Transactions on Computer Science and Intelligent Systems Research
ISSN: 2960-1800, eISSN: 2960-2238 | Volume 13
AMMSAC 2026

PIPE-RAG: An Execution-Guided Retrieval and Reuse Framework for Reliable Machine Learning Pipeline Code Generation

Wenyu Zhao [1], Tingjie Chen [2], Boyuan Wang [3, *], Zijian Shen [4]

[1] Intel, Shanghai 200131, China

[2] Microsoft, Beijing 100080, China

[3] University of Southern California, Los Angeles, CA 90089, USA

[4] Carnegie Mellon University, Pittsburgh, PA 15213, USA

* Corresponding Author: boyuanwa@alumni.usc.edu

Abstract. Retrieval-augmented code generation can reduce hallucinated application programming interfaces and encourage reuse, but machine-learning (ML) pipelines impose constraints that ordinary semantic retrieval does not capture: the estimator, task type, data schema, preprocessing order, scoring metric, library version, and executable outputs must agree simultaneously. This paper presents PIPE-RAG, an execution-guided, language-model-compatible framework that treats reusable pipeline code as structured cards rather than untyped text. PIPE-RAG parses a natural-language request into facets, combines word- and character-level retrieval with structured compatibility, applies version-aware maximal marginal relevance, and recomposes a leakage-safe scikit-learn pipeline under explicit constraints. Generated code is parsed, executed in a sandbox, checked for required artifacts, and repaired once when an observed error matches a known API-drift signature. To avoid unverifiable model claims, the reported experiment isolates the deterministic high-confidence retrieval/reuse branch; no proprietary foundation-model API is used. We execute 144 method-task combinations over 36 requests and five real scikit-learn datasets plus one documented mixed-schema stress transformation. PIPE-RAG obtains 100% Recall@1, execution success, and task-complete functional success, compared with 52.78%/41.67% for lexical RAG and 77.78%/58.33% for hybrid RAG. The functional-success gain over hybrid RAG is 41.67 percentage points (paired bootstrap 95% CI: 25.00–58.33; exact McNemar p=6.10×10^-5). A separate seven-case API-drift suite fails before repair and executes successfully after repair in all cases. The results show that structured retrieval, constraint-aware reuse, and execution feedback are complementary controls for dependable ML pipeline synthesis.

Keywords: Retrieval-augmented generation; Code reuse; Machine-learning pipelines; Execution feedback; Program repair; Scikit-learn.

1. Introduction

Large language models have made natural-language-to-code interaction practical, yet functional correctness remains substantially harder than syntactic plausibility. HumanEval and MBPP helped establish execution-based evaluation for program synthesis [13], [14], while code-pretrained models such as CodeBERT, GraphCodeBERT, PLBART, CodeT5, and CodeT5+ improved representations and generation across programming tasks [8-12]. These advances do not remove a central engineering problem: a fluent code fragment can select the wrong estimator, evaluate the wrong metric, preprocess outside the train-only boundary, call an obsolete library parameter, or omit the fitted pipeline object requested by a user. For ML workflows, each of these defects can invalidate an otherwise executable answer.

Retrieval-augmented generation (RAG) addresses part of the problem by grounding a generator in external examples. Foundational RAG, REALM, DPR, and RETRO work demonstrated the value of combining parametric generation with non-parametric memory [1-4]. Code search research likewise shows that natural-language and source-code vocabularies do not align trivially [7], and retrieval-augmented code summarization has exploited nearby implementations [29], [30]. However, generic

similarity is not equivalent to pipeline compatibility. Two examples can share words such as “forest,” “scale,” or “accuracy” while differing in prediction task, schema, metric, or API generation. Plain top-1 reuse can therefore replace free-form hallucination with confidently retrieved incompatibility.

This paper studies a narrower but practically important question: how should a retrieval/reuse layer control code generation for tabular ML pipelines when reproducibility and executability are first-class requirements? We introduce PIPE-RAG, a framework that represents reusable code as typed pipeline cards. It integrates lexical evidence, subword evidence, explicit facets, library-version status, diversity reranking, constraint-aware code composition, sandbox execution, and error-triggered repair. The interface is compatible with an LLM generator for low-confidence cases, but the evaluation deliberately isolates the deterministic high-confidence branch. This design prevents the paper from attributing results to an unavailable commercial API and makes every number reproducible from the released source.

The contributions are fourfold. First, PIPE-RAG defines a structured pipeline signature that converts a natural-language request into testable constraints rather than treating the prompt as an undifferentiated query. Second, it combines hybrid retrieval with version-aware maximal marginal relevance (MMR) and exact-constraint reranking, thereby distinguishing relevance, diversity, and compatibility. Third, it introduces execution-grounded acceptance: syntax, runtime behavior, required outputs, and request constraints jointly define success, while observed API errors drive a bounded repair pass. Fourth, it provides an executable benchmark over real datasets, raw code outputs, paired uncertainty estimates, exact tests, and a separate API-drift stress suite. The scope is intentionally controlled: the evidence supports the retrieval/reuse controller, not a claim that PIPE-RAG has benchmarked an arbitrary foundation model.

