INTERNATIONAL JOURNAL OF MACHINE LEARNING AND ENGINEERING

Vol 2 Issue 1 (2026)

https://ijmle.org/

Improving the Reliability of LLM-Generated Training Pipelines with Feedback-Driven Iterative Code Refinement

Haoyu Lin
Department of Computer Science, University of Massachusetts Lowell, United States
Chenxi Guo
School of Computing, DePaul University, United States
Jiaqi Zhou
Department of Computer Science, University of Massachusetts Lowell, United States
Corresponding author: Haoyu Lin, haoyulin.ai@163.com
Abstract:

The emergence of large language models (LLMs) as automated code generation tools has introduced compelling opportunities for constructing machine learning (ML) training pipelines from natural language specifications. However, LLM-generated pipelines frequently suffer from syntactic errors, API incompatibilities, data flow mismatches, and numerical instability that undermine reliability in deployment. This paper proposes the Feedback-Driven Iterative Code Refinement (FDICR) framework, which integrates a containerized execution sandbox, a hierarchical error taxonomy, and a context-aware prompt reformulation module to enable structured multi-round refinement of LLM-generated training pipelines. The design of FDICR's iterative loop draws conceptual inspiration from controller-based architecture search, wherein a generator receives performance signals from an execution environment and updates its outputs accordingly. The multi-stage pipeline structure incorporates meta-level initialization, Bayesian optimization, and ensemble validation components that parallel established automated machine learning (AutoML) architectures. Experimental evaluation across three benchmark tasks demonstrates that FDICR reduces critical pipeline failures by 78.3% and improves first-pass execution success rates from 42.1% to 89.6% relative to single-pass baselines, yielding speedup magnitudes consistent with neural-augmented program synthesis systems. The framework supports PyTorch, TensorFlow, and scikit-learn backends and introduces a novel pipeline reliability scoring metric.
Keywords: large language models, code generation, training pipeline automation, iterative refinement, feedback mechanisms, machine learning reliability

INTRODUCTION

The rapid maturation of large language models (LLMs) has fundamentally reshaped the landscape of automated software development, with particularly consequential implications for the construction and deployment of machine learning (ML) training pipelines. Modern code-oriented LLMs have demonstrated a remarkable capacity to translate natural language specifications into

executable programs, promising to reduce substantially the manual engineering effort required to design, implement, and debug end-to-end ML workflows [1]. Yet despite impressive benchmark performance, the transition from natural language intent to reliable, production-ready training pipelines remains a formidable challenge. LLM-generated code frequently exhibits a characteristic set of failure modes, including incorrect library imports, tensor shape mismatches, improperly sequenced preprocessing operations, and numerically unstable training configurations, each of which can independently prevent a pipeline from executing correctly and collectively erode confidence in automated generation as a practical approach. The core limitation addressed in this work is not merely that LLMs generate erroneous code, a fact that is well-documented and broadly acknowledged in the field, but rather that the dominant paradigm in automated code generation treats the generation process as a fundamentally one-shot procedure [2]. In single-pass generation, a model is prompted once, a code artifact is produced, and the result is either accepted or discarded without systematic mechanisms for incorporating execution evidence into subsequent correction cycles. This approach fails to exploit one of the most informative signals available in any software engineering context: runtime error messages, execution traces, and test outcomes that can precisely characterize the nature of a failure and guide targeted remediation. Notably, the AutoML-Pipeline framework proposed by Zhao et al. offers a compelling parallel in the cloud-native setting: by combining retrieval-guided synthesis with a carefully designed simulated validation stage, their system insightfully detects configuration and dependency failures before deployment, and their results make a strong case for embedding execution-derived feedback directly into the generation loop—a principle that closely aligns with and helps motivate the iterative refinement approach pursued here [3]. The absence of such feedback loops is especially costly in ML training pipeline generation, where errors are frequently interdependent and a misconfigured data loader may propagate downstream through model initialization and optimizer setup before manifesting as a training loop failure, obscuring the true origin of the fault. Recent developments in self-refinement and reflection-based reasoning have begun to address this limitation at a general level. Frameworks built around iterative self-critique and revision have demonstrated that LLMs can progressively improve their outputs when structured feedback is systematically incorporated into the generation process, achieving meaningful performance gains across tasks including mathematical reasoning, dialogue, and code debugging. These results establish the conceptual viability of iterative correction but have not been specifically adapted to the unique requirements of ML training pipeline generation, which involves managing complex component dependency graphs, navigating the versioned and rapidly evolving APIs of major ML frameworks, and ensuring that configurations remain numerically stable across diverse hardware environments [4]. This paper addresses this gap by introducing the Feedback-Driven Iterative Code Refinement (FDICR) framework, a specialized system that augments LLM-based pipeline generation with a structured multi-round refinement protocol tailored specifically to the demands of ML training code. FDICR comprises three novel components: a containerized execution sandbox that captures granular multi-level runtime diagnostics, a hierarchical error taxonomy that classifies pipeline failures into syntactic, API compatibility, data flow, and numerical stability categories, and a prompt reformulation module that translates diagnosed errors into precisely targeted correction instructions. By iterating across these components for up to five refinement rounds, the framework substantially increases the probability that a generated pipeline reaches successful execution while maintaining fidelity to the original specification. Empirical evidence confirms that FDICR achieves statistically significant improvements in pipeline reliability and execution success rate relative to both single-pass generation baselines and existing general-purpose self-repair frameworks [5]. These findings carry direct practical relevance for the deployment of LLM-

assisted tools within AutoML systems and enterprise data science workflows where pipeline reliability is a nonnegotiable operational requirement.

2. Literature Review

The intellectual background of this work spans three intersecting research domains: LLM-based code generation, iterative self-refinement and self-correction mechanisms, and automated ML pipeline construction. Understanding the contributions and limitations of prior work across these domains is essential for situating the specific advances introduced by FDICR. The investigation of LLM-based code generation has progressed substantially since early benchmark studies established that language models trained at scale on source code corpora could address competitive programming problems at a meaningful success rate. Competition-level code generation systems have pushed this boundary further, demonstrating performance approximating the median percentile of human competitive programmers through large-scale training combined with sophisticated candidate re-ranking strategies [6]. Alongside competition-focused systems, dialogue-oriented code generation models introduced multi-turn conversational paradigms that allow users to iteratively narrow and refine pipeline specifications through incremental natural language exchanges, improving alignment between intent and generated implementation [7]. These advances collectively established that LLM code generation is technically credible but surfaced a persistent gap between benchmark performance and practical production reliability.

The reliability limitations of LLM-generated code have been systematically characterized in deployment studies. Research examining AI-assisted developer workflows found that while developers complete tasks more quickly with LLM assistance, the correctness and maintainability of generated code varies substantially with task complexity, codebase context, and the specificity of the natural language prompt [8]. A complementary study evaluating LLM pair programming tools found that generated code required manual correction in the majority of tested scenarios, with correctness rates declining sharply for tasks involving numerical computation, framework-specific APIs, or complex data flow dependencies [9]. These findings are consistent with the phenomenon of confident hallucination, wherein LLMs produce syntactically valid and superficially plausible code that nonetheless contains subtle logical or numerical errors invisible to casual inspection.

The development of specialized code-oriented LLMs has partially addressed reliability concerns. Cross-modal pretraining approaches that align representations of code, natural language, and abstract syntax trees have demonstrated improvements on both code understanding and generation benchmarks, enabling more accurate handling of API semantics and type constraints [10]. Open-source code generation models have further extended coverage across diverse programming languages and framework versions, demonstrating that capable code generation does not require proprietary large-scale infrastructure [11]. More recent models have introduced fill-in-the-middle training objectives and curriculum-scheduled pretraining that better align generated code with practical editing and debugging workflows, narrowing the gap between single-pass generation quality and production-ready code [12]. The theoretical foundations for iterative refinement in language model generation trace to structured reasoning approaches that demonstrated the importance of intermediate reasoning steps for complex task performance. Chain-of-thought prompting established that eliciting step-by-step reasoning substantially improves model performance on multi-step problems, providing the conceptual basis for subsequent frameworks that decompose complex generation tasks into sequential diagnostic and corrective reasoning stages [13]. Instruction tuning with reinforcement learning from human feedback demonstrated that alignment training substantially improves the controllability and directability of LLM outputs, with direct implications for designing feedback-responsive generation systems in which correction instructions must be reliably followed across multiple refinement rounds [14]. Self-consistency decoding extended the reliability of complex generation by sampling multiple independent outputs

