mtr 09:25:16.502 0.4	✗								
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:25:22.361 1	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:07:59.070 1	✓							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:25:23.364 0	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:08:00.072 0	✓							
XF:11BMB-ES(Ch:Smpl-Ax:Z)Mtr 09:25:23.986 0.6000000000000000	XF:11BMB-ES(Ch:Smpl-Ax:X)Mtr 18:08:00.617 0.4	✗							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:25:29.852 1	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:08:06.349 1	✓							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:25:30.855 0	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:08:07.352 0	✓							
XF:11BMB-ES(Ch:Smpl-Ax:X)Mtr 09:25:31.497 0.3	XF:11BMB-ES(Ch:Smpl-Ax:X)Mtr 18:08:07.596 0.15	✗							
XF:11BMB-ES(Ch:Smpl-Ax:Z)Mtr 09:25:32.001 0.0	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:08:13.339 1	✗							
		XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:08:14.342 0	✗						
		XF:11BMB-ES(Ch:Smpl-Ax:X)Mtr 18:08:14.887 0.30000000000000004	✗						
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:25:38.266 1	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:08:20.314 1	✓							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:25:39.269 0	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:08:21.317 0	✓							
XF:11BMB-ES(Ch:Smpl-Ax:Z)Mtr 09:25:39.915 0.2	XF:11BMB-ES(Ch:Smpl-Ax:X)Mtr 18:08:21.871 0.50000000000000001	✗							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:25:45.768 1	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:08:27.658 1	✓							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:25:46.772 0	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:08:28.653 0	✓							
XF:11BMB-ES(Ch:Smpl-Ax:Z)Mtr 09:25:47.401 0.4	XF:11BMB-ES(Ch:Smpl-Ax:X)Mtr 18:08:29.199 0.15000000000000002	✗							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:25:53.206 1	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:08:34.625 1	✓							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:25:54.209 0	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:08:35.628 0	✓							
XF:11BMB-ES(Ch:Smpl-Ax:Z)Mtr 09:25:54.827 0.6000000000000000	XF:11BMB-ES(Ch:Smpl-Ax:X)Mtr 18:08:36.175 3.688224830031759e-16	✗							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:26:00.959 1	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:08:41.613 1	✓							
XF:11BMB-ES(Det:PIL2M):cam:Acquire 09:26:01.962 0	XF:11BMB-ES(Det:PIL2M):cam:Acquire 18:08:42.616 0	✗							
SUMMARY:									
Ground Truth: 38 log entries									
Predicted: 36 log entries									
Matches: 24									
Mismatches: 16									
Difference: 2 entries									

Exact PV match (non-temp): False
PV match rate (non-temp): 60.00%
PV mismatch rate (non-temp): 40.00%
Timing match: True (score: 0.871)
Full match: False (score: 0.654)

S6 Debug Baseline

To establish an approximate upper bound for performance and to validate the stability of both the simulator and the EnvTrace framework, we conducted a series of *debug* evaluations with the first code snippet from the GT. Because the beamline control simulator is not fully deterministic due to small timing variations in device updates and asynchronous process scheduling, each debugging test was executed three times to account for the natural runtime variability of the environment. Timing variations also exist in real experiments. As shown in Fig. S4, even when the same ground-truth code is evaluated against itself, small variations in timing can lead to minor deviations in the final score. While identical executions would ideally yield a perfect score of 100%, the simulator operates in real time with asynchronous device updates, introducing slight, expected timing jitter. These effects are small and consistent, serving mainly to illustrate that EnvTrace captures realistic execution variability rather than artificially idealized behavior. Accordingly, the debug scores represent an empirical upper bound of achievable performance under ideal functional conditions. Overall, the debug baselines demonstrate that EnvTrace is stable, reproducible within expected variability, and able to distinguish genuine functional correctness from superficial code similarity.

Figure S2: EnvTrace full score and component metrics (PV match, timing, temperature) for complex-flow tasks. Temperature score is computed only when applicable.

