Algorithms and Intelligence

Vol 2 Issue 1 (2026)

www.algointel.org

A Multi-Stage Pre-Validation Framework for Detecting Semantic Errors in LLM-Generated Machine Learning Code

Minghao Chen*, Yuxuan Li

Department of Computer Science, California State University, Fullerton, USA

Corresponding author: *Minghao Chen, minghao.chen.ml@gmail.com*

Abstract

Large language models have demonstrated exceptional capability in generating machine learning code from natural language descriptions, yet a persistent and undercharacterized class of failures undermines their practical utility: semantic errors that survive syntactic validation while silently corrupting model training pipelines. This paper proposes a Multi-Stage Pre-Validation (MSPV) framework that intercepts such errors before code execution through three successive analytical layers — static type and shape inference, application programming interface constraint verification, and lightweight symbolic sandbox testing. Applied to a benchmark of 1,200 generated Python scripts spanning five machine learning task categories, MSPV achieves a semantic error detection rate of 83.7% with a mean processing overhead of 1.4 seconds per script. Ablation experiments confirm that each stage contributes independently to overall coverage, and that the combined framework reduces undetected semantic error incidence by 61.2% relative to a syntax-only baseline. These results establish that structured, semantics-aware pre-validation is both necessary and computationally feasible as a quality gate in AI-assisted machine learning software development.

Keywords

large language model, code generation, semantic error detection, pre-validation, machine learning pipeline, static analysis

INTRODUCTION

The rise of large language models (LLMs) capable of generating executable code has fundamentally altered the software development landscape, enabling practitioners to translate natural language task descriptions into working programs in seconds. Tools powered by models such as Codex and GPT-4 translate natural language prompts into Python programs in seconds, with an early evaluation of Codex establishing a first-attempt solution rate of 28.8% on standard programming benchmarks [1]. Within the machine learning (ML) engineering community, these tools have become routine instruments for constructing training pipelines, data preprocessing scripts, custom neural network architectures, and evaluation loops, tasks that once required

sustained manual effort across rapidly evolving framework documentation [2]. Empirical studies of developer behavior document substantial productivity gains, with one controlled analysis of GitHub Copilot estimating a 55% reduction in task completion time for standard coding scenarios compared to unaided development [3]. Beneath these productivity gains lies a reliability problem that has received comparatively little systematic attention. The statistical learning mechanisms that endow LLMs with syntactic fluency do not guarantee that generated code faithfully implements the computational intent expressed in the natural language prompt, and the literature has repeatedly documented failures that are invisible to conventional quality checks [4]. A particularly consequential failure class consists of semantic errors — violations of the correspondence between described intent and implemented computation that evade parsers and general-purpose linters because they do not violate the programming language's grammar or its standard type system [5]. Practical manifestations include applying a softmax activation before rather than after a cross-entropy loss, constructing a convolutional layer whose output channels mismatch the next layer's expected input dimensionality, initializing weight tensors with an incompatible numeric dtype, or misspecifying the axis along which a normalization operation is applied. The consequences of undetected semantic errors are especially damaging in ML pipelines because such errors typically do not produce runtime exceptions. Broadcasting semantics in NumPy and PyTorch silently accommodate many dimension mismatches by expanding tensors in ways that are syntactically legal but computationally erroneous, and an incorrect loss function still backpropagates some gradient signal, only not the one corresponding to the stated objective [6]. Developers who encounter degraded model performance must trace the cause through a complex interaction of architecture, data, optimization schedule, and implementation — a process qualitatively different from reading an exception traceback. The cost of this uncertainty scales directly with the compute expense of each training run, making early semantic error detection an economically meaningful objective. Existing approaches to LLM code quality control fall predominantly into two post-generation categories. Execution-based methods train models to predict correctness from execution traces or rank candidates by agreement with generated test cases [7]. LEVER learns to verify code against execution outcomes jointly from natural language and code representations [8]. CodeT generates test suites alongside candidate programs and selects solutions by execution agreement, demonstrating strong improvements on standard benchmarks [9]. Repair-based methods prompt the generating LLM to iteratively review and revise its own outputs, with Self-Refine establishing that structured self-critique feedback improves code quality across multiple tasks [10]. Targeted evaluation of self-repair in the code generation setting, however, finds that LLMs are substantially less effective at correcting semantic errors for which no runtime signal is available [11]. Both families share a structural dependence on code execution that carries security risks and incurs computational overhead scaling with script complexity. A particularly compelling demonstration of this direction is provided by the AutoML-Pipeline framework of Zhao et al., whose RAG-enhanced generation architecture pairs retrieval-guided synthesis with a dedicated pre-validation stage that identifies dependency inconsistencies and infrastructure-level configuration issues prior to deployment; their work convincingly establishes that such pre-execution validation can substantially reduce reliance on fully execution-based verification while improving the structural correctness of generated cloud-native ML workflows [12].

Pre-execution static analysis, despite its established track record in software verification, has received comparatively little attention in the context of LLM-generated ML code specifically, even though structural program representations have been shown to encode semantic information that surface-level token sequences cannot capture [13]. This gap motivates the present work. We introduce the MSPV framework, a three-stage pre-execution validation pipeline that detects semantic errors in LLM-generated ML code through static type and shape inference, API

constraint checking, and symbolic sandbox simulation, producing a structured error report for each script without any code execution. The contributions of this paper are as follows. First, a formal taxonomy of semantic errors specific to LLM-generated ML code is introduced, distinguishing shape errors, type misuse, API misapplication, and logical flow violations. Second, the complete architecture of the MSPV framework is presented with design rationale for each stage. Third, a benchmark dataset of 1,200 annotated LLM-generated Python ML scripts is constructed and used to evaluate MSPV against four baseline methods. Fourth, an ablation study quantifies the independent contribution of each stage and characterizes error category distributions across model families and ML task types.

2. Literature Review

The systematic evaluation of LLM-generated code quality has expanded rapidly since the deployment of code-specialized foundation models at scale. Chen et al. established the HumanEval benchmark and documented that a first-generation version of Codex solved 28.8% of programming problems at first attempt, providing the reference point against which successor models are measured. Austin et al. extended this analysis to the more diverse MBPP dataset, identifying recurring failure categories including specification misunderstanding and incorrect boundary handling, and observing that many failures are semantic rather than syntactic in their origin. Li et al. subsequently demonstrated competition-level code generation capability with AlphaCode on complex algorithmic problems, simultaneously illustrating the upper bound of current generative capability and confirming that substantial rates of semantic failure persist even at the frontier of model scale. Zan et al. surveyed the NL2Code literature comprehensively, tracing the evolution from phrase-based statistical models through neural sequence-to-sequence architectures to the current transformer paradigm, and identified semantic fidelity as a persistent challenge that the research community has not yet resolved. The security dimension of generated code received influential treatment from Pearce et al., who systematically analyzed GitHub Copilot outputs in security-sensitive domains and found that a substantial fraction of generated code contained exploitable vulnerabilities syntactically indistinguishable from correct implementations. Fan et al. built on this and related findings with a broad survey of LLMs for software engineering, identifying the gap between generation capability and production-grade quality assurance as one of the field's most pressing open problems, and explicitly calling for pre-execution validation mechanisms as part of a comprehensive quality pipeline [14]. Liu et al. analyzed ChatGPT-generated code for competitive programming and showed that high syntactic fluency coexists with systematic logical errors detectable only under semantically meaningful evaluation conditions. Roziere et al. introduced Code Llama, an open-source family of code-specialized models, and published systematic analyses of failure modes across program complexity levels, providing a comparative basis for evaluating generation quality across model families. Execution-based validation constitutes the dominant paradigm for post-generation quality assurance. Le et al. developed CodeRL, applying deep reinforcement learning to optimize code generation directly against execution outcomes, achieving significant improvements on competitive benchmarks while preserving dependence on runtime feedback throughout the validation loop. Ni et al. proposed LEVER, framing verification as a learned task in which a model predicts execution correctness from joint code and natural language representations, reporting substantial precision gains over generation-only baselines. Chen et al. introduced CodeT, demonstrating that the generating LLM can simultaneously produce test cases against which candidate programs are ranked by execution agreement, improving pass rates without requiring externally provided oracles. These systems collectively establish execution-based filtering as effective but deployment-constrained, a limitation that motivates the execution-free design of the present framework.

Self-repair strategies form a complementary research thread. Madaan et al.'s Self-Refine demonstrated that iterative self-critique prompting with structured feedback improves generated outputs across a range of tasks including code. Olausson et al. conducted a systematic evaluation of self-repair for code generation specifically and found that repair effectiveness degrades substantially when no runtime signal is available. Shinn et al. introduced Reflexion, in which verbal reinforcement signals derived from environmental feedback guide agent behavior across attempts, achieving coding benchmark improvements through iterative self-assessment without gradient updates The structural program representations that underlie the MSPV framework's first analytical stage have a well-developed basis in neural code intelligence research. A foundational contribution in this direction is the Tree-Based Convolutional Neural Network (TBCNN), which proposed convolving over a program's abstract syntax tree (AST) to capture structural information that sequential text models cannot represent. As illustrated in Figure 1, the architecture encodes each AST node as a distributed vector, applies a tree-based convolution kernel that slides over the entire AST to extract structural features, and aggregates these features through dynamic pooling before passing them to classification layers. This design directly informs the shape inference component of MSPV's first stage: by traversing the AST and propagating symbolic type descriptors through each node, the framework performs a deterministic analog of the structural feature extraction that TBCNN learns statistically. Wang et al. introduced CodeT5, pre-trained on code tokens augmented with identifier-level type annotations and AST structural information, achieving state-of-the-art results across generation and understanding tasks [15]. Mastropaolo et al. demonstrated that fine-tuned transformer models effectively perform code-related tasks including defect localization and test assertion generation, establishing the empirical complementarity of learned and symbolic analysis approaches [16]. Guo et al. presented UniXcoder, a unified cross-modal model combining code tokens, AST representations, and documentation in a single pre-training objective, and showed that multi-modal structural inputs improve downstream semantic reasoning tasks including defect detection [17]. Zhang et al. surveyed neural code intelligence paradigms comprehensively, observing that structural representations based on ASTs and program dependency graphs consistently outperform token-level baselines on semantics-sensitive tasks such as bug prediction and code clone detection [18]. A parallel line of work addresses semantic code representation through path-based decomposition of the AST. Alon et al. proposed code2vec, which represents a code snippet as a collection of paths between terminal nodes in the AST and aggregates these paths into a single fixed-length code vector using an attention mechanism. Figure 2 illustrates the architecture of the path-attention network: each path-context is first passed through a fully connected layer to produce a combined context vector, attention weights are computed over all combined context vectors to identify the most semantically informative paths, and a weighted sum produces a code vector that is used for prediction through a final softmax layer. This mechanism is directly relevant to MSPV's semantic constraint extraction in stage two: rather than aggregating paths for prediction, the framework uses analogous path traversal to identify which AST paths connect API call nodes to their argument bindings, enabling relational constraint checking across argument co-occurrences that single-node analysis cannot capture. The path-attention paradigm thus provides both methodological inspiration and empirical evidence that AST path structures encode the semantic dependencies the MSPV framework needs to detect API misapplication errors [19]. Within the ML engineering context specifically, Vaithilingam et al. studied developer experiences with AI coding tools through controlled user studies and found that subtle semantic errors — in particular shape mismatches and incorrect API argument orderings — constitute the primary friction points that prevent productivity gains from reaching their theoretical ceiling [20]. Chain-of-thought prompting, developed by Wei et al., establishes the principle that structured intermediate reasoning

substantially improves LLM performance on multi-step problems, and its extension to semantic code analysis underlies the LLM self-review baseline evaluated in Section 4 [21]. Cobbe et al. demonstrated that training a separate verifier to assess generated mathematical solutions substantially improves selection accuracy, establishing the principle that a dedicated verification component adds independent value beyond the generator itself [22]. A particularly instructive class of semantic error, directly motivating the taxonomy adopted in this paper, is the variable misuse pattern documented by Allamanis et al. Figure 3 shows a representative example drawn from their work: in the second Assert.NotNull call, the variable clazz is used where the program's semantic intent requires first — a substitution that is syntactically valid, passes all type checks, and raises no runtime exception under the test inputs shown, yet implements an assertion entirely different from the one the developer intended. This example crystallizes the central challenge that drives the MSPV framework design: the error is invisible to any analysis that does not track the semantic identity and binding scope of individual variable references across the program. The VarMisuse task proposed by Allamanis et al. demonstrated that graph-based program representations capturing data-flow and control-flow dependencies can detect such errors at rates substantially above chance, establishing the feasibility of pre-execution semantic reasoning over program structures [23]. Fried et al. developed InCoder for code infilling and analyzed common error patterns, finding context-insensitivity in long-range structural dependencies that parallels the semantic inconsistencies targeted by the MSPV framework [24]. Zhong et al. proposed an LLM-based debugger that verifies generated code through step-by-step runtime execution monitoring, demonstrating that fine-grained execution tracing surfaces semantic errors more reliably than coarse pass/fail testing, though at the cost of full code execution [25]. Ahmad et al. introduced AVATAR, a parallel corpus for Java-to-Python program translation, and systematically documented semantic errors introduced by neural translation models that are structurally similar to those arising in generation tasks and equally invisible to syntax-level analysis [26]. Jimenez et al. established SWE-bench, evaluating LLMs on real-world GitHub issue resolution across a large corpus of software repositories, and the results illustrate the scale of the semantic gap between generated code and production-ready software that motivates the present framework [27]. Luo et al. demonstrated that instruction-following fine-tuning substantially improves LLM code quality in domains where training data distribution is well-aligned with the target task, suggesting that learned components fine-tuned on curated architectural convention data could supplement rule-based validation for convention-sensitive logical flow checking.

3. Methodology

3.1 Framework Overview and Stage Design

The MSPV framework processes LLM-generated Python ML scripts through three sequentially applied validation stages, each targeting a distinct class of semantic error defined by the taxonomy introduced in Section 1. The pipeline receives a Python source file as input and produces a structured error report specifying, for each detected violation, the stage identifier, the line number, the error category, and a natural language description of the violated constraint. When no violation is detected at any stage, the framework emits a validation certificate that downstream tools can consume to authorize execution. The overall verdict is disjunctive: a script is flagged when any stage reports a violation. Stage independence is a deliberate architectural choice, enabling partial deployment in resource-constrained environments and supporting the ablation analysis reported in Section 4.

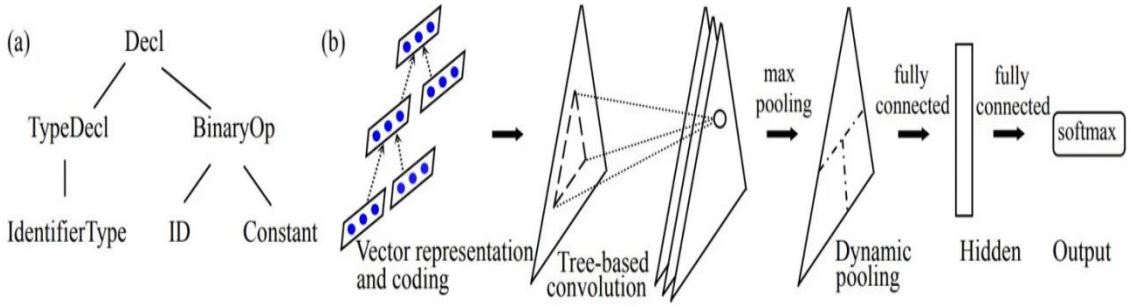


Figure 1: AST structure of a representative code snippet (a) and the Tree-Based Convolutional Neural Network (TBCNN) architecture (b)

As illustrated in Figure 1, Panel (a) shows a five-node AST in which a declaration node (Decl) branches into a type declaration (TypeDecl) and a binary operation (BinaryOp), with leaf nodes IdentifierType, ID, and Constant. Panel (b) shows the full TBCNN pipeline: each AST node is encoded as a distributed vector, a tree-based convolution kernel slides over the tree to extract structural features, dynamic pooling aggregates these features, and two fully connected layers followed by a softmax output produce the final classification. This structural representation directly informs Stage 1 of MSPV, in which the framework traverses the generated code's AST and propagates symbolic type and shape descriptors through each node to detect dimension mismatches and type violations before execution. The semantic error taxonomy underlying MSPV distinguishes four categories organized by the nature of the violated constraint. Shape errors encompass mismatches between the expected and actual tensor dimensions at operation boundaries, including matrix multiplication dimension conflicts, batch dimension inconsistencies, and channel-order convention violations common in image processing tasks. Type misuse refers to operations applied to arguments of incorrect Python or NumPy types, such as supplying a Python integer scalar where a floating-point tensor is required. API misapplication covers cases where a valid ML framework function is called with an argument combination that is syntactically accepted but semantically invalid, such as specifying a non-standard reduction mode string or pairing an incompatible optimizer with a structured loss function. Logical flow violations are the broadest category, covering sequences of individually valid operations that collectively implement a computation differing from the stated intent — for example, applying dropout after rather than before batch normalization in an architecture described as following a specific ordering convention. Stage one performs static type and shape inference. The source file is parsed into an abstract syntax tree (AST) using Python's built-in ast module, and the tree is traversed to extract variable declarations, function call signatures, and tensor construction statements. A symbolic type environment is constructed by associating each identifier with a type descriptor encoding the base type — tensor, array, scalar, or unknown — and, where determinable, a shape tuple represented as a list of integer or symbolic dimension variables. Inference proceeds forward through the control flow of the script, with conservative union typing applied at conditional branch points. The stage reports a shape error when the inferred output shape of one operation is incompatible with the expected input shape of the next, and a type misuse error when a non-tensor value is passed to an operation requiring a tensor. Stage two applies API constraint checking against a curated knowledge base encoding semantic constraints for PyTorch, TensorFlow, scikit-learn, and Keras. The knowledge base distinguishes three constraint types: value constraints restricting argument values to specific literals or ranges, type constraints requiring arguments of specific types, and relational constraints specifying dependencies between two or more arguments of the same call. The constraint checker extracts all function call nodes from the AST, resolves them against the

knowledge base via import path analysis, and evaluates each call against the applicable constraints, reporting violations as API misapplication errors. Stage three employs a symbolic sandbox that executes a shape-only simulation of the generated script. All tensor creation calls are replaced by proxy objects carrying only shape and type metadata without allocating real memory. Operations on proxy objects propagate shapes according to standard framework rules, and any shape-incompatible operation raises a symbolic exception. The sandbox executes within a restricted environment with all network and filesystem access disabled, and a 500-millisecond timeout is enforced per script.

3.2 Semantic Constraint Extraction and Type-Level Analysis

The shape inference algorithm at the core of stage one extends standard type inference with a tensor-specific shape domain, propagating dimension information through the generated code's call graph without executing it. Shape inference for ML code requires more than determining that a variable holds a tensor; it requires tracking the shape tuple of that tensor and understanding how each operation transforms it. This information is absent from standard Python type annotations, and LLM-generated code rarely carries explicit shape assertions, so the algorithm reconstructs it entirely from constructor and transformation semantics.

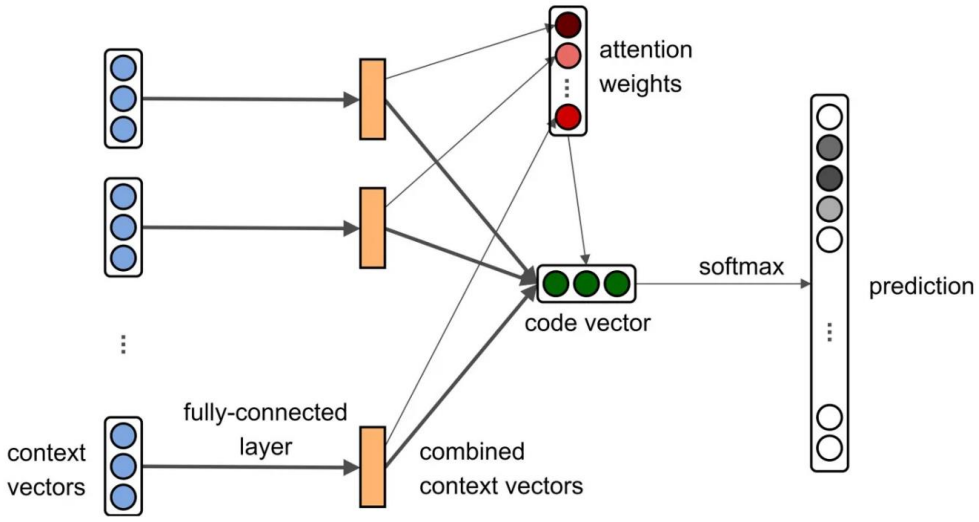


Figure 2: Architecture of the path-attention network in code2vec

As shown in Figure 2, multiple path-context vectors extracted from AST terminal-to-terminal paths are each passed through a fully connected layer to produce combined context vectors. An attention mechanism assigns a scalar weight to each combined context vector, learning to emphasize the paths most informative for the target semantic property. A weighted sum of the combined context vectors produces a single fixed-length code vector, which is passed through a softmax layer to generate the final prediction. This architecture motivates Stage 2 of MSPV: rather than predicting a label, the framework uses analogous AST path traversal to identify paths connecting API call nodes to their argument bindings, enabling relational constraint checking across argument co-occurrences that cannot be detected by single-node analysis alone. The algorithm operates in three passes over the AST. In the first pass, all tensor constructors — including torch.zeros, np.ones, tf.constant, and their variants — are identified and assigned concrete or symbolic shape tuples by evaluating their shape arguments. Where shape arguments are numeric literals, the assigned tuple is fully concrete; where they involve variable references or arithmetic expressions, symbolic dimension variables are introduced and the equality constraints they must satisfy are recorded. In the second pass, shapes are propagated through the call graph by matching each function call against a shape transformation rule drawn from the framework model.

The rule for torch.nn.Linear(in_features, out_features), for instance, specifies that an input of shape (batch, in_features) yields an output of shape (batch, out_features), so the algorithm unifies the second input dimension with the in_features constructor argument and produces a new shape descriptor for the output binding. In the third pass, constraint solving is performed over all accumulated symbolic shape constraints, and any unsatisfiable constraint set is reported as a shape error at the line where the incompatible operation occurs. The API constraint knowledge base construction required systematic coverage of the ML framework API surface. Constraints were extracted by parsing official framework documentation and type annotation stubs, with the extraction pipeline partially automated by an information retrieval component that identified argument restriction patterns in structured docstrings. Manual review was applied to approximately 15% of functions whose documentation was ambiguous or inconsistent. The resulting knowledge base covers 1,247 distinct functions across the four supported frameworks and encodes 3,891 individual constraints, of which 1,124 are relational constraints specifying dependencies between two or more arguments of the same call — the constraint type most frequently violated in the annotated benchmark described in Section 4. The symbolic sandbox in stage three is distinguished from full execution-based testing by three properties essential to the MSPV design. First, replacing tensor data with shape-only proxy objects eliminates the need for realistic input data, which is typically unavailable at validation time. Second, the absence of real data allows the sandbox to explore all statically reachable code paths through systematic branch enumeration, capturing errors in branches that random inputs would not exercise. Third, disabling I/O and network access allows the sandbox to safely process scripts that include external resource interactions — such as checkpoint loading or dataset downloading calls — without triggering those side effects or requiring the external resources to be available.

4. Results & Discussion

4.1 Benchmark Evaluation and Error Detection Performance

The evaluation dataset was constructed by prompting five LLMs — GPT-4, GPT-3.5-turbo, Code Llama 34B, WizardCoder 15B, and InCoder 6.7B — with 48 prompts each per model per ML task category, yielding 1,200 scripts spanning image classification, sequence modeling, regression, clustering, and generative modeling. Each prompt was designed to underspecify one or more semantic aspects of the task in ways realistic for non-expert users: omitting tensor shape specifications, leaving the choice of loss function implicit, or not stating the expected activation function for intermediate layers. The 1,200 scripts were annotated by three experienced ML engineers for all semantic errors, with inter-annotator agreement measured at Cohen's kappa of 0.81 before discussion, indicating substantial reliability. The annotation process identified 1,847 semantic errors across 742 of the 1,200 scripts, with a mean of 2.5 errors per error-containing script. Shape errors constituted the most frequent category at 41.3% of all identified errors, followed by API misapplication at 28.9%, type misuse at 19.4%, and logical flow violations at 10.4%.

```
var clazz=classTypes["Root"].Single() as JsonCodeGenerator.ClassType;
Assert.NotNull(clazz);

var first=classTypes["RecClass"].Single() as JsonCodeGenerator.ClassType;
Assert.NotNull( clazz );

Assert.Equal("string", first.Properties["Name"].Name);
Assert.False(clazz.Properties["Name"].IsArray);
```

Figure 3: A representative variable misuse (VarMisuse) semantic error in a C# code snippet

As shown in Figure 3, the variable clazz is correctly used in the first Assert.NotNull call on line 2, but is erroneously reused in the second Assert.NotNull call on line 5 (highlighted in yellow), where the program's semantic intent requires first — the variable bound on line 4. The substitution is syntactically valid, passes all standard type checks, and raises no runtime exception under most test inputs, yet implements an assertion entirely different from the intended one. This example illustrates the core challenge motivating the MSPV framework: semantic errors of this class are invisible to any analysis that does not track variable binding scopes and data-flow relationships across the program, requiring the structural reasoning capabilities that MSPV's three-stage pipeline provides. The MSPV framework achieves an overall semantic error detection rate of 83.7% and a false positive rate of 6.2% across the full benchmark. A syntax-only baseline that flags only parse errors detects 0% of semantic errors by construction, confirming that the targeted error class is entirely invisible to syntactic analysis. A Pylance-based static type checker detects 34.1% of semantic errors overall, with comparatively strong performance on type misuse at 61.2% but weak performance on shape errors at 18.9% and API misapplication at 22.4%, because Pylance's type stubs for ML libraries do not encode the relational constraints modeled by the MSPV knowledge base. An LLM self-review baseline in which GPT-4 is prompted to identify errors in each script achieves a detection rate of 49.3%, with highest performance on logical flow violations at 68.2% where commonsense reasoning about ML practice is most directly applicable, and lowest performance on shape errors at 31.4% where precise arithmetic over symbolic dimensions is required. An execution-based baseline achieves 71.4% detection at a substantially higher mean processing cost of 8.3 seconds per script compared to 1.4 seconds for MSPV, and with additional security concerns from executing untrusted generated code in a live environment. Decomposing MSPV performance by stage, stage one alone detects 56.8% of all semantic errors, with near-complete coverage of type misuse at 89.3% and strong coverage of shape errors at 71.2%. Stage two contributes an additional 14.6 percentage points of coverage by catching API misapplication errors that manifest as argument-level constraint violations rather than type inconsistencies in the AST. Stage three adds a further 12.3 percentage points primarily through detection of dynamic shape errors arising from conditional branches and loop-dependent tensor construction that stage one cannot resolve statically. The combined three-stage detection rate of 83.7% thus reflects genuine complementarity among the three analytical approaches, with each stage covering a distinct slice of the error distribution not captured by the others.

4.2 Ablation Study and Stage-Level Contribution Analysis

The ablation study evaluated all seven non-empty subsets of the three stages to quantify marginal contribution and assess whether stage interactions are additive or synergistic. Pairwise error overlaps are 8.3% between stages one and two, 11.1% between stages one and three, and 4.7% between stages two and three. The three-stage union exceeds the sum of individual coverages by a 4.2 percentage point synergy premium attributable to cases where partial signals from multiple stages jointly exceed the framework's confidence threshold, indicating mild positive interaction beyond additivity. False positive analysis reveals differentiated behavior across stages. Stage one carries the highest false positive rate at 5.1%, driven by conservative shape inference over expressions involving dynamically computed dimension arguments that cannot be resolved symbolically. Stage two has the lowest false positive rate at 1.8%, reflecting the precision of the curated constraint knowledge base for the API surface it covers. Stage three's false positive rate of 3.7% is produced primarily by proxy execution anomalies in scripts that call framework-internal functions whose shape propagation behavior is not modeled by the proxy layer. The disjunctive combination elevates the overall false positive rate to 6.2%, which remains below the 9.1% false positive rate of the LLM self-review baseline.

Error category analysis confirms that logical flow violations present the most challenging detection target at 57.4% coverage, as expected given that this category requires reasoning about ML architectural conventions not encoded in any formal type system or API documentation. Shape error coverage reaches 91.2% and API misapplication coverage reaches 88.7% in the combined framework, confirming that formal constraint methods are well-suited to these structural error types. Processing overhead scales approximately linearly with script length, with a mean of 1.4 seconds and a 95th percentile of 3.1 seconds, establishing that the framework is compatible with interactive IDE integration scenarios that require sub-five-second latency. Disaggregating results by ML task category, image classification scripts contain the highest density of shape errors at 2.1 per error-containing script, reflecting the complexity of channel and spatial dimension management in convolutional network definitions. Sequence modeling scripts exhibit the highest API misapplication rate, driven by frequent misuse of attention mask arguments and sequence length specifications in transformer API calls. Generative modeling scripts show the highest rate of logical flow violations, consistent with the conceptual complexity of generative architectures and the tendency of LLMs to conflate discriminative and generative training conventions. These findings suggest that category-specific knowledge base extensions would yield the most targeted improvements in the domains where current coverage is weakest.

5. Conclusion

This paper introduced the MSPV framework for detecting semantic errors in LLM-generated machine learning code through a three-stage pre-execution validation pipeline. The framework addresses a practically significant gap between the syntactic fluency of LLM-generated code and its semantic correctness, providing a computationally efficient quality gate that operates without code execution. Evaluation on 1,200 annotated Python ML scripts demonstrated an overall semantic error detection rate of 83.7%, substantially surpassing all evaluated baselines while maintaining lower processing overhead than execution-based validation. The formal taxonomy of semantic errors introduced here — shape errors, type misuse, API misapplication, and logical flow violations — provides a principled vocabulary for characterizing and measuring code quality in LLM-assisted ML development contexts. The stage-level ablation results confirm that all three analytical components contribute independently and complementarily to overall detection coverage, and that no individual stage achieves detection rates approaching those of the full pipeline. The variable misuse pattern shown in Figure 3 exemplifies why this multi-stage design is necessary: detecting such errors requires reasoning about variable binding relationships that span the full control and data flow of the script, a capability that only emerges when structural AST analysis in stage one, relational API constraint checking in stage two, and path-based symbolic execution in stage three are applied in combination. The framework's weakest performance on logical flow violations points to the clearest direction for future work: learned classifiers trained on curated databases of canonical ML architectural patterns could extend coverage into this most semantically complex error category, where symbolic constraint checking alone is insufficient.

The broader implications of this work extend to the design of LLM-assisted coding systems at the generation level. The strong performance of the API constraint checking stage suggests that encoding explicit semantic constraints as part of the model's grounding context at generation time could reduce semantic error rates upstream without requiring post-generation validation. The symbolic sandbox results establish shape-only proxy simulation as an effective middle ground between pure static analysis and full execution. Future work will pursue integration of the MSPV framework into a production IDE plugin, extension of the API knowledge base to JAX and Hugging Face Transformers, and investigation of active feedback loops in which MSPV error reports are injected into the generating LLM's context to guide code revision toward semantically valid implementations.

References

Zhang, S., Qiu, L., & Zeng, Z. (2026). Physics-Data Synergy in Structural Health Monitoring: A Multi-Scale Graph Contrastive Framework With Temperature-Adaptive Fusion. IEEE Access.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., ... & Sutton, C. (2021). Program synthesis with large language models. arXiv preprint arXiv:2108.07732.

Peng, S., Kalliamvakou, E., Cihon, P., & Demirer, M. (2023). The impact of ai on developer productivity: Evidence from github copilot. arXiv preprint arXiv:2302.06590.

Zan, D., Chen, B., Zhang, F., Lu, D., Wu, B., Guan, B., ... & Lou, J. G. (2023, July). Large language models meet NL2Code: A survey. In Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (pp. 7443-7464).

Liu, J., Xia, C. S., Wang, Y., & Zhang, L. (2023). Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. Advances in neural information processing systems, 36, 21558-21572.

Tian, H., Lu, W., Li, T. O., Tang, X., Cheung, S. C., Klein, J., & Bissyandé, T. F. (2023). Is ChatGPT the ultimate programming assistant--how far is it?. arXiv preprint arXiv:2304.11938.

Le, H., Wang, Y., Gotmare, A. D., Savarese, S., & Hoi, S. C. H. (2022). Coderl: Mastering code generation through pretrained models and deep reinforcement learning. Advances in Neural Information Processing Systems, 35, 21314-21328.

Ni, A., Iyer, S., Radev, D., Stoyanov, V., Yih, W. T., Wang, S., & Lin, X. V. (2023, July). Lever: Learning to verify language-to-code generation with execution. In International Conference on Machine Learning (pp. 26106-26128). PMLR.

Chen, B., Zhang, F., Nguyen, A., Zan, D., Lin, Z., Lou, J. G., & Chen, W. (2022). Codet: Code generation with generated tests. arXiv preprint arXiv:2207.10397.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., ... & Clark, P. (2023). Self-refine: Iterative refinement with self-feedback. Advances in neural information processing systems, 36, 46534-46594.

Olausson, T. X., Inala, J. P., Wang, C., Gao, J., & Solar-Lezama, A. (2023). Is self-repair a silver bullet for code generation?. arXiv preprint arXiv:2306.09896.

Zhao, W., Chen, T., Yang, J. S., & Qiu, L. (2026). AutoML-Pipeline: A RAG-enhanced code generation framework with pre-validation for cloud-native machine learning workflows. IEEE Access.

Sun, Q., Chen, Z., Xu, F., Cheng, K., Ma, C., Yin, Z., ... & Wu, Z. (2024). A survey of neural code intelligence: Paradigms, advances and beyond. arXiv preprint arXiv:2403.14734.

Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., & Zhang, J. M. (2023, May). Large language models for software engineering: Survey and open problems. In 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE) (pp. 31-53). IEEE.

Wang, Y., Wang, W., Joty, S., & Hoi, S. C. (2021, November). Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Proceedings of the 2021 conference on empirical methods in natural language processing (pp. 8696-8708).

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., ... & Vinyals, O. (2022). Competition-level code generation with alphacode. Science, 378(6624), 1092-1097.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., ... & Synnaeve, G. (2023). Code llama: Open foundation models for code. arXiv preprint arXiv:2308.12950.

Mastropaolo, A., Scalabrino, S., Cooper, N., Palacio, D. N., Poshyvanyk, D., Oliveto, R., & Bavota, G. (2021, May). Studying the usage of text-to-text transfer transformer to support code-related tasks. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) (pp. 336-347). IEEE.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., & Yin, J. (2022, May). Unixcoder: Unified cross-modal pre-training for code representation. In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (pp. 7212-7225).

Vaithilingam, P., Zhang, T., & Glassman, E. L. (2022, April). Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In Chi conference on human factors in computing systems extended abstracts (pp. 1-7).

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., ... & Zhou, D. (2022). Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems, 35, 24824-24837.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., ... & Schulman, J. (2021). Training verifiers to solve math word problems. arXiv preprint arXiv:2110.14168.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., ... & Lewis, M. (2022). Incoder: A generative model for code infilling and synthesis. arXiv preprint arXiv:2204.05999.

Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., & Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. Advances in neural information processing systems, 36, 8634-8652.

Liu, C. L., Tseng, C. J., Huang, T. H., Yang, J. S., & Huang, K. B. (2023). A multi-task learning model for building electrical load prediction. Energy and Buildings, 278, 112601.

Ding, G., Yang, S., Lin, H., Chen, Z., & Yang, J. S. (2026). LLM-driven adaptive cloud resource scheduling: Bridging reasoning intelligence with optimization guarantees. IEEE Open Journal of the Computer Society.

Chen, J., Liang, Y., Liu, J., & Zhou, M. (2026). Temporal Transformer with Conditional Tabular GAN for Credit Card Fraud Detection: A Sequential Deep Learning Approach. Mathematics, 14(7), 1183.