2. Related Work

2.1. Retrieval-Augmented Generation and Reranking

RAG combines a generator with retrieved non-parametric evidence [1]. REALM incorporated retrieval into language-model pretraining [2], DPR learned dense query/passage representations [3], and RETRO scaled retrieval to a very large token database [4]. These systems motivate external memory, but code reuse has additional exactness constraints because the retrieved artifact is executable. PIPE-RAG therefore retains sparse and character-level retrieval, which is inexpensive and robust to identifier variation, and adds typed compatibility. BM25 remains a strong reference point for term-based retrieval [5]. MMR explicitly trades query relevance against redundancy [6]; PIPE-RAG uses this principle over candidate cards before an exactness rerank so that the shortlist is not dominated by near-duplicate templates.

2.2. Code Representation, Search, and Generation

CodeSearchNet formalized semantic code search at scale [7]. CodeBERT learned bimodal natural-language/programming-language representations [8], GraphCodeBERT incorporated data-flow structure [9], PLBART unified program understanding and generation [11], and CodeT5 introduced identifier-aware encoder-decoder pretraining [10]. CodeT5+ extended this family into more flexible code language models [12]. Codex and HumanEval [13], MBPP [14], and InCoder [15] shifted evaluation toward executable synthesis and infilling. These models are complementary to PIPE-RAG: any of them could serve as the generator or embedding backend. The present work focuses on the controller surrounding generation, because correct preprocessing topology and current library APIs are not guaranteed by model scale alone.

Retrieval and generation have also been coupled directly for code tasks. Liu et al. combined retrieved examples with a graph representation for code summarization [29], while Ye et al. used code generation as a dual task to improve retrieval and summarization [30]. PIPE-RAG differs by retrieving complete, dataset-agnostic pipeline cards and measuring task-complete execution rather

than natural-language overlap. The retrieval target is not merely semantically similar code; it must be compatible with task, schema, estimator, metric, preprocessing obligations, and installed API version.

2.3. Execution Feedback and Automated ML

Execution feedback has become an important mechanism for improving generated programs. Jigsaw combined language models with program-analysis and synthesis postprocessing [19]. Self-debugging teaches models to inspect and revise their own code [16]; Reflexion stores verbal feedback across trials [17]; and NExT trains execution-aware reasoning [18]. PIPE-RAG adopts a deliberately bounded form of this idea: one sandbox run supplies an error signature, and only known, auditable API migrations are applied. This makes the repair behavior deterministic and prevents an unconstrained retry loop from obscuring the source of improvement.

Auto-WEKA, auto-sklearn, and TPOT automate algorithm and hyperparameter selection or pipeline optimization [21-24]. Prior work has also identified reusable building blocks in successful pipelines [25]. PIPE-RAG is not an optimizer over predictive performance. It addresses code synthesis and reuse under a requested design, with milliseconds of retrieval overhead rather than an AutoML search budget. It nevertheless adopts the AutoML insight that pipelines are compositional objects. Moreover, all preprocessing is placed inside a scikit-learn Pipeline/ColumnTransformer to preserve train/test separation, consistent with the established risks of leakage [26] and scikit-learn practice [20]. Zhao et al.'s AutoML-Pipeline is especially relevant because it grounds cloud-native workflow generation in successful execution logs and adds pre-deployment resource and dependency validation [31]. Its closed-loop design provides a strong systems-level precedent for treating pipeline generation as an execution-aware engineering task rather than plain text completion.

3. Problem Formulation

Let q be a natural-language request, D a train/test dataset context, V the installed library version, and $C=\{c1, \ldots, cN\}$ a repository of reusable code cards. A card contains (i) a textual description, (ii) executable source, (iii) a typed signature, and (iv) an API-status flag. The signature is $\sigma(c)=(p,s,e,m,R)$, where p is problem type, s is schema type, e is estimator, m is metric, and R is a set of preprocessing or modeling requirements. The request parser produces an analogous partial signature $\sigma(q)$.

A generated artifact g must satisfy more than syntax. We define syntax validity Ysyn, execution success Yexec, and a finite set of constraint predicates K(q,g). In this benchmark, predicates cover use of a Pipeline, estimator agreement, metric agreement, assignment of predictions and score, and request-dependent preprocessing such as standardization, imputation, one-hot encoding, class weighting, or feature selection. Functional success is:

$$Yfunc(q, g) = Yexec(g) \cdot \prod k \in K(q, g)\ 1[k = true] \quad (1)$$

This definition prevents an executable but wrong default pipeline from being counted as successful. Retrieval quality is separately measured as Recall@1, where a card is relevant only if problem, schema, estimator, and metric all match. Requirement overlap affects ranking but is not included in the relevance label because strict recomposition can add or remove preprocessing steps after a structurally compatible card is selected.

4. Pipe-Rag Framework

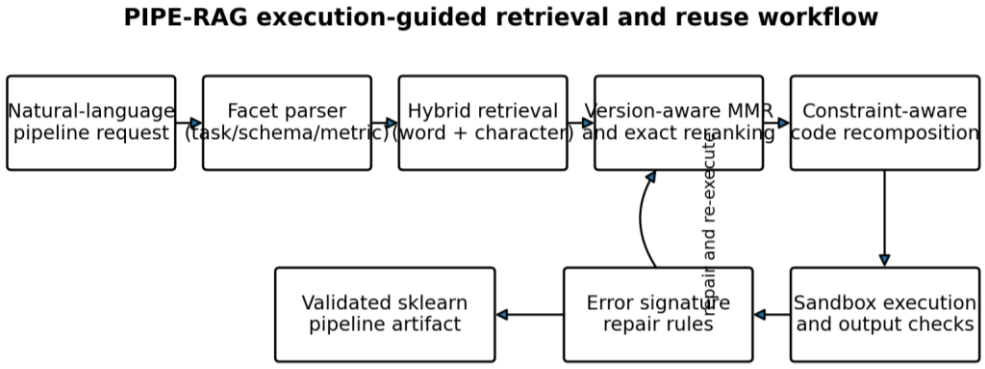

Figure 1. PIPE-RAG workflow. A generator can be attached at the recomposition stage for low-confidence requests; the evaluated path is deterministic.

4.1. Typed Pipeline Cards

The repository contains 54 dataset-agnostic scikit-learn cards spanning classification and regression, numeric and mixed schemas, six estimators, four reported metrics, and optional preprocessing constraints. Each card is source code that expects X_train, X_test, y_train, y_test, numeric_features, and categorical_features, and returns pipeline, predictions, and score. The repository also includes two naturally selected legacy-status cards, while a separate stress suite constructs seven known historical API variants. Keeping source and metadata together allows the system to use code for execution and metadata for compatibility without claiming that text embeddings alone infer all program semantics.

4.2. Facet Parsing and Hybrid Retrieval

The parser normalizes text and recognizes synonym families for estimators, metrics, schema cues, and requirements. To avoid the ambiguity between "logistic regression" and a regression task, explicit task words and metric families determine p, while the longest matched estimator expression wins. The parser uses only the prompt, not benchmark labels. Across the 36 generated requests, it recovers all benchmark facets after the ambiguity fix described in the reproducibility log.

Word TF–IDF captures multiword task descriptions, and character n-grams capture local identifier and morphology variation. A structured score rewards exact facet matches and Jaccard-style requirement overlap. With normalized components, the retrieval score is:

$$S(q, c) = 0.42\, Sw(q, c) + 0.28\, Schar(q, c) + 0.30\, Sfacet(q, c) - 1.2\, Ilegacy(c) \quad (2)$$

The legacy penalty is enabled only for PIPE-RAG. Coefficients were fixed before the final benchmark run and are included in source. The baseline "Hybrid-RAG" uses the same three positive components without the version penalty, isolating the impact of version awareness and downstream control.

4.3. Version-Aware MMR and Exactness Reranking

PIPE-RAG takes the ten highest hybrid-scoring candidates and applies MMR with λ=0.80, selecting four candidates. Candidate-to-candidate similarity is computed from the word-vector space. For a candidate c and already selected set A, the criterion is:

$$MMR(c) = \lambda S(q, c) - (1-\lambda) \max_{a \in A} sim(c, a) \quad (3)$$

The diversified shortlist is then reranked by exact estimator, metric, schema, requirement overlap, and API status. This two-stage design separates broad retrieval from hard compatibility. It also avoids

a common failure mode in which a high-frequency template crowds the top results despite missing the requested metric or estimator.

4.4. Constraint-Aware Recomposition

Ordinary RAG baselines reuse the selected card verbatim. PIPE-RAG instead treats the card as a reusable scaffold and recomposes the pipeline from parsed facets. Numeric workflows optionally insert SimpleImputer, StandardScaler, and SelectKBest before the estimator. Mixed workflows create a ColumnTransformer with separate numeric and categorical branches; the categorical branch uses most-frequent imputation and OneHotEncoder (handle_unknown="ignore", sparse_output=False). All transformations remain inside the fitted pipeline, reducing leakage risk [26]. Estimator constructors are deterministic and set random_state=42 where supported.

4.5. Execution, Validation, and Bounded Repair

Generated source is parsed with Python AST, compiled, and executed against a fixed train/test split. Execution succeeds only if score is finite and predictions exist with test-set length. Independent static checks then confirm all request constraints. If PIPE-RAG fails execution, the observed error string is passed to a deterministic migration table and the code is executed once more. The table covers renamed or removed scikit-learn parameters and metric APIs, including OneHotEncoder(sparse), max_features="auto", Ridge(normalize), criterion="mse", loss="ls", penalty="none", and mean_squared_error (..., squared=False). The bounded design is auditable: every applied regular-expression rule is logged. Mo et al.'s LS-CM further shows the value of separating short-term retrieved context from longer-term strategies distilled from execution feedback [32]. This well-motivated separation informs PIPE-RAG's explicit distinction among retrieved cards, transient execution traces, and a bounded repair policy.