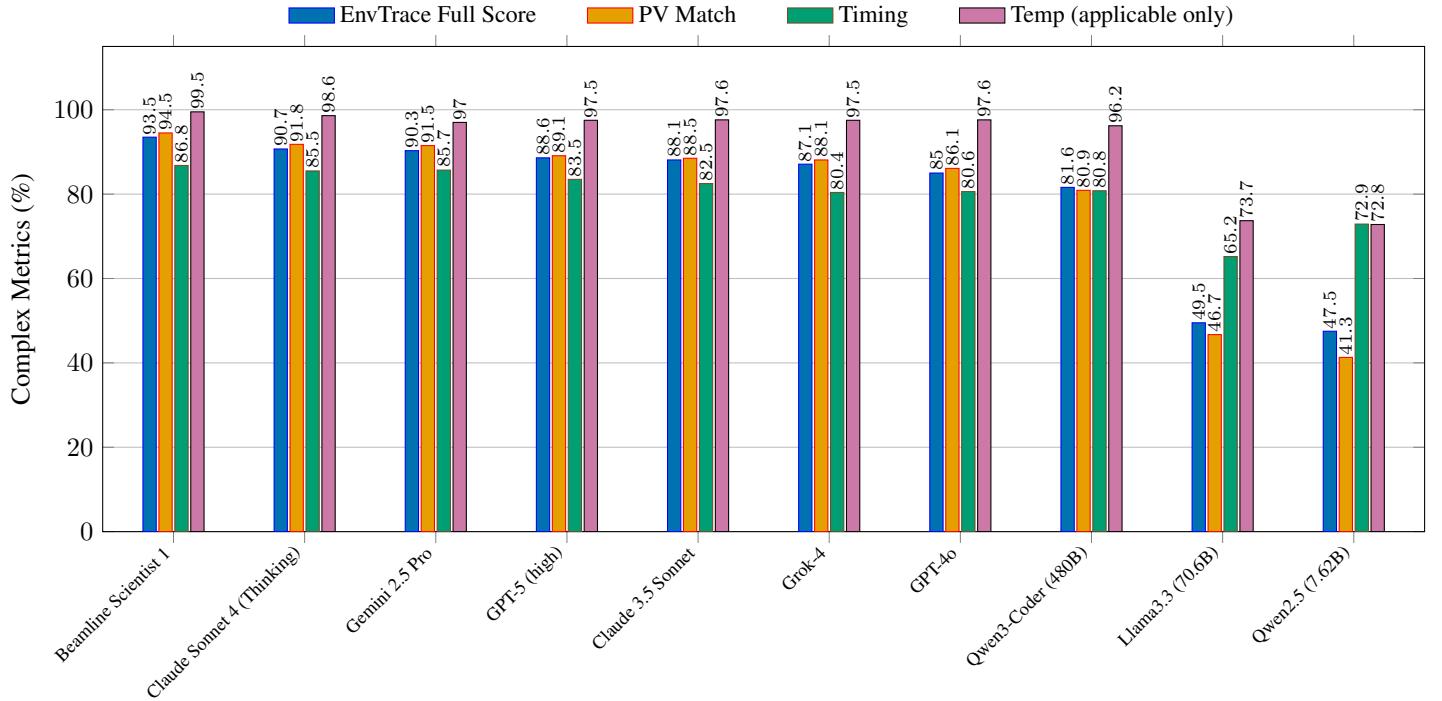


Table S4: EnvTrace performance of debug ground-truth runs, representing an approximate upper bound of achievable scores.

Evaluation Mode	EnvTrace Full Score (%)	EnvTrace Accuracy (%)
Simple-flow (Debug First GT)	98.1 ± 0.6	89.4 ± 1.8
Complex-flow (Debug First GT)	99.8 ± 0.1	100.0 ± 0.0

S7 Bluesky Experiments

To show the generalizability of VISION [31] and EnvTrace, we have adapted the operator cog (the part of the tool that takes care of generating operations) to produce code for a small benchmark dataset of generic Bluesky plans. These generic Bluesky plans can be applied across multiple Bluesky-supported beamlines without requiring additional documentation or code changes but only function input parameters need to be specified. A *beamline specifics* text file is dynamically loaded and included into the system prompt. This file contains beamline specific information, such as detector or motor names. These Bluesky plan definitions were parsed from the Bluesky documentation. Usage examples were produced both manually and through LLM generation with documentation reference and human verification. LLM performance for generic Bluesky plans is shown in Table S5 and Fig. S3.

S8 Semantic versus Syntactic Metrics

To further support the claim that syntactic metrics are an unsuitable proxy for functional correctness, here we show the semantic EnvTrace full score against CodeBLEU (comb 7) and normalized Levenshtein distance (nLD) for simple- and complex-flow tasks based on Table 1 and Table 2, respectively.

For simple-flow tasks consisting of single or short sequences of commands, the relationship between EnvTrace full score and nLD is shown in Fig. S4. The CodeBLEU metric is not applicable for simple-flow tasks, as data-flow analysis component requires more complex code structures to function correctly; for single-line or short commands, there is typically no data-flow to analyze. Complex flow tasks are those that require the generation of structured control

Table S5: Performance of LLMs on generic Bluesky tasks (N=20)

Model	EnvTrace Full Score (%) ^a	EnvTrace Accuracy (%) ^b	Exact Match Accuracy (%) ^c	Norm. Lev. Dist. (%) ^d	Inference Time (s)
Closed Source Models					
<i>OpenAI / Microsoft</i>					
o3 (high) ^t	93.1 ± 2.4	88.3 ± 2.9	55.0	11.5 ± 1.3	4.6 ± 0.8
GPT-4o (Abacus)	92.7 ± 0.2	85.0	65.0	7.2	1.9 ± 0.1
GPT-4o	87.4 ± 4.5	76.7 ± 2.9	61.7 ± 2.9	7.4 ± 0.2	1.5 ± 0.2
<i>Anthropic</i>					
Claude Opus 4 (Thinking) ^t	91.5 ± 0.1	85.0	65.0	7.2	8.1 ± 0.3
Claude Sonnet 4 (Thinking) ^t	91.5	85.0	65.0	7.2	5.9 ± 0.2
Claude 3.5 Sonnet	87.7 ± 0.1	78.3 ± 2.9	60.0	7.6	2.9 ± 0.2
Open Source Models					
Qwen2.5-Coder (32.8B)	62.3	55.0	30.0	26.5	3.4
Llama3.3 (70.6B)	55.0 ± 0.2	38.3 ± 2.9	20.0	29.9	7.1
Athene-v2 (72.7B)	35.8 ± 0.1	35.0	20.0	19.2	6.4
Mistral-NeMo (12.2B)	4.7	0.0	0.0	34.5	1.3
Phi-3.5 (3.8B)	0.0	0.0	0.0	84.3 ± 1.8	5.5 ± 1.2
Qwen3 (32B)	0.0	0.0	0.0	96.8	48.7

^a EnvTrace Full Score (%): The continuous score from 0 to 100, reflecting the weighted average of state and temporal fidelity.

^b EnvTrace Accuracy (%): The percentage of test cases passing the strict binary criteria for semantic equivalence.

^c Exact Match (%): The percentage of test cases where the generated code string is identical to a ground-truth solution [31].

^d Normalized Levenshtein Distance (%): A value of 0% indicates identical strings.

^e All results are the mean and standard deviation over three runs.

^f Models are grouped and then sorted by descending Average Full Score.

^t Models run with reasoning enabled. “(high)” and “(minimal)” denote different reasoning levels.

logic, such as for or while loops, and conditional if statements. For these tasks, the potential for stylistic and logical variation is high, making them suitable tests for the robustness of evaluation metrics. Figure S6 shows the relationship between EnvTrace score and CodeBLEU, and Fig. S5 for nLD. The dotted diagonal lines represent a hypothetical perfect one-to-one mapping between the syntactic and semantic scores (i.e., $y = x$ for CodeBLEU and $y = 1 - x$ for Levenshtein distance). The distance of a data point from the ideal line reflects the extent to which the syntactic metric may be unreliable.

While syntactic metrics can be useful for “sanity-checking” simple code generation (Fig. S4), their utility rapidly diminishes as task complexity increases (Fig. S5), reinforcing the need for a robust semantic evaluation framework. As seen in Figure S6, many models lie far from the ideal line. Points located significantly above the line, such as Claude-3.5-Sonnet and Claude-Opus-4- (Thinking), represent “false negatives” for the CodeBLEU metric; their functional performance is much higher than their syntactic similarity score would suggest. Conversely, points below the line, such as Athene-v2 and Qwen2.5-Coder, are “false positives”; their syntactic similarity is misleadingly high given their lower functional performance.

S9 AI Evaluator

In the VISION GUI, once a user has completed a simulation of the generated code, they can request an AI evaluation. This AI evaluation is driven by a prompt that takes in the user’s natural language query, the results of the simulation (the tracked PV changes), the operator cog prompt (the dynamically generated, beamline specific, coding prompt), and the generated code. Once the AI evaluation button is passed, it will give a short evaluation commenting on the pipeline (natural language query to code to PV trace) to the user. Here we used GPT-5-low as the AI evaluator, as it strikes a good balance between inference time and performance. An example in which AI evaluation provides correction is given in Fig. S7.

Figure S3: Model performance on the generic Bluesky-plan dataset (N=20). Left bars show the EnvTrace full score and right stacked bars give the string-based exact match (bottom, orange) and improvement (top, green) with EnvTrace accuracy.

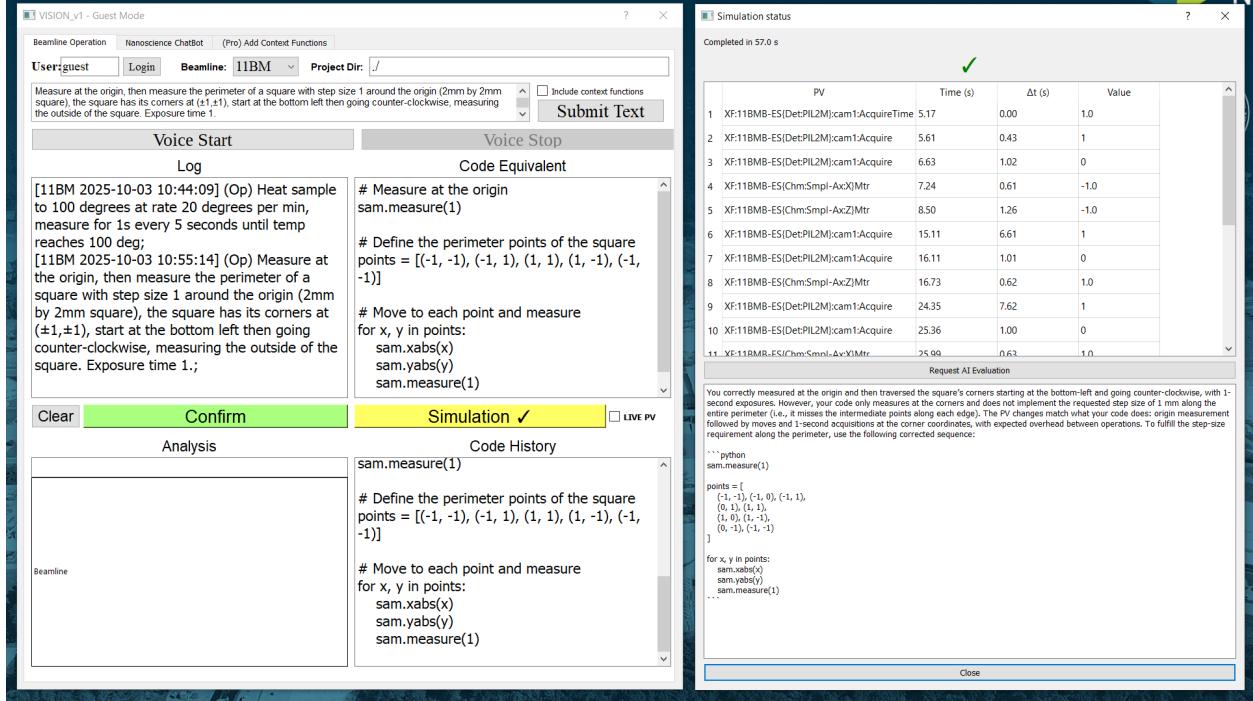
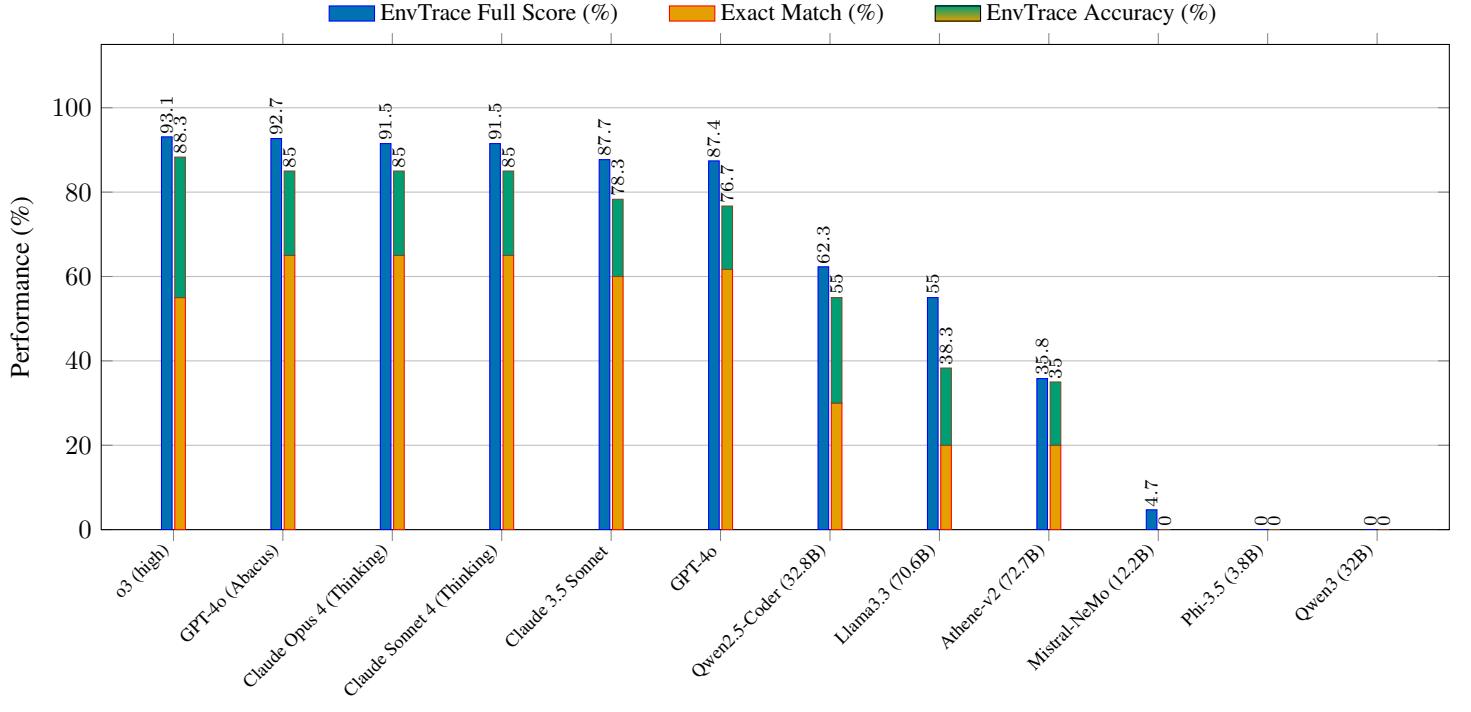


Figure S7: Example of the AI evaluator providing corrections.

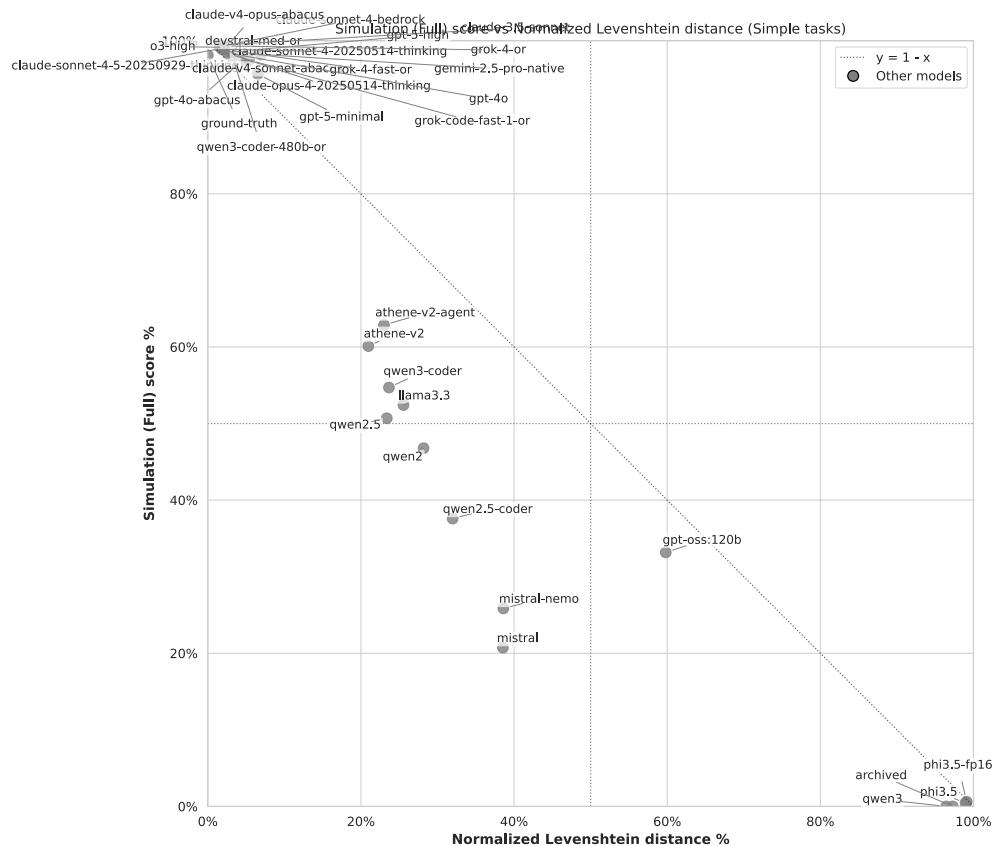


Figure S4: Plot of the semantic EnvTrace full score versus the normalized Levenshtein distance for simple-flow tasks.

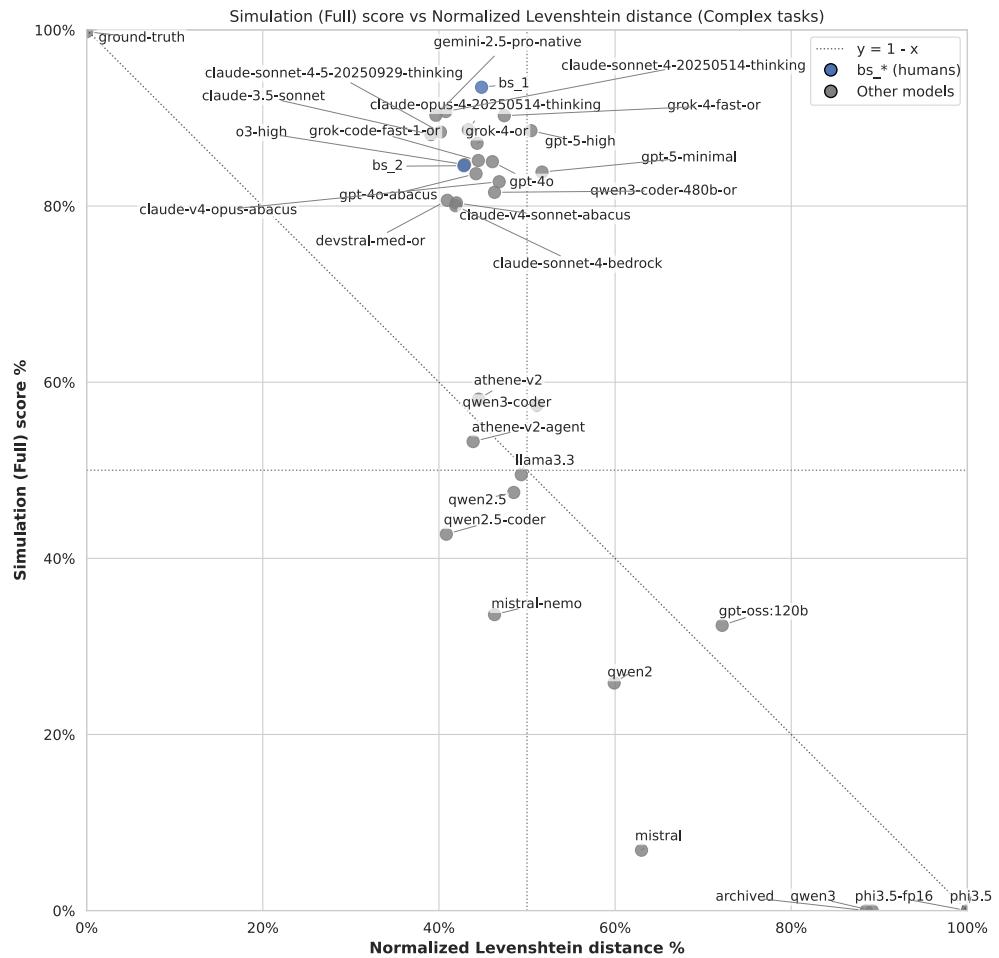


Figure S5: Plot of the semantic EnvTrace full score versus the normalized Levenshtein distance for complex-flow tasks.

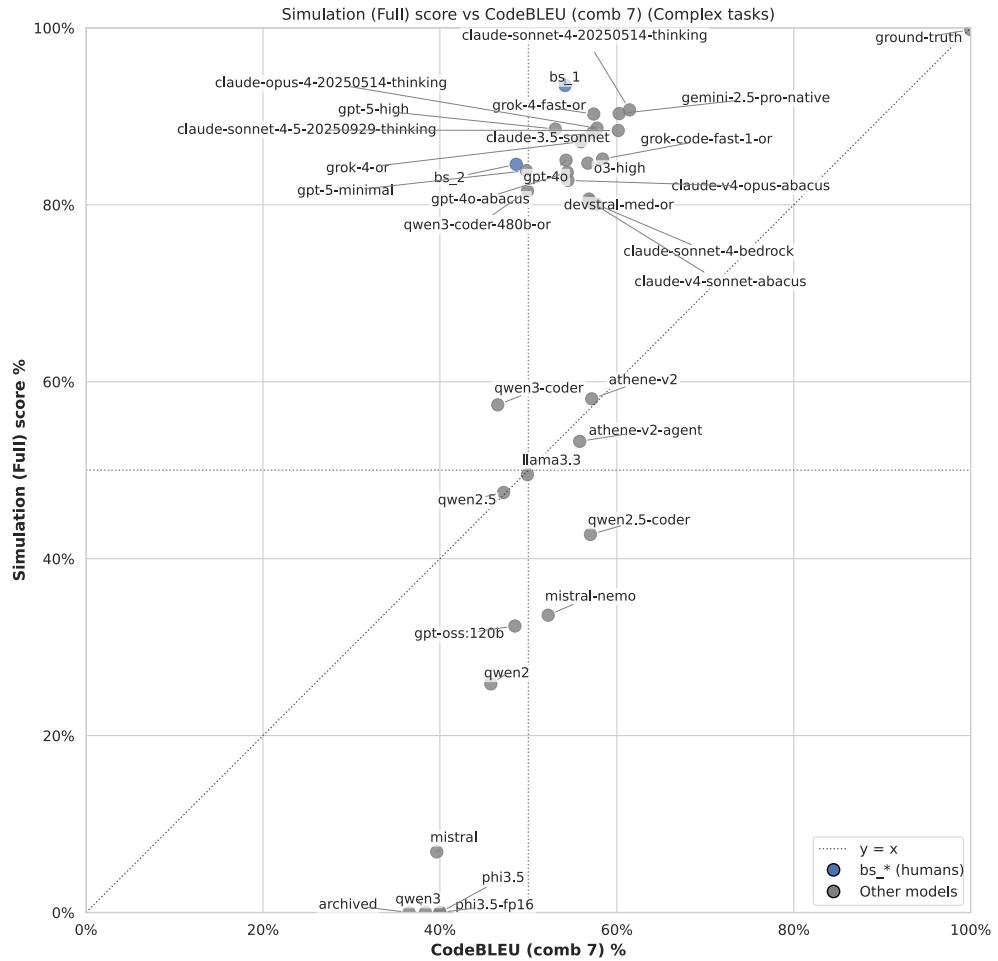


Figure S6: Plot of the semantic EnvTrace full score versus the syntactic CodeBLEU score for complex-flow tasks.