Academic Journal of Applied Sciences
ISSN: 3105-0549 (Print), ISSN: 3105-0557 (Online) | Vol. 3, No. 2, 2026

Budgeted Two-Stage Pre-Validation for Retrieval-Grounded Machine Learning Pipelines: A Group-Held-Out Evaluation

Maya Jenkins [1, *], Thomas Baker [1], Theo Hernandez [2]

[1] College of Computing & Informatics, Drexel University, USA

[2] Department of Computer Sciences, University of Wisconsin–Madison, USA

[*] **Corresponding author**: Maya Jenkins

Abstract: Retrieval-grounded machine-learning pipeline synthesis must reconcile code templates with package versions and resource budgets. We study a two-stage pre-validator adapted from concrete dependency-satisfiability and resource-prediction mechanisms: an inexpensive retrieval-profile margin ranks candidate manifests; a budgeted second stage invokes a gradient-boosted peak-memory predictor and applies bounded repairs. Actual execution of 48 public-dataset learning workflows supplies memory profiles, while eight controlled configuration mutations per workload define explicit local feasibility outcomes. Four group-disjoint test folds provide 384 held-out candidates. At an exact 50% second-stage invocation budget, retrieval-prioritized validation yields 372/384 feasible configurations (96.9%), matching exhaustive checking, versus 281/384 (73.2%) for one matched random schedule. Across 300 random schedules the mean is 74.2%. A 16-group paired bootstrap puts the advantage over that seeded random baseline at 19.8–27.1 percentage points. Uncertainty offsets and semantic novelty are not independently beneficial on this coarse-fault benchmark: point prediction and a simple memory-margin ranking tie the proposed method. Calibration also deteriorates on a shifted fold. These findings support auditable, low-cost screening for synthetic configuration faults, without treating local feasibility as actual cloud deployment success or deterministic template synthesis as LLM generation.

Keywords: Machine learning pipelines, retrieval-guided generation, budgeted pre-validation, resource prediction, dependency satisfiability, uncertainty under shift.

1. Introduction

Machine-learning pipelines are collections of executable steps whose code, dependencies, and runtime allocations must agree. Configuration drift, hidden data interfaces, and external dependencies can be as consequential as model logic [1, 2]. Production architectures such as TFX demonstrate the value of standardized orchestration [3], whereas deployment case studies show that cross-component integration remains difficult [4]. The practical goal is therefore not only to synthesize plausible source files but also to discriminate between configurations that can and cannot run under the chosen execution contract.

Retrieval offers one form of grounding. The retrieval-augmented generation paradigm combines an external memory with a generator [5], and code-completion systems have demonstrated how code context can be recovered from external repositories [6, 7]. Documentation retrieval is useful for API selection [8]. Yet a high textual match does not establish that the retrieved environment has a compatible library version, nor that its memory budget covers the intended workload. Such mismatches are especially visible in automated ML workflows.

We isolate a measurable operational contract instead of conflating configuration repair with neural code-generation accuracy. A deterministic retrieval-conditioned synthesizer emits Python training scripts and YAML manifests. Public datasets are used to fit classification models and measure process-level peak resident-set size (RSS), while controlled mutations introduce version and memory faults. The experimental oracle checks package compatibility and observed RSS; it does not submit jobs to a cloud scheduler.

The contributions are threefold. First, we realize an executable version-CSP and historical-trace resource predictor, explicitly extending the concrete pre-validation components studied in prior cloud-native work. Second, we introduce a gated two-stage procedure: a cheap retrieved-profile score schedules expensive model predictions under an exact batch budget, so skipped candidates do not invoke the predictor. Third, we evaluate disjoint held-out workload groups against matched random screening, exhaustive validation, simple-margin screening, and uncertainty ablations, with group-clustered intervals and a separate calibration-shift diagnosis.

2. Related Work

2.1. Code Synthesis and Retrieval

General-purpose code models, including CodeT5 [9] and CodeT5+ [10], establish strong learned code representations but are separate from deployment-time configuration checking. CodeRL uses learned criticism and test feedback during program generation [11]. Self-RAG demonstrates that retrieval need not be invoked uniformly for every request [12]. Our question is different: given a candidate executable configuration, which validation should be applied before it is accepted? No neural model is trained or queried in our experiment, and the benchmark is not a code-completion leaderboard.