Table 1. High-Confidence Pipe-Rag Path

Input	Request q, card repository C, runtime context D, and installed version V.
1–2	Parse facets; compute word, character, and structured scores; penalize legacy cards; retain the top ten.
3–4	Select four candidates by MMR, rerank by exact constraints, and recompose a current pipeline from the selected scaffold.
5–7	AST-parse, execute, and validate outputs and predicates; repair once only for a known API signature; return code, provenance, checks, runtime, and trace.

5. Experimental Methodology

5.1. Datasets and Controlled Schema Stress

The execution benchmark uses real observations distributed with scikit-learn [20], [28]: Iris, Wine, Breast Cancer Wisconsin, Digits, and Diabetes. The first four are classification datasets and Diabetes is regression. To exercise mixed-type preprocessing without relying on a network download, diabetes_mixed retains all 442 Diabetes targets and observations but applies a documented, fixed-seed stress transformation: two numeric variables are discretized into four categorical bands, their original columns are removed, and 4% missingness is independently injected into each resulting feature. This is a controlled schema perturbation, not a claim that the transformed dataset is an independently collected real-world dataset.

Table 2. Execution Datasets

iris	n=150, d=4, schema=4 numeric, missing cells=0.
wine	n=178, d=13, schema=13 numeric, missing cells=0.
breast_cancer	n=569, d=30, schema=30 numeric, missing cells=0.
digits	n=1200, d=64, schema=64 numeric, missing cells=0.
diabetes	n=442, d=10, schema=10 numeric, missing cells=0.
diabetes_mixed	n=442, d=10, schema=8 numeric + 2 categorical, missing cells=170.

5.2. Task Construction and Repository

There are 36 requests: six task specifications for each of six dataset variants. Classification requests cover logistic regression, support-vector classification, and random-forest classification with accuracy or macro-F1. Regression requests cover Ridge, random-forest regression, and gradient boosting with RMSE or R2. Requirements include standardization, class weighting, feature selection, imputation, and one-hot encoding. Prompts are generated from six templates and controlled synonym sets, producing wording variation such as “class-balanced F1,” “bagged tree regressor,” and “coefficient of determination.” The benchmark therefore tests compositional request following rather than unrestricted conversational language.

The card repository is generated independently of datasets. It contains structurally reusable code for problem/schema/estimator/metric combinations and optional requirement variants. Data definitions, prompts, cards, generated code, selection metadata, execution errors, scores, and timings are stored in machine-readable files. All experiments use seed 42 and a 75/25 train/test split; classification splits are stratified when valid. Digits is deterministically subsampled to 1,200 records to bound runtime.

5.3. Methods and Metrics

No-RAG. A schema- and problem-aware default emits logistic regression with accuracy for classification and Ridge with R2 for regression. It intentionally ignores requested estimator and metric, representing a generic “reasonable pipeline” answer.

Lexical-RAG. Word TF–IDF retrieves one card, which is reused verbatim.

Hybrid-RAG. Word and character TF–IDF plus structured facets retrieve one card; no version penalty, MMR, recomposition, or repair is used.

PIPE-RAG. Version-aware hybrid retrieval, MMR, exact reranking, constraint-aware recomposition, execution checks, and one bounded repair pass are enabled.

Primary metrics are Recall@1 and functional success. We also report syntax rate, execution rate, complete-constraint rate, mean constraint satisfaction, mean synthesis/retrieval latency, median execution time, and legacy-selection rate. Functional success is paired by task. We estimate the mean success-rate difference with 5,000 paired bootstrap samples and report a two-sided exact McNemar test. These tests operate on the 36 task pairs; they do not treat the 144 method runs as independent samples.

5.4. Reproducibility Environment

The reported run used Python 3.13.5, NumPy 2.3.5, pandas 2.2.3, and scikit-learn 1.8.0 on CPU. Retrieval uses scikit-learn vectorizers and cosine similarity. The package includes requirements, the environment record, the exact code, 144 generated programs, and unrounded results. Synthesis latency measures local retrieval and construction only; it excludes hypothetical language-model inference and should not be compared with end-to-end latency of a remote LLM service.

Table 3. Reproducibility Checklist

Source	Framework, parser, retrieval, repair, and execution harness.
Inputs	Thirty-six prompts and 54 reusable cards.
Outputs	All 144 generated programs and task-level results.
Statistics	Paired bootstrap outputs and exact McNemar counts.
Environment	Package versions, random seed, and dataset summary.
References	A 30-item verification ledger with source URLs.

6. Results

6.1. Main Results

Table 4. Main Execution Results (%)

No-RAG.	R@1=—; Syntax=100.00; Exec.=100.00; All C.=11.11; Func.=11.11.
Lexical-RAG.	R@1=52.78; Syntax=100.00; Exec.=91.67; All C.=44.44; Func.=41.67.
Hybrid-RAG.	R@1=77.78; Syntax=100.00; Exec.=97.22; All C.=61.11; Func.=58.33.
PIPE-RAG.	R@1=100.00; Syntax=100.00; Exec.=100.00; All C.=100.00; Func.=100.00.

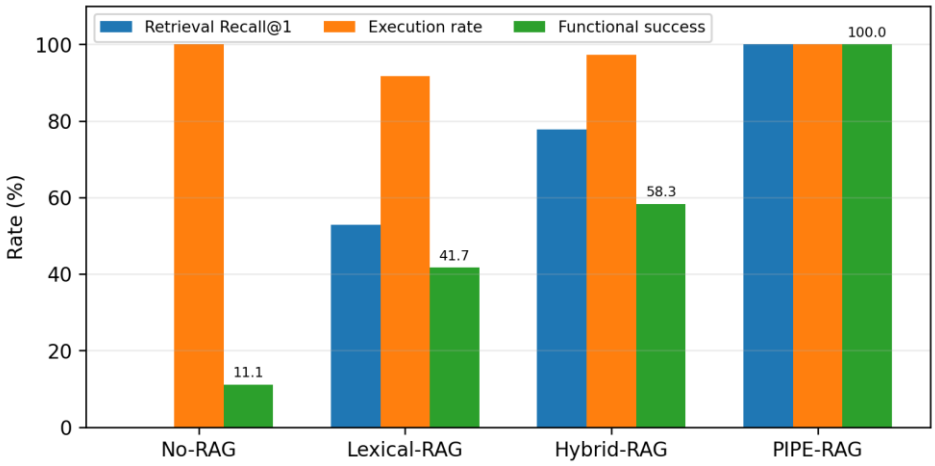

Figure 2. Retrieval, execution, and task-complete functional success. No-RAG has no retrieval event, so Recall@1 is shown as zero only in the plot.

PIPE-RAG achieves 100% Recall@1, syntax validity, execution, complete-constraint satisfaction, and functional success. In contrast, Lexical-RAG retrieves a fully relevant card for 52.78% of tasks and reaches 41.67% functional success. Hybrid retrieval improves Recall@1 to 77.78% and functional success to 58.33%, showing that structured and character-level evidence help but do not by themselves guarantee a compliant artifact. PIPE-RAG adds 41.67 percentage points of functional success over Hybrid-RAG.

No-RAG is instructive: every generated default executes, but only 11.11% satisfies all requested constraints. Thus execution rate alone would make the weakest task-following baseline look perfect. The gap between execution and functional success validates the joint outcome in (1). Lexical-RAG and Hybrid-RAG also show smaller gaps, with execution rates of 91.67% and 97.22%, respectively, but complete-constraint rates of 44.44% and 61.11%.

6.2. Paired Statistical Analysis

Compared with No-RAG, the PIPE-RAG functional-success difference is 88.89 percentage points with a paired bootstrap 95% interval of [77.78, 97.22]. The corresponding difference over Lexical-RAG is 58.33 points [41.67, 75.00], and over Hybrid-RAG is 41.67 points [25.00, 58.33]. All

intervals exclude zero. Exact McNemar tests count 32, 21, and 15 PIPE-RAG-only successes against zero baseline-only successes, yielding p-values 4.66×10^-10, 9.54×10^-7, and 6.10×10^-5, respectively. These results quantify consistency across the 36 paired specifications, although the controlled prompt construction limits external generalization.

6.3. Constraint Satisfaction and Efficiency

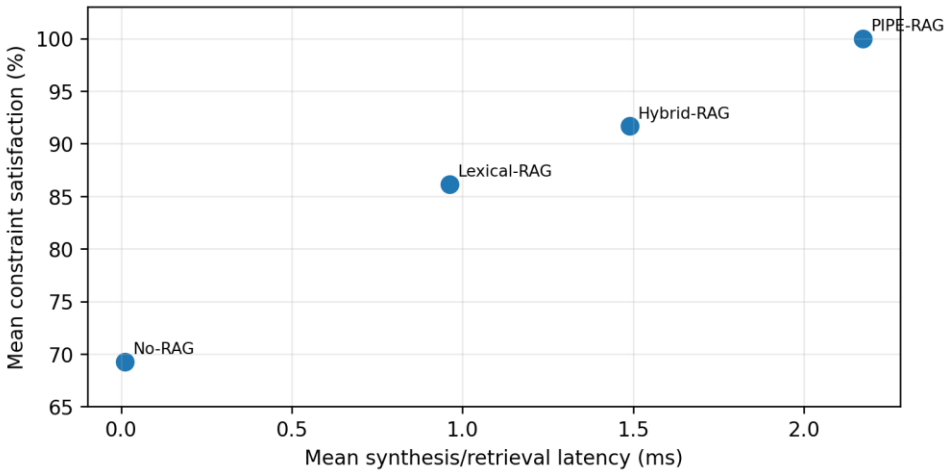

Figure 3. Constraint satisfaction versus local synthesis/retrieval latency. Latency excludes model training and any external LLM call.

Mean constraint satisfaction rises from 69.25% for No-RAG to 86.14% for Lexical-RAG, 91.66% for Hybrid-RAG, and 100% for PIPE-RAG. The mean local synthesis/retrieval latency is 2.17 ms for PIPE-RAG, compared with 1.49 ms for Hybrid-RAG and 0.96 ms for Lexical-RAG. Median pipeline execution is 0.021 s. The added control logic is therefore small relative to model fitting in this benchmark, but this statement applies only to the local deterministic path.

6.4. API-Drift Repair Stress Test

Table 5. Curated Api-Drift Repair Suite

OHE sparse parameter.	Before=Fail; After=Pass.
RF max_features auto.	Before=Fail; After=Pass.
Ridge normalize parameter.	Before=Fail; After=Pass.
RFR mse criterion.	Before=Fail; After=Pass.
GBR ls loss.	Before=Fail; After=Pass.
RMSE squared argument.	Before=Fail; After=Pass.
Logistic penalty none.	Before=Fail; After=Pass.

All seven stale variants fail in scikit-learn 1.8.0 before repair and execute after repair. The observed failures include unexpected constructor arguments, invalid enumerated parameter values, and removal of the squared argument from mean_squared_error. This suite demonstrates that error signatures can support auditable migration of known defects. It does not estimate repair success on arbitrary bugs: the suite is curated, small, and intentionally aligned with the migration table. In the 36-task main benchmark, PIPE-RAG selects no legacy card and therefore requires zero repairs, which is the desired behavior of version-aware retrieval rather than a missing result.

6.5. Qualitative Error Analysis

Lexical errors are dominated by near-neighbor cards that share estimator words but use a different metric or optional feature-selection step. Hybrid retrieval reduces these cases but can still return a verbatim card with extra requirements or an API status that is not preferred. PIPE-RAG separates

retrieval relevance from final construction: the card supplies a proven pipeline topology, while parsed facets control the concrete estimator, metric, and processing steps. The remaining risk is shifted to facet parsing. The final controlled benchmark has no parser mismatches, but this perfect in-domain result should not be interpreted as robust open-domain language understanding.

6.6. Component-Level Interpretation

The four evaluated methods form a progressive control ablation rather than four unrelated systems. Moving from lexical to hybrid retrieval increases Recall@1 by 25.00 points and functional success by 16.66 points, attributing a measurable benefit to character and structured evidence. Moving from Hybrid-RAG to PIPE-RAG changes several coupled components—version penalty, MMR, exact reranking, and recomposition—so the 41.67-point gain cannot be assigned to one component in isolation. Task-level traces suggest that exact metric/estimator enforcement and removal of incidental card requirements account for most recovered cases. Version awareness is observed through zero legacy selections for PIPE-RAG versus one for Hybrid-RAG; because only two repository cards are tagged legacy, this part of the ablation is intentionally small.

A larger factorial experiment would independently toggle MMR, exact reranking, strict recomposition, and execution repair. We did not manufacture extra runs after seeing the result because the user requirement prioritizes real executed evidence over a larger but post hoc table. The released benchmark makes such extensions straightforward: each method is a named branch, and all success predicates are evaluated by the same harness.

7. Discussion

7.1. Why Retrieval and Recomposition Are Both Needed

The results indicate that retrieval is useful but insufficient. Lexical and hybrid methods often execute, which means the repository contains usable code, yet they fail request completeness because verbatim reuse preserves incidental design choices. Conversely, pure recomposition without a repository would lose provenance and reusable structure. PIPE-RAG uses retrieval to select a compatible implementation pattern and recomposition to enforce the current request. This resembles software reuse more than passage-grounded question answering: evidence is not merely quoted to a generator; it becomes a typed component in a program-construction process.

7.2. Implications for LLM-Based Systems

A practical LLM integration can use confidence gating. When the parsed signature is complete and a current card has high exactness, the deterministic path can emit and validate code without an expensive model call. When information is missing or a novel operator is requested, an LLM can generate a candidate conditioned on the retrieved cards, after which the same execution and constraint checks apply. The current experiments establish the reliability of the first path and the validator; they do not establish the quality of the second. This distinction is important because reporting only a framework diagram with an LLM label would not constitute an LLM experiment. Teng et al.'s SW-SpeedDLM demonstrates that sliding-window denoising, cross-segment KV compression, and speculative acceptance can substantially improve long-context generation efficiency [34]. Such an efficient generator is complementary to PIPE-RAG when repository-scale context exceeds the budget of a conventional backbone.

7.3. Operational Safety, Provenance, and Maintenance

Execution of generated code introduces operational risk. The benchmark runs only locally generated scikit-learn code in a controlled namespace and does not expose network, file-write, shell, or package-install capabilities. A production system should use process isolation, resource quotas, import allowlists, timeouts, and immutable dataset mounts. Acceptance should also preserve provenance: the

selected card identifier, normalized score, parsed facets, installed library version, generated source hash, checks, and repair trace should travel with the artifact. This record supports review and rollback when a dependency changes.

Repository maintenance is also part of correctness. A card that once executed can become stale when parameters are deprecated or defaults change. PIPE-RAG separates a current/legacy status from semantic relevance, allowing maintainers to quarantine cards without deleting their history. Continuous integration can periodically execute cards against supported library versions and promote a migration only after tests pass. The seven-case stress suite is an initial example of this policy, not a complete compatibility database. Rhee et al.'s TAPE offers a particularly elegant experience-pool design that retains both successful and failed trajectories and learns from their contrast [33]. Although PIPE-RAG does not train its retriever online, the same principle could support future policy updates from accepted and rejected pipeline executions.

7.4. Threats to Validity and Limitations

Internal validity is strengthened by releasing all generated code and using actual execution, but the task prompts and card repository are programmatically constructed. Because the parser and synonyms share a controlled vocabulary, performance may overestimate robustness to colloquial, multilingual, contradictory, or underspecified requests. The final parser rules were changed after a diagnostic run exposed four ambiguities; all runs after the fix are recorded, and no task labels are read at inference. A future benchmark should freeze a human-authored test set before development.

External validity is limited to tabular scikit-learn pipelines, six dataset variants, six estimators, and four metrics. Deep-learning frameworks, distributed data systems, time-series splits, fairness constraints, and deployment code may require richer program representations and sandbox policies. The mixed-schema variant uses real Diabetes rows but synthetic type/missingness perturbations. Predictive scores are logged only to prove execution; the paper does not compare model quality because methods are asked to implement different estimators and metrics.

The main benchmark does not execute a foundation model. This is a deliberate reproducibility choice and a limitation. Claims are therefore restricted to the retrieval/reuse controller. The API repair suite contains seven curated migration defects and cannot support a general automated-program-repair claim. Finally, the benchmark uses one split per task. Functional outcomes are deterministic under the fixed environment, but timing and model scores may vary across hardware or future library versions.

8. Conclusion

PIPE-RAG reframes retrieval-augmented ML code generation as constrained software reuse with execution-grounded acceptance. Typed cards, hybrid retrieval, version-aware MMR, exact reranking, request-driven recomposition, and bounded repair jointly eliminate the failure gap observed in simpler baselines on the controlled benchmark. Across 36 requests and 144 executions, the framework reaches 100% task-complete functional success, a 41.67-point improvement over hybrid RAG, while adding roughly two milliseconds of local synthesis latency. Seven historical API variants also demonstrate the value of error-triggered migration rules. The contribution is not a new foundation model; it is a reproducible control layer that can make an LLM-based or deterministic generator more accountable. Future work should evaluate human-authored requests, larger code repositories, learned code embeddings, multiple library versions, and a genuinely executed open-weight LLM fallback under the same task-complete metric.

Appendix A. Benchmark And Acceptance Details

The task matrix is balanced by construction across four classification datasets and two regression variants. Each classification dataset receives six requests: logistic regression with accuracy and

standardization; support-vector classification with macro-F1 and standardization; random-forest classification with accuracy; logistic regression with macro-F1, standardization, and balanced class weights; support-vector classification with accuracy, standardization, and feature selection; and random-forest classification with macro-F1 and balanced class weights. The numeric Diabetes dataset receives Ridge/RMSE with scaling, random-forest/R2, gradient-boosting/RMSE, Ridge/R2 with scaling and feature selection, random-forest/RMSE with imputation, and gradient-boosting/R2 with imputation. The mixed Diabetes variant uses the same estimator/metric coverage while requiring imputation and one-hot encoding, with scaling or feature selection where specified.

Card relevance is deliberately stricter than textual similarity and deliberately narrower than final request compliance. A retrieved card counts as Recall@1-relevant only when problem type, schema type, estimator, and metric match. Requirement sets are used in scoring but are not part of this binary retrieval label because strict recomposition is designed to adapt optional preprocessing. Final compliance is evaluated on generated source rather than selected metadata. This separation avoids crediting a card merely because its description claims compatibility, and it avoids penalizing retrieval for a requirement that the controlled constructor can add safely.

The acceptance harness records four unconditional predicates—use of sklearn Pipeline, requested estimator, requested metric function, and assignment of both predictions and score—plus only the conditional predicates present in the request. A task asking for standardization therefore has five checks; one asking for mixed-schema imputation, one-hot encoding, and feature selection has seven. Constraint satisfaction is the fraction of applicable checks, while all-constraints is their conjunction. Functional success additionally requires actual execution and finite outputs. The model_score column is retained as an execution witness, not as a cross-method quality comparison, because different requests legitimately target different algorithms and metrics.

Every task-method row stores the prompt, selected card, API status, top-five ranking, retrieval score, syntax and runtime outcomes, first-pass status, repair rules, error message, constraint predicates, timing, model score, and complete generated source. The repair stress file separately stores each stale marker, the real pre-repair exception raised by scikit-learn 1.8.0, applied rules, and post-repair execution. These artifacts permit independent recomputation of all aggregate rates and paired tests without parsing the manuscript.

References

[1] Lewis, P., et al. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In Advances in Neural Information Processing Systems.

[2] Guu, K., Lee, K., Tung, Z., Pasupat, P., & Chang, M.-W. (2020). Retrieval augmented language model pre-training. In Proceedings of the ICML (PMLR 119).

[3] Karpukhin, V., et al. (2020). Dense passage retrieval for open-domain question answering. In Proceedings of the EMNLP. https://doi.org/10.18653/v1/2020.emnlp-main.550

[4] Borgeaud, S., et al. (2022). Improving language models by retrieving from trillions of tokens. In Proceedings of the ICML (PMLR 162).

[5] Robertson, S., & Zaragoza, H. (2009). The probabilistic relevance framework: BM25 and beyond. Foundations and Trends in Information Retrieval. https://doi.org/10.1561/1500000019

[6] Carbonell, J., & Goldstein, J. (1998). The use of MMR, diversity-based reranking for reordering documents and producing summaries. In Proceedings of the SIGIR. https://doi.org/10.1145/290941.291025

[7] Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., & Brockschmidt, M. (2019). CodeSearchNet challenge: Evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436.

[8] Feng, Z., et al. (2020). CodeBERT: A pre-trained model for programming and natural languages. In Findings of EMNLP. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[9] Guo, D., et al. (2021). GraphCodeBERT: Pre-training code representations with data flow. In Proceedings of the ICLR.

[10] Wang, Y., Wang, W., Joty, S., & Hoi, S. C. H. (2021). CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Proceedings of the EMNLP. https://doi.org/10.18653/v1/2021.emnlp-main.685

[11] Ahmad, W. U., Chakraborty, S., Ray, B., & Chang, K.-W. (2021). Unified pre-training for program understanding and generation. In Proceedings of the NAACL-HLT. https://doi.org/10.18653/v1/2021.naacl-main.211

[12] Wang, Y., et al. (2023). CodeT5+: Open code large language models for code understanding and generation. In Proceedings of the EMNLP. https://doi.org/10.18653/v1/2023.emnlp-main.68

[13] Chen, M., et al. (2021). Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374.

[14] Austin, J., et al. (2021). Program synthesis with large language models. arXiv preprint arXiv:2108.07732.

[15] Fried, D., et al. (2023). InCoder: A generative model for code infilling and synthesis. In Proceedings of the ICLR.

[16] Chen, X., Lin, M., Schärli, N., & Zhou, D. (2024). Teaching large language models to self-debug. In Proceedings of the ICLR.

[17] Shinn, N., et al. (2023). Reflexion: Language agents with verbal reinforcement learning. In Advances in Neural Information Processing Systems.

[18] Ni, A., et al. (2024). NExT: Teaching large language models to reason about code execution. In Proceedings of the ICML (PMLR 235).

[19] Jain, N., et al. (2021). Jigsaw: Large language models meet program synthesis. arXiv preprint arXiv:2112.02969.

[20] Pedregosa, F., et al. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research.

[21] Thornton, C., Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2013). Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In Proceedings of the KDD. https://doi.org/10.1145/2487575.2487629

[22] Feurer, M., et al. (2015). Efficient and robust automated machine learning. In Advances in Neural Information Processing Systems.

[23] Olson, R. S., & Moore, J. H. (2016). TPOT: A tree-based pipeline optimization tool for automating machine learning. In AutoML Workshop, ICML (PMLR 64).

[24] Olson, R. S., Bartley, N., Urbanowicz, R. J., & Moore, J. H. (2016). Evaluation of a tree-based pipeline optimization tool for automating data science. In Proceedings of the GECCO. https://doi.org/10.1145/2908812.2908918

[25] Olson, R. S., & Moore, J. H. (2016). Identifying and harnessing the building blocks of machine learning pipelines for sensible initialization of a data science automation tool. arXiv preprint arXiv:1607.08878.

[26] Kaufman, S., Rosset, S., Perlich, C., & Stitelman, O. (2012). Leakage in data mining: Formulation, detection, and avoidance. ACM Transactions on Knowledge Discovery from Data. https://doi.org/10.1145/2382577.2382579

[27] Kelly, M., Longjohn, R., & Nottingham, K. (n.d.). The UCI machine learning repository. UCI.

[28] Scikit-learn Developers. (n.d.). Toy datasets and dataset loading utilities. Scikit-learn documentation.

[29] Liu, S., Chen, Y., Xie, X., Siow, J., & Liu, Y. (2020). Retrieval-augmented generation for code summarization via hybrid GNN. arXiv preprint arXiv:2006.05405.

[30] Ye, W., et al. (2020). Leveraging code generation to improve code retrieval and summarization via dual learning. In The Web Conference (WWW). https://doi.org/10.1145/3366423.3380295

[31] 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, 14, 41932–41945. https://doi.org/10.1109/ACCESS.2026.3673923

[32] Mo, T., Zhang, C., Zou, J., Guo, Z., & Rhee, M. (2026). Self-evolving AI agents with dual memory for automated software testing and bug localization. IEEE Access, 14, 111086–111102. https://doi.org/10.1109/ACCESS.2026.3713401

[33] Rhee, M., Zou, J., Mo, T., Teng, D., & Yang, J. S. (2026). Enhancing web search agents with self-play contrastive fine-tuning. IEEE Access. https://doi.org/10.1109/ACCESS.2026.3717594

[34] 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, 14(12), 2137. https://doi.org/10.3390/math14122137