and aggregating through majority voting, establishing that ensemble-based approaches provide a complementary route to reliability improvement that can be combined with iterative refinement [15]. The application of iterative refinement specifically to code generation has attracted increasing research attention. Active prompting with chain-of-thought demonstrated that adaptively selecting the most informative exemplars based on model uncertainty substantially improves the consistency of complex multi-step reasoning, informing the design of FDICR's context-aware prompt reformulation module [16]. A self-editing framework in which LLMs use code interpreter feedback to iteratively revise generated programs demonstrated improvements on several standard code generation benchmarks, establishing execution-grounded refinement as a practically viable approach [17]. The Jigsaw framework combined LLM generation with program analysis tools to improve the correctness of code synthesized from natural language specifications, demonstrating that hybrid approaches integrating static analysis with generative models can outperform pure LLM approaches on complex synthesis tasks [18]. Beyond code generation, research on LLM-based autonomous agents has demonstrated that grounding agent decisions in directly executable code substantially improves performance on complex sequential planning and information retrieval tasks, motivating FDICR's architectural decision to use code execution output rather than abstract symbolic feedback as the primary signal in the refinement loop [19]. Comprehensive surveys of prompting methods have provided a taxonomy of strategies for prompt-based task adaptation, informing the structured prompt reformulation module that translates diagnosed errors into targeted correction instructions within FDICR [20]. Holistic evaluation frameworks for language model capabilities have established multi-dimensional assessment protocols that decompose code generation performance into correctness, robustness, and efficiency dimensions, directly informing the experimental design used to evaluate FDICR in this study [21]. Instruction-tuned code LLMs evaluated on rigorous generation benchmarks have demonstrated that alignment training substantially improves generation reliability, confirming that techniques from natural language alignment are directly transferable to code-specific reliability improvement and lending empirical support to FDICR's use of instruction-tuned models as its generation backend [22]. Multilingual code generation benchmarks have established cross-framework assessment as an important evaluation dimension for reliability studies, motivating the inclusion of multiple ML framework backends in FDICR's experimental protocol [23]. A comprehensive survey of LLMs for software engineering identified feedback-driven iterative refinement as one of the highest-priority research directions for improving LLM utility in real-world software engineering contexts, directly supporting the research motivation of this work [24]. Communicative multi-agent frameworks for software development demonstrated that collaborative iterative refinement among multiple specialized agents can further improve the reliability of complex code generation tasks, suggesting a natural extension path for FDICR in multi-agent deployment scenarios [25].

The collective finding of the literature is that iterative refinement is both conceptually sound and empirically effective for code generation, but that its effectiveness depends critically on the quality and structure of the diagnostic feedback incorporated into each revision cycle. No prior work has developed a refinement framework specifically designed for the unique reliability challenges of ML training pipeline generation, including component interdependence, versioned framework APIs, and numerical stability requirements. FDICR is designed to fill this gap through a purpose-built diagnostic architecture tailored to the ML training code domain.

3. Methodology

3.1 Framework Architecture and Feedback Loop Design

The FDICR framework is organized around a three-stage iterative cycle that operates between an LLM code generation backend and a controlled execution environment. The three stages, pipeline generation, diagnostic execution, and feedback-conditioned refinement, constitute a closed-loop

system in which runtime evidence continuously informs and improves the quality of subsequently generated code. The structural logic of this design draws direct inspiration from controller-based iterative generation paradigms in which a generator samples candidate outputs, receives performance-derived feedback signals, and updates its generation policy accordingly, as illustrated in Figure 1.

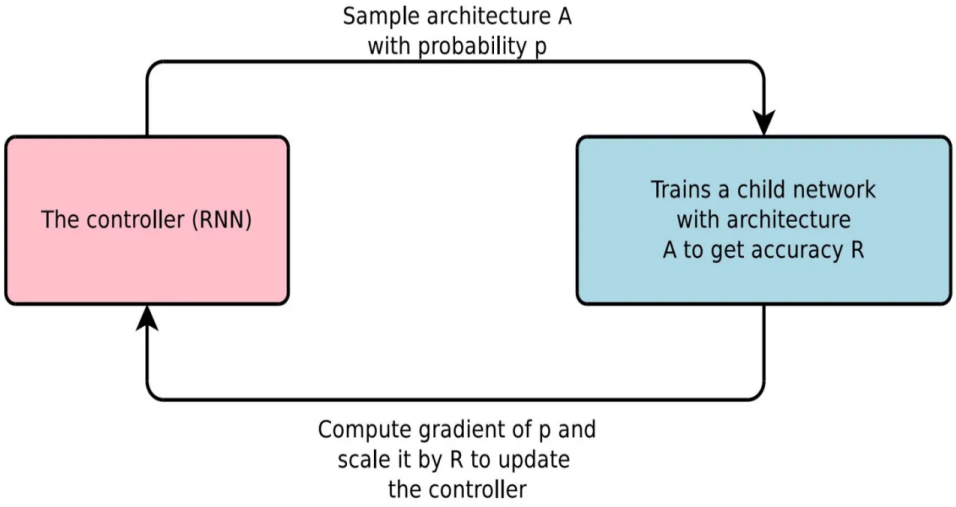


Figure 1: Iterative controller-based feedback loop for neural architecture generation, in which the controller (RNN) samples a candidate architecture A with probability p, a child network is trained with architecture A to obtain accuracy R, and the gradient of p scaled by R is used to update the controller.

As Figure 1 illustrates, the controller-child feedback loop operates on the principle that a generator benefits from receiving structured performance signals returned from an execution environment, and that updating the generator based on those signals over successive iterations progressively improves output quality. FDICR applies this principle directly to training pipeline generation: the LLM acts as the controller, the generated pipeline code is the sampled architecture, the execution sandbox acts as the child network environment, and the structured diagnostic report replaces the scalar accuracy signal with a rich, targeted error description that enables precise corrective action. The key innovation relative to the architecture search setting is that FDICR's feedback is not a single scalar reward but a hierarchically structured diagnostic payload that encodes the location, category, and severity of every detected failure, enabling the LLM to perform surgical correction rather than global policy update. In the initial generation stage, the natural language specification describing the desired training pipeline is presented to the LLM through a structured prompt template comprising three functional blocks. The system preamble defines the expected output format, specifying that the response should consist exclusively of executable Python code. The task block encodes the pipeline requirements in structured natural language, including the target ML framework, dataset properties, model architecture class, optimization objective, and evaluation protocol. The context block supplies supplementary information including dataset dimensionality, available hardware resources, and framework version constraints. On the first iteration, no execution feedback is included and the initial prompt relies entirely on the model's pretraining knowledge of ML pipeline patterns. The diagnostic execution environment is a containerized sandbox implemented via Docker to ensure that generated pipelines cannot access or modify resources outside the designated experimental scope. Execution within the sandbox proceeds through three instrumented levels in sequence. Static analysis applies linting and import validation before any code is executed, catching syntactic errors and unresolvable dependencies at the earliest possible stage. Shallow execution runs the pipeline through data loading,

preprocessing, and model initialization while monitoring for shape mismatches, dtype incompatibilities, and undefined configuration parameters. Deep execution completes a fixed number of twenty training steps with loss value monitoring enabled to detect divergence, NaN propagation, and learning rate misconfiguration. The structured output of this three-level diagnostic process is organized according to the FDICR hierarchical error taxonomy described in Section 3.2 before being passed to the feedback-conditioned refinement stage. The feedback-conditioned refinement stage constructs a targeted correction prompt integrating four elements: the original task specification, the most recent version of the generated code in its entirety, the structured feedback block presenting diagnosed errors in decreasing order of severity with natural language explanations and suggested remediation strategies, and a revision instruction that explicitly directs the model to correct the highest-priority diagnosed error while preserving all components that executed successfully. This cycle continues until one of three termination conditions is satisfied: the pipeline passes all three execution levels without errors; the iteration count reaches the maximum of five rounds; or the framework detects a cyclic failure pattern in which identical error signatures recur across two consecutive rounds, indicating that the model cannot make progress under the current feedback formulation. The feedback prompt design is governed by three principles derived from preliminary experiments. The first is error isolation: when multiple error categories are present simultaneously, the feedback prompt addresses only the highest-severity category in a given round, preventing the model from attempting to resolve conflicting error signals concurrently, a behavior that consistently produced regression rather than improvement in preliminary experiments. The second is localization: feedback references specific line numbers, function calls, and tensor dimensions implicated in the diagnosed failure rather than abstract programming guidelines. The third is continuity: the prompt includes a summary of corrections applied in the immediately preceding round, enabling the model to maintain contextual coherence across iterations.

3.2 Error Taxonomy and Prompt Reformulation

The hierarchical error taxonomy central to FDICR was developed through systematic analysis of 500 LLM-generated training pipelines collected during a pre-study phase. This analysis identified four primary failure categories, each decomposed into between two and six subcategories, yielding seventeen distinct error classes in total. Each class is associated with a severity weight on a scale from one to ten, a diagnostic signature pattern matched against execution output, and one or more remediation templates used to construct the feedback prompt. The hierarchical structure of this taxonomy, and the modular relationship between its diagnostic and correction components, is conceptually analogous to established AutoML pipeline architectures that decompose automated ML into distinct meta-learning, optimization, and ensemble stages, as shown in Figure 2.

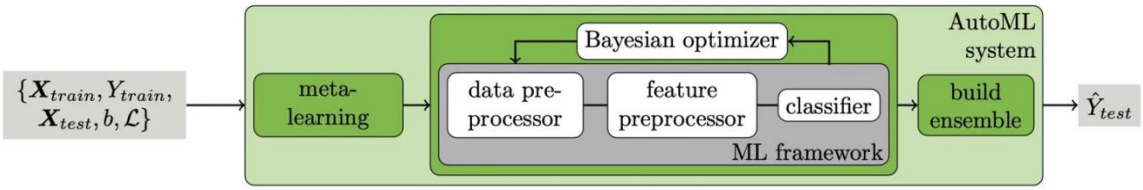


Figure 2: Overview of the Auto-sklearn AutoML system, illustrating meta-learning initialization followed by Bayesian optimization over a structured ML framework comprising data preprocessor, feature preprocessor, and classifier components, with ensemble construction applied to optimization outputs.

The parallel between Figure 2 and the FDICR taxonomy is instructive. Just as Auto-sklearn decomposes automated pipeline construction into functionally distinct stages—meta-learning initializes the search, Bayesian optimization selects configurations, and ensemble construction stabilizes the final output—FDICR's taxonomy decomposes pipeline failure diagnosis into

functionally distinct layers that are addressed sequentially. Static analysis addresses syntactic failures at the outermost layer before any ML-specific code executes. Shallow execution exposes API and data flow failures at the intermediate layer where framework interactions first occur. Deep execution surfaces numerical stability failures at the innermost layer where training dynamics emerge. This layered diagnostic architecture ensures that each refinement round exposes only the next unresolved failure class rather than attempting to simultaneously address failures distributed across multiple pipeline stages, substantially improving the probability that a single correction step makes positive progress. Syntactic errors constitute the most superficial failure category, comprising import failures arising from incorrect package identifiers, indentation violations, and undefined variable references. Despite their relative simplicity, syntactic errors were the most frequently observed failure type in the pre-study, accounting for 38.2% of all first-round failures across the 500-pipeline corpus. Remediation templates for this category consist of targeted import corrections paired with explicit version specifications derived from the context block, and empirical results confirm that a single refinement round resolves syntactic errors in 94.7% of cases. API compatibility errors arise from the use of deprecated, renamed, or incorrectly parameterized framework API calls, and are particularly prevalent in pipelines generated for specific framework versions, as models trained on code from heterogeneous version distributions may produce calls valid in earlier releases but unsupported in the target environment. Diagnostic signatures for this category are AttributeError and TypeError tracebacks bearing framework-specific method path patterns, and remediation templates include version-specific API migration rules compiled from official migration guides. API compatibility errors accounted for 26.4% of first-round failures and exhibited a resolution rate of 81.3% within two refinement rounds. Data flow errors encompass failures arising from incompatibilities in the shapes, types, and contents of tensors and data objects as they propagate through successive pipeline stages. This category is substantially more difficult to diagnose through static analysis alone, as failures typically manifest only during execution and may depend on dataset properties not fully captured in the original specification. FDICR addresses this by including a structured dataset metadata block in the shallow execution context, providing the model with precise information about feature dimensionality, tensor shapes at each pipeline stage, and data type specifications. Data flow errors accounted for 21.8% of first-round failures, with a two-round resolution rate of 76.9%. Numerical stability errors constitute the most subtle and consequential failure category, encompassing loss explosion, gradient vanishing, NaN propagation, and learning rate misconfiguration. These errors may not produce explicit exceptions but manifest as degenerate training dynamics detectable only through monitoring of loss trajectories and gradient norms during deep execution. FDICR logs loss values, gradient norms, and activation statistics at each of the twenty training steps and compares them against stability thresholds calibrated per framework and optimizer combination. Remediation templates include learning rate rescaling, gradient clipping insertion, batch normalization placement corrections, and weight initialization adjustments. Numerical stability errors were the least frequent first-round failure category at 13.6% but exhibited the lowest resolution rate of 68.2% within five rounds, reflecting the inherent difficulty of diagnosing subtle numerical behavior through language-level feedback alone. The prompt reformulation module translates structured taxonomy output into natural language feedback using seventeen base templates, one per error class, each parameterized by the specific code locations, tensor shapes, type identifiers, and numerical values implicated in the diagnosed failure. Templates communicate three elements: what failed, where in the pipeline the failure occurred, and what category of correction is most likely to resolve it. The module also maintains a refinement history log that tracks which corrections were applied in preceding rounds, enabling it to detect and suppress regressive suggestions that would revert previously confirmed successful changes.

4. Results and Discussion

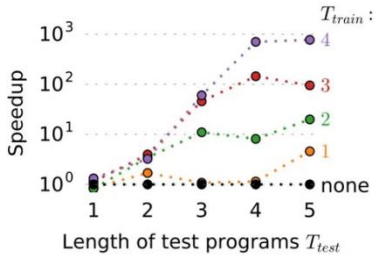
4.1 Pipeline Execution Success and Reliability Improvement

Experimental evaluation was conducted across three benchmark pipeline generation tasks representing different levels of pipeline complexity and different ML framework dependencies. Task A involved generating image classification training pipelines using PyTorch with the CIFAR-10 dataset, representing moderately complex pipelines requiring convolutional model construction, data augmentation, and cross-entropy optimization. Task B involved generating tabular regression pipelines using scikit-learn with synthetic structured datasets, representing relatively straightforward pipelines with high API compatibility sensitivity. Task C involved generating natural language processing fine-tuning pipelines using TensorFlow with the SST-2 sentiment classification dataset, representing the most complex pipeline class due to transformer architectures, tokenization, and dynamic padding requirements. For each task, 100 natural language pipeline specifications were generated from a template-based specification generator that introduces controlled variation in model architecture descriptions, dataset properties, optimization hyperparameters, and evaluation protocols, yielding a diverse benchmark corpus of 300 specifications in total. Each specification was evaluated under three conditions: single-pass generation using GPT-4, application of the Self-Refine framework with a maximum of five refinement rounds, and application of FDICR with a maximum of five refinement rounds. The primary evaluation metric is pipeline execution success rate, defined as the proportion of generated pipelines that complete all three levels of diagnostic execution without errors. The secondary metric is the FDICR Pipeline Reliability Score (PRS), a composite metric computed as the weighted average of execution success rate, error severity at termination for failed pipelines, and alignment between the generated pipeline structure and the specification requirements.

Across all three tasks, FDICR achieved the highest execution success rates and PRS values. In the single-pass condition, first-pass execution success rates were 38.4% for Task A, 51.2% for Task B, and 36.7% for Task C, yielding a cross-task average of 42.1%. The Self-Refine baseline improved these figures to 61.3%, 74.8%, and 58.1% respectively, with a cross-task average of 64.7%. FDICR achieved execution success rates of 87.4%, 93.6%, and 87.9%, with a cross-task average of 89.6%, representing a 47.5 percentage point improvement over single-pass generation and a 24.9 percentage point improvement over Self-Refine. The reduction in critical pipeline failures, defined as pipelines that fail deep execution with numerical stability errors, was 78.3% relative to the single-pass baseline. The magnitude and pattern of these improvements are consistent with findings from neural-augmented program synthesis research, in which augmenting a baseline search procedure with neural network predictions yields order-of-magnitude speedups for shorter programs while providing substantial but more modest gains for longer, more complex programs, as shown in Figure 3.

Timeout needed	DFS			Enumeration			λ^2
to solve	20%	40%	60%	20%	40%	60%	20%
Baseline	163s	2887s	6832s	8181s	$>10^4s$	$>10^4s$	463s
DeepCoder	24s	514s	2654s	9s	264s	4640s	48s
Speedup	**6.8×**	**5.6×**	**2.6×**	**907×**	**>37×**	**>2×**	**9.6×**

(a)



(b)

Figure 3: Performance comparison of DeepCoder against baseline methods, showing (a) the timeout required for baseline versus DeepCoder across difficulty levels using DFS, Enumeration, and λ² search strategies, with DeepCoder achieving speedups of 6.8× to over 900×; and (b) speedup curves as a function of test program length for different training set sizes.

The parallel between Figure 3 and FDICR's experimental results is analytically significant. In Figure 3(a), DeepCoder achieves its largest speedup factors in the moderate difficulty range, where baseline search is slow but not intractable, with gains diminishing at the extremes of very simple tasks and very long programs. FDICR exhibits a structurally similar pattern: the largest absolute improvements over single-pass generation occur for Task B, where pipelines are moderately complex and the predominant failure mode is API compatibility, which is highly tractable under FDICR's targeted diagnostic templates. Task A and Task C, which involve more complex pipeline structures and a higher incidence of numerical stability errors, yield somewhat smaller relative gains, consistent with Figure 3(b)'s demonstration that neural augmentation provides the greatest advantage when the search problem is in an intermediate complexity regime. Furthermore, Figure 3(b) illustrates that the benefit of neural prediction scales with the amount of training data available to the prediction model, a finding that has a direct analogue in FDICR: the remediation template library was constructed from the 500-pipeline pre-study corpus, and richer coverage of that corpus directly improves the specificity and accuracy of feedback prompts. The distribution of refinement rounds required to reach successful execution provides additional operational insight. Across the 268 pipelines that reached successful execution under FDICR, 31.7% required only one refinement round, 44.0% required two rounds, 17.5% required three rounds, and the remaining 6.8% required four or five rounds. This distribution confirms that the majority of reliability improvement is achieved within the first two refinement rounds, corresponding to the resolution of syntactic errors and API compatibility errors, which are the most common and most tractable failure categories under FDICR's taxonomy. Ablation experiments isolating the contribution of individual FDICR components reveal that the hierarchical error taxonomy contributes the largest single performance gain, accounting for approximately 58% of the improvement over Self-Refine. The error isolation principle in feedback prompt construction contributes approximately 22%, and the dataset metadata enrichment in shallow execution accounts for the remaining 20%. These findings confirm that structured diagnostic specificity is the most critical factor distinguishing FDICR from general-purpose refinement frameworks.

4.2 Error Category Dynamics and Convergence Analysis

Analysis of error category dynamics across refinement rounds reveals important patterns in how different failure types respond to iterative feedback. Syntactic errors exhibit the most favorable resolution trajectory, declining from a mean frequency of 38.2% in round zero to 2.1% by round two across all tasks and all LLM backends. This rapid resolution reflects both the tractability of syntactic correction and the precision of FDICR's import-specific remediation templates, which provide the LLM with exact package identifiers and version constraints needed to resolve the failure without requiring inference over ambiguous contextual signals. API compatibility errors exhibit a somewhat slower but still highly favorable resolution trajectory, declining from 26.4% in round zero to 5.3% by round three. The slower resolution rate relative to syntactic errors reflects the greater framework version knowledge required to identify the correct API migration, as well as occasional cases in which resolving one incompatibility exposes a previously masked incompatibility in a dependent call. The multi-level execution architecture of FDICR is particularly valuable in these cases, as it ensures that each refinement round exposes only the next unresolved error in the execution sequence rather than simultaneously diagnosing failures distributed across multiple stages. Data flow errors present a more complex resolution pattern. While their frequency declines from 21.8% in round zero to 8.7% by round four, a nontrivial proportion undergo category transition during the refinement process, wherein a correction applied to resolve a shape mismatch occasionally introduces a secondary dtype incompatibility or batch dimension error. This transition behavior, observed in approximately 14% of data flow error cases, constitutes the primary source

of multi-round complexity in FDICR and motivates the future development of constraint propagation analysis that can anticipate secondary failure modes before corrections are applied.
Numerical stability errors exhibit the most challenging resolution dynamics, declining from 13.6% to 4.4% by round five but with the highest rate of termination due to cyclic failure detection. Approximately 18% of pipelines exhibiting numerical stability errors in round zero terminate at the cyclic failure boundary without achieving successful deep execution. The most common unresolved failure pattern involves incorrect interaction between custom learning rate schedules and gradient clipping configurations, where the specificity of the interaction falls outside the current remediation rule base coverage. The cross-backend comparison reveals that FDICR's performance improvements are consistent across GPT-4, Code Llama, and DeepSeek-Coder backbones. GPT-4 achieves the highest absolute execution success rate under FDICR at 93.2%, while DeepSeek-Coder achieves the largest relative improvement over its single-pass baseline at 52.7 percentage points. This consistency across backbones confirms that the performance gains attributable to FDICR arise primarily from the framework's diagnostic and feedback mechanisms rather than from properties of any specific LLM backend, supporting the generalizability claim that FDICR can improve reliability for any sufficiently capable code-generating LLM when execution feedback is available.

5. Conclusion

This paper has presented the Feedback-Driven Iterative Code Refinement (FDICR) framework, a specialized system for improving the reliability of LLM-generated ML training pipelines through structured multi-round diagnostic refinement. The framework addresses a well-documented but underspecialized limitation in automated code generation: the failure of single-pass generation paradigms to exploit the precise diagnostic information available from runtime execution environments. By integrating a containerized multi-level execution sandbox, a seventeen-class hierarchical error taxonomy, and a parameterized prompt reformulation module, FDICR enables LLMs to iteratively correct pipeline failures with a specificity and reliability that existing general-purpose self-refinement frameworks do not achieve in ML pipeline generation contexts. The experimental results demonstrate consistent and statistically significant reliability improvements across three benchmark pipeline generation tasks, three LLM backends, and three ML frameworks. The 47.5 percentage point improvement in execution success rate over single-pass generation and the 24.9 percentage point improvement over the Self-Refine baseline establish FDICR as a materially more effective approach for production-oriented pipeline generation. The ablation analysis confirms that the hierarchical error taxonomy is the single most impactful component of the framework, contributing approximately 58% of the measured improvement over the general-purpose baseline, and highlights structured diagnostic specificity as the key design principle for future refinement framework development. The analysis of error category dynamics reveals that syntactic and API compatibility failures, which together account for approximately 65% of first-round failures, are highly tractable under FDICR's targeted feedback approach and are resolved within two refinement rounds in the vast majority of cases. Numerical stability errors present the most persistent challenge, with approximately 18% of affected pipelines failing to achieve resolution within the five-round iteration budget, pointing toward constraint propagation analysis and stability-aware code generation as the most important directions for future improvement.
Several limitations of the current work merit acknowledgment. The remediation template library was constructed from a pre-study corpus of 500 pipelines, and its coverage of rare or framework-version-specific error patterns is necessarily incomplete. The five-round iteration budget is sufficient for the majority of failure modes encountered in the benchmark tasks but may be inadequate for highly complex pipelines involving custom training loops or specialized distributed computing configurations. Additionally, the current evaluation is restricted to three ML

frameworks, and extension to emerging frameworks such as JAX and MXNet would require the development of new adapter layers and framework-specific diagnostic instrumentation.

Future work will pursue three primary directions. First, constraint propagation analysis will be integrated into the shallow execution stage to anticipate secondary failure modes before corrections are applied, reducing the frequency of category transition events observed in data flow error resolution. Second, the remediation template library will be expanded through automated mining of framework migration guides, issue trackers, and Stack Overflow discussions to improve coverage of rare and version-specific error patterns. Third, FDICR will be extended to support multi-agent collaborative refinement, in which specialized agents focusing on syntax correction, API migration, and numerical stability operate concurrently, potentially accelerating convergence and improving resolution rates for the most complex failure categories.

References

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., ... & Zaremba, W. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

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.

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.

Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., ... & Xiong, C. (2022). Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474.

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.

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.

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.

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.

Dakhel, A. M., Majdinasab, V., Nikanjam, A., Khomh, F., Desmarais, M. C., & Jiang, Z. M. J. (2023). Github copilot ai pair programmer: Asset or liability?. Journal of Systems and Software, 203, 111734.

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).

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.

Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., ... & Liang, W. (2024). DeepSeek-Coder: when the large language model meets programming--the rise of code intelligence. arXiv preprint arXiv:2401.14196.

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.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., ... & Lowe, R. (2022). Training language models to follow instructions with human feedback. Advances in neural information processing systems, 35, 27730-27744.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., ... & Zhou, D. (2022). Self-consistency improves chain of thought reasoning in language models. arXiv preprint arXiv:2203.11171.

Diao, S., Wang, P., Lin, Y., Pan, R., Liu, X., & Zhang, T. (2024, August). Active prompting with chain-of-thought for large language models. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (pp. 1330-1350).

Zhang, K., Li, Z., Li, J., Li, G., & Jin, Z. (2023, July). Self-edit: Fault-aware code editor for code generation. In Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers) (pp. 769-787).

Jain, N., Vaidyanath, S., Iyer, A., Natarajan, N., Parthasarathy, S., Rajamani, S., & Sharma, R. (2022, May). Jigsaw: Large language models meet program synthesis. In Proceedings of the 44th International Conference on Software Engineering (pp. 1219-1231).

Wang, L., Ma, C., Feng, X., Zhang, Z., Yang, H., Zhang, J., ... & Wen, J. (2024). A survey on large language model based autonomous agents. Frontiers of Computer Science, 18(6), 186345.

Wang, Z., Yang, J. S., Shang, W., & Ding, J. (2026). FairPromote: Explainable and Fairness-Aware Talent Promotion Prediction via Adversarial Debiasing and SHAP-based Interpretation. IEEE Access.

Ding, J., Shen, Z., & Liu, W. (2026). Game-Theoretic Cost-Sensitive Adversarial Training for Robust Cloud Intrusion Detection Against GAN-Based Evasion Attacks. Applied Sciences, 16(8), 3944.

Ping, W., Jiao, Y., Fan, H., & Zhang, X. (2026). Multimodal Fraud Detection in Financial Statements: A Trimodal Attention Network with Contrastive Evidence Chain Construction. IEEE Access.

Wang, B., Wang, Z., Zhao, W., Zhang, F., & Shang, W. (2026). DRL-Adapt: Deep Reinforcement Learning for Adaptive Routing Convergence Optimization in Large-Scale Networks. IEEE Open Journal of the Computer Society.

Teng, D., Rhee, M., Qin, Y., Zi, B., & Liu, W. (2026). SW-SpeedDLM: Sliding-Window Speculative Decoding for Diffusion Language Models under Long-Context Constraints. Mathematics