2.2. Executable Pipeline Validation

Data-quality verification can be formulated declaratively and executed automatically, as illustrated by Schelter et al. [13]. Workload-aware pipeline optimization also benefits

from profiling and cost models; Plumber exemplifies this approach for data processing stages [14]. CodeSearchNet provides a large, independently useful retrieval benchmark [15], but its code-search labels cannot establish memory safety or environment viability. We therefore construct a smaller operational benchmark whose success criterion can be audited against local execution traces and explicit constraints. PIPE-RAG demonstrates how retrieved machine-learning workflow exemplars can be combined with execution checks to improve the reliability of generated pipeline artifacts [27].

2.3. Retrieval Features and Confidence

Probabilistic lexical retrieval methods such as BM25 [16], sentence-level dense representation learning [17], and dense passage retrieval [18] represent alternative relevance signals. To avoid a large encoder dependency in the reproduction package, our generator uses TF–IDF cosine similarities combined with a fixed structural compatibility score. Retrieval quality and runtime safety are deliberately treated as distinct concerns: the former determines the initial proposal; the latter is a constraint-checking decision on that proposal.

3. Scope and Formal Problem

A task s specifies the input dataset, estimator family, sample fraction, and a local CPU execution target. A generated artifact consists of a Python script implementing load–split–fit–evaluate stages and a YAML manifest with three package constraints and an assigned resident-memory budget. Its pipeline graph is acyclic by construction. The historical collection H stores task descriptions, executable synthesis templates, installed-environment constraints, and measured peak resident set size (RSS). Each retrieved profile is backed by a training subprocess, rather than an asserted cloud profile.

We define local operational feasibility F(c) as the conjunction of a satisfiable installed-version domain and a budget at least as large as the measured peak RSS of the corresponding task. This is a post hoc ground-truth oracle for the benchmark; it is not exposed to the pre-validator at test time. The actual training procedure is executed to establish RSS and predictive accuracy; resource insufficiency is assessed against the saved RSS, rather than by applying an operating-system hard memory cap. The latter could change failure behavior through allocator and cgroup effects.

$$F(c) = 1\{\, CSP(c) = SAT \,\} \cdot 1\{\, M_alloc(c) \geq M_peak(s) \,\} \quad (1)$$

For a policy π that selects manifest candidates for second-stage resource prediction within a batch C, the objective is to maximize mean post-repair local feasibility subject to the cap $\sum_{c \in C} \pi(c) \leq \lfloor B|C| \rfloor$. The accounting unit is an invoked resource predictor; an unselected manifest never calls this predictor. Package-CSP checking is performed on every candidate. All candidates share the same held-out RSS evaluation oracle after repair, independent of the checking policy.

4. Method

4.1. Trace-grounded Retrieval and Synthesis

Each fold indexes historical execution profiles from nine training dataset–estimator groups (27 workflows). Natural-language descriptions use train-fitted unigram/bigram TF–IDF vectors; the query is compared with historical descriptions using cosine similarity. Dual retrieval also scores dataset, estimator, and platform matches at respective weights 0.35, 0.40, and 0.25, then blends lexical and structural scores in a 0.55:0.45 ratio. The best historical profile supplies the initial memory estimate. Eight controlled perturbations scale its proposed memory between 0.61 and 1.80 and may insert a conflicting NumPy range.

$$score(s,h) = 0.55 \cos(T(s),T(h)) + 0.45\, str(s,h) \quad (2)$$

These perturbations form a controlled repair benchmark, not samples of a language model's error distribution. The generated Python is a template implementation for one of four fixed sklearn estimator families. This design intentionally makes operational checking observable, eliminates API billing variability, and yields a baseline that can be rerun offline. The scripts and YAML outputs are saved separately so that code synthesis is more than an in-memory label-generation step.

4.2. Implemented Dependency and Resource Checks

Zhao et al. describe a pre-validation agent that combines version-constraint satisfiability with learned resource estimation and iterative feedback [19]. We implement the two concrete checks and the targeted repair step in a deliberately narrower, locally pinned setting. The dependency checker creates package nodes for NumPy, SciPy, and scikit-learn and dependency edges from scikit-learn to its prerequisites. For each package it intersects the requirement specifier with the finite installed-version domain; an empty domain or absent required node fails validation. When a conflict is detected, the repair step writes the known-compatible training-environment constraints to the YAML manifest.

$$CSP(c) = SAT \iff \forall p,\ D_installed(p) \cap Spec(c,p) \neq \emptyset \quad (3)$$

When selected, the second stage fits or consults a gradient-boosted decision-tree regressor that predicts peak RSS from sample count, feature count, class count, dataset, and estimator family. Gradient boosting implements stage-wise function approximation [20]. The trained regressor is distinct from the inexpensive historical-profile memory value available at retrieval time. It is called only for selected test candidates in the new implementation. A violated memory budget is raised to the predicted requirement (optionally with a calibrated upper residual) plus 2 MiB. The per-candidate records explicitly log whether prediction ran.

4.3. Residual Uncertainty and Selective Triggering

Nine training groups (27 workflows) train the resource model, and three further groups (nine workflows) calibrate a one-sided split residual offset within each fold. The nominal confidence parameter is 0.90 and the resulting finite-sample order statistic is clipped below at zero, following the split-conformal construction [21, 22]. Because the evaluation deliberately holds out entire dataset–estimator groups, residual exchangeability may fail. We therefore report actual upper-bound coverage for each fold and do not infer a 90% guarantee under workload shift.

$$q = \max\{0,\ r_(\lceil (n_cal+1)\cdot 0.90 \rceil)\},\ r = M_peak - M_pred \quad (4)$$

The first-stage scheduling score requires only the retrieved historical peak M_ref, the candidate allocation M_alloc, and retrieval similarity sim; it does not use a gradient-boosted

prediction. For each 96-candidate test batch, select exactly the 48 highest-ranked cases (ties resolved by stable task/variant order). A chosen candidate then computes U(c)=M_pred(c)+q and updates its allocation to U(c)+2 MiB only if below U(c). Thus the cheap rank controls when the resource model runs, while residual uncertainty controls how much a selected allocation changes.

$$\text{priority}(c) = (M_\text{ref}(c) - M_\text{alloc}(c))/M_\text{ref}(c) + 0.16\cdot(1 - \text{sim}(c)) \quad (5)$$

The relevance/novelty adjustment increases priority for less familiar contexts but is a heuristic rather than a calibrated error probability. We compare against the identical selection rule with q=0, a margin-only score that omits novelty, semantic-only retrieval, exhaustive model calls, and exact-budget random ranking. This separation tests whether uncertainty changes repair outcomes without confusing the extra computation of a full precomputation stage with the invocation budget. The policy resembles selective evaluation rather than selective classification [23]; confidence calibration references [24, 25] concern related but distinct calibration targets.

4.4. Generation and Correction Algorithm

For each input batch, build candidates from retrieved reference templates and controlled perturbations, check every package version CSP, and repair conflicts to the known target image. Rank candidates by the inexpensive score in (5) and select the top half. For selected candidates only, invoke the gradient-boosted memory model, add the fold-specific residual offset, and repair insufficient allocations. Exhaustive validation selects every candidate; random validation selects a reproducible half without inspecting candidate feasibility; margin-only validation removes the novelty term. At no stage is the observed RSS of the held-out task used to guide selection or repair.

4.5. Computational Rationale and Safety Interpretation

Dependency version checking parses a few locally pinned specifiers. The first-stage memory score uses a previously retrieved scalar and similarity; it is cheaper than fitting or invoking a separate gradient-boosted model. Crucially, the revised implementation postpones that model call until after screening rather than using its prediction to decide whether to call it. The achieved 50% budget is therefore a count of actual estimator calls along each method's evaluation path. It does not translate directly to a 50% wall-time or dollar reduction, since retrieval and other tasks still run for all candidates.

For an input candidate with a negative memory margin, the policy can increase the allocation without executing the candidate under a hard memory limit. This is intentional: validation must happen before deployment. The model may nevertheless be wrong. An erroneously low upper bound permits underprovisioning even after exhaustive checking; an overly conservative bound preserves feasibility while wasting memory. A dependency fix can also pass the local installed-version test without guaranteeing behavior under different package build tags, ABI constraints, or transitive registries. These residual hazards are tracked separately from the success indicator in our interpretation.

An alternative is to send every configuration to a sandbox runtime and iteratively refine after a failure. That is appropriate when cloud test namespaces are available, but it confounds code quality with service interruptions and increases the number of executions. The present prototype instead preserves the two informational steps—resolve package constraints and predict memory pressure—while using measured executions as a post hoc oracle for an isolated resource contract. This separation enables a paired comparison of scheduling policies without claiming full emulation of cloud deployment semantics.

4.6. Executable Policy Specification

Algorithm 1 describes the revised batch-aware implementation (robust_evaluation.py). Every test group is encountered once as a held-out group among four folds. Each fold forms 96 mutated configurations from 12 tasks and enforces an exact 48-candidate second-stage budget. Selection does not use a test resource oracle; all oracle RSS values originate from prior, individually executed workload measurements. The older single-split median-threshold variant remains separately reproducible in run_experiment.py and is retained as a historical sensitivity check, not merged with the four-fold outcomes.

ALGORITHM 1. Budgeted two-stage pre-validation (robust_evaluation.py).

Table 1. Executed stages and call-gating semantics of the two-stage policy.

Step	Executed operation
1	Retrieve a historical run profile; generate candidate manifests
2	Run pinned-version CSP on all candidates and repair conflicts
3	Compute cheap reference-memory risk score; rank the batch
4	Select exactly B·N candidates, without model prediction
5	Only for selected: call boosted RSS predictor and add q
6	If allocated < predicted upper bound, raise budget by 2 MiB
7	Evaluate all candidates with independently measured RSS oracle

5. Experimental Design

5.1. Data Provenance and Profiling

Four bundled, public benchmark datasets are loaded through scikit-learn [26]: Iris (150 examples, four features), Wine (178 examples, 13 features), Breast Cancer Wisconsin Diagnostic (569 examples, 30 features), and optical handwritten Digits (1,797 examples, 64 features). We cross four deterministic estimator families—logistic regression, a 24-tree random forest, a depth-limited decision tree, and Gaussian naive Bayes—with subsampling fractions of 55%, 75%, and 100%. This generates 48 unique training tasks; actual model fits produce each task's RSS profile and holdout classification accuracy.

An explicitly synthetic temporary feature-staging buffer is allocated and touched during fitting to create heterogeneous memory pressure. Its configured size depends on dataset, estimator, and sample fraction. All reported RSS values are measured via the Linux peak-resident-memory counter in a separate Python subprocess, inclusive of imported libraries and this buffer. The feature-staging stressor is a test harness manipulation, not a naturally observed cloud telemetry trace. Dataset arrays themselves are shipped with the normal scikit-learn distribution; no external dataset download or private log

access is required.

5.2. Split, Faults, And Comparisons

The 16 dataset–estimator families are shuffled into four disjoint folds of four test groups each. For every fold, the remaining 12 groups are divided deterministically into nine training and three calibration groups. All 48 measured task profiles occur as test observations in exactly one fold; each test fold contains 12 actual workloads and 96 mutated manifest candidates. Across all folds, this yields 384 group-held-out candidates per policy. Two or more mutations of the same workload never cross training, calibration, and test boundaries within a fold.

The main comparisons are dual retrieval without checking, dependency CSP alone (the version-constraint check of [19]), exhaustive upper-bound checking, exhaustive point prediction, retrieval-prioritized checking with and without residual q, simple margin-only checking, semantic-only prioritization, and a fixed-seed random scheduler with the identical 50% call budget. The 300 additional random schedules isolate sampling variability, not independently rerun training. All variants use matching held-out candidates and training splits within each fold. We do not compare against an unexecuted neural generator.

5.3. Metrics and Uncertainty Reporting

We report post-repair local feasibility, the fraction of candidates actually invoking the boosted predictor, and overall memory over-allocation relative to the measured RSS oracle. A reproducible fixed-seed random schedule permits per-candidate paired comparisons. Intervals use 5,000 bootstrap resamples of the 16 unique held-out dataset–estimator groups, rather than treating all 384 mutated configurations as independent. The 300 random schedules additionally yield a dispersion summary for the random-budget comparator. Residual quantile coverage is reported per fold because workload-group shift can violate exchangeability.

5.4. Execution Contract and Implementation Traceability

All measurement subprocesses run with a fixed estimator seed. The requested 55% or 75% fraction is obtained using stratified training data selection, and each training task is further split into 75% fitting and 25% held-out scoring data. The reported peak RSS includes the loaded dataset, imported numerical libraries, the deliberately materialized feature-staging array, and fitted model state. Classification accuracy is stored to establish that actual estimators ran, but it is not optimized by any validation policy and is not presented as a code-generation benefit.

The installed local package versions are recorded in the metadata and form the complete version-domain catalog. For each candidate, the checker evaluates the `packaging.specifiers` semantics on the installed NumPy, SciPy, and scikit-learn versions and the two specified dependency edges. Repair overwrites the candidate with the benchmark environment's compatible version ranges; this is a known target image repair, rather than unrestricted dependency solving. The memory predictor never receives the mutation label or test oracle RSS. Its features are computed from the task schema, independently of the injected configuration.

Although the scripts use the syntax of a multi-step workflow, they do not launch a pipeline scheduler: they perform load, split, fit, and evaluation in one local process. The adjacent YAML is an intermediate manifest rather than an Azure or Kubeflow native submission. We deliberately retain both files to show that retrieved metadata influences a concrete generated artifact. End-to-end cloud execution, multi-container topology, policy permissions, and pip-wheel resolution are deferred to a deployment-specific implementation, where the same checks could precede a full platform smoke test.

6. Results and Analysis

6.1. Main Comparison

Table 2. Four group-disjoint test folds: local post-repair feasibility (384 controlled manifests).

Method	Feasible	Predict	Overalloc.
Retrieval only	101/384	0.0%	20.6%
CSP only	197/384	0.0%	20.6%
Random 50%	281/384	50.0%	21.5%
Margin 50%	372/384	50.0%	22.4%
Priority, q=0	372/384	50.0%	21.7%
Priority, q	372/384	50.0%	22.4%
Exhaustive, q	372/384	100.0%	22.4%
Semantic, q	372/384	50.0%	22.1%
Exhaustive, q=0	372/384	100.0%	21.7%

Across four disjoint held-out folds, prioritized checking produces 372/384 feasible configurations (96.9%) using 192 resource-model invocations. Retrieval alone produces 101/384 (26.3%), and compulsory CSP without resource prediction yields 197/384 (51.3%). One fixed-seed 50%-budget random scheduler produces 281/384 (73.2%). The priority–random paired advantage is 23.7 percentage points (95% group-cluster bootstrap interval 19.8–27.1 points). These results measure correction of intentionally injected version and allocation faults; they do not represent an LLM benchmark or production deployment rate.

Both the fully exhaustive procedure (384 predictions) and the prioritized procedure (192 predictions) finish with 372 feasible candidates. The reduction from 384 to 192 is a real, instrumented change in model invocation count. Prioritized scheduling allocates calls to manifests with insufficient retrieved-resource margins, whereas a random budget frequently spends checks on already-feasible allocations. Across 300 rerandomizations of the 50%-budget scheduler on the same candidate sets, its mean feasibility is 74.2% (standard deviation 1.2 percentage points; 5th–95th percentile 72.4–76.0%).

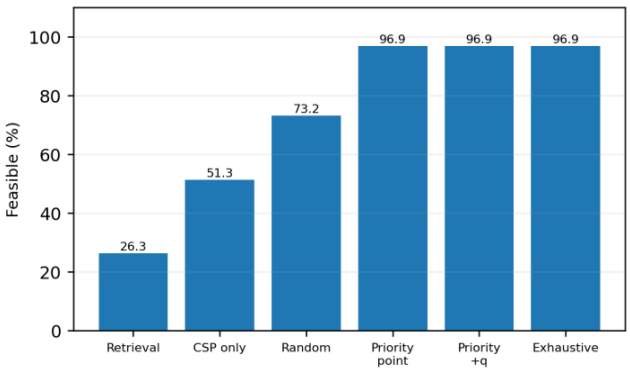


Figure 1. Cross-fold local feasibility for budgeted, exhaustive, and control policies. Each bar uses the 384 executed per-candidate records.

6.2. Ablation and Source Mechanism

Version CSP follows the concrete version-constraint analysis strategy used in the pre-validation framework of Zhao et al. [19]. Its contribution is isolated here: the CSP-only policy yields 197/384 (51.3%), compared with 101/384 (26.3%) for unchecked retrieval. The check operates on a local finite version domain; complete PyPI transitive resolution lies outside its scope. All resource-aware policies also perform CSP repair, so their gains cannot be credited solely to memory prediction.

The strongest ablations are deliberately inconclusive about sophisticated uncertainty: prioritized scoring with the residual q yields the same 372/384 successes as q=0, and margin-only prioritization also ties. Switching to semantic-only retrieval preserves feasibility in the observed synthetic task distribution. These ties show that the simple memory headroom ranking already identifies the large injected underallocations; novelty penalties and calibrated residual offsets are not independently validated as sources of improvement. The principal empirically supported mechanism is ranking calls to the second-stage predictor under an exact cap.

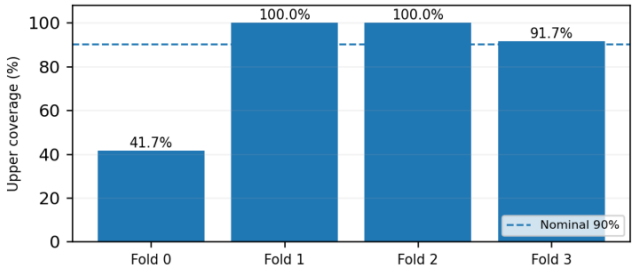


Figure 2. Group-held-out empirical coverage of the one-sided upper RSS predictions. The nominal reference is 90%; one shifted fold falls far below it.

6.3. Calibration Quality and Execution Checks

The four fitted residual offsets q are 0.00, 2.74, 5.64, and 1.55 MiB. Their corresponding held-out task-level upper-bound coverages are 41.7%, 100.0%, 100.0%, and 91.7%; the aggregate is 83.3%, below the nominal 90%. Because four groups are held out per fold and only nine calibration tasks determine q, the gap is a material calibration risk rather than a nuisance to hide. The first fold exposes a shifted resource family whose model underestimates memory. Calibration quality must be monitored independently of binary local configuration feasibility.

In addition to 48 measured workload profiles, the benchmark materializes 48 local Python entrypoints and 384 corresponding post-repair YAML configurations, one for each controlled fault in its held-out fold. All 48 generated entrypoints were executed in separate local Python subprocesses and returned exit code zero with a classification-accuracy result. Their YAML files are static local manifests and do not enforce a hard memory cap or orchestrate cloud services; their success is distinct from the configuration-feasibility oracle.

6.4. Resource-Model Diagnostics and Budget Response

Across the 48 actually run workflows, peak RSS ranges from 172.1 to 218.3 MiB. Under the four-fold analysis, per-fold held-out mean absolute errors are 3.51, 1.79, 3.41, and 4.30 MiB; these values describe resource prediction across unseen workload groups, including the deliberately materialized scratch buffer. They are not estimates of actual cloud resource consumption.

Table 3 separates model error and calibration coverage by disjoint test fold. Its 96 manifest variants per fold inherit only 12 distinct observed peak-RSS task outcomes, so the appropriate unit for calibration coverage is the task, not every correlated manifest perturbation. The fold-0 failure demonstrates why q should not be called a distribution-free safety bound when calibration and test task groups differ.

Table 3. Group-held-out resource calibration across four disjoint test folds.

Fold	q (MiB)	Coverage	RSS MAE
0	0.00	41.7%	3.51
1	2.74	100.0%	1.79
2	5.64	100.0%	3.41
3	1.55	91.7%	4.30
Pooled	—	83.3%	3.25

The four-fold comparison additionally checks whether selective schedules merely exploit their 50% call limit. They do not: the seeded random scheduler lags the prioritized schedule on the same inputs by 91 feasible candidates, and the 300 random reschedules remain below the proposed method. Nevertheless, the margin-only scheduler matches the full priority score, demonstrating that sophisticated similarity and uncertainty features do not improve ordering for these eight coarse perturbations. Validation gains should therefore be ascribed to selective headroom checking, rather than an unsupported claim of calibrated epistemic uncertainty handling.

6.5. Counterexamples and Failure Interpretation

Even exhaustive prediction leaves 12/384 candidates locally infeasible. In one held-out fold the resource estimator is biased downward and q collapses to zero; a corrected allocation can still fall below independently observed peak RSS. This is a resource-model failure shared by exhaustive and prioritized checking, rather than a failure of the selection budget. A safe deployment system would need shift detection, an out-of-distribution fallback, and a real sandbox before accepting high-stakes workloads.

The additional randomized comparison also has a clear limitation: the 300 schedules reuse the same 48 observed workload executions and the same synthetic fault set. It estimates variation due to random verification selection, not to new environments or new datasets. Our cluster bootstrap resamples 16 disjoint test groups within these experiments; overlapping datasets and training folds still make its numerical interval narrower in scope than a population-level confidence statement about cloud infrastructure.

A further distinction is between policy calls and platform cost. The procedure prevents unnecessary gradient-boosting predictions, unlike a risk gate that must run the same regressor before deciding whether it should be called. The observed counts are 192 and 384, but retrieval, manifest parsing, and first-stage sorting run regardless, and no timed cloud charges or GPU allocations were measured. In a production pre-validator, the first stage would be useful when second-stage simulations or telemetry models have material computational or service overhead.

6.6. Predictor Invocation and Accounting Semantics

Prioritization is evaluated under a *batch* budget, so the selection of one manifest can depend on the other manifests in the batch. Per-case responses therefore cannot be interpreted as independently scheduled online decisions without a queue or windowing policy. The distinction is important in practice: a fully online arrival-by-arrival threshold might check more or fewer than 50% of a shifted batch, whereas the rank-based policy guarantees exactly 48 of 96 model calls. The benchmark tests the latter service-level budget, with no oracle information included in the rank.

All task profiles were measured first by subprocess execution, before any fold-specific resource predictor was trained. In the four-fold experiment, the same 48 profiles are reused with strictly disjoint train/calibration/test group assignments within each fold. Prediction call counts refer to calls made for a configuration under its evaluated policy, not model fitting on training data or calls to generate the previously saved profiles. A deployment system would amortize training separately and would need to report end-to-end latency and resource cost in addition to invoked predictions.

6.7. Uncertainty Ablation and Interpretation

The point-prediction variant and upper-residual variant tie in all four test folds, notwithstanding different q values. A positive q changes some repaired budgets and therefore mean over-allocation, but it does not change the binary outcome of these coarse controlled cases. The first fold additionally has a zero clipped offset; no claim of robust conditional coverage is available from that fold. This supports a narrow reading of the evidence: simple margin screening identifies the fault structure while the present residual uncertainty estimator provides no demonstrated incremental feasibility improvement. Natural LLM code, real registry conflicts, and heterogeneous devices would be needed to probe whether uncertainty changes which cases are repaired. Compiler-in-the-loop retrieval-augmented program migration provides a complementary experimental example of separating syntactic validity checks from downstream functional verification [28].

7. Validity, Reproducibility, And Implications

Several features constrain external validity. The generator is deterministic and template-bound, so observed gains cannot be carried over to unconstrained LLM output or code semantics not present in the template grammar. The environment comprises three installed Python packages, not a full resolver for heterogeneous environments or transitive registry constraints. This makes the CSP stage operational and auditable, but relatively easy compared with resolving real cloud container images. The synthetic fault schedule intentionally stresses a narrow, interpretable set of pre-deployment failures.

The datasets are public and all 48 training operations are executed, but staging-memory buffers and mutated manifests are benchmark constructions. Peak RSS is a process-level measurement and may change with operating-system page accounting, versions, and parallel load. Constraint feasibility under a measured memory envelope is also weaker than deployment viability; network permissions, storage access, scheduler contention, and unmodeled APIs are outside the test oracle. A more demanding validation would use real cloud job logs and cgroup limits, multiple runtime images, and a natural-error distribution collected from candidate code models.

The strongest supported contribution is budgeted pre-validation: in a 384-candidate four-fold local benchmark, a low-cost resource headroom ranking matches exhaustive post-repair feasibility while invoking the boosted model on half the candidates. That conclusion does not imply that the novelty term, conformal residual, or dual retrieval is individually superior. Future work should first challenge the ranking on natural LLM-generated allocation errors and improve shift-aware uncertainty calibration before drawing cloud-deployment conclusions.

The research artifact includes the four group-disjoint folds, exact-budget randomized baselines, 5,000-replicate group bootstraps, measured-profile reuse, and 300 randomized selection repetitions. Files include fourfold_per_case.csv, fourfold_analysis.json, and the complete fold group definitions, alongside generated Python/YAML examples and 48 authentic execution profiles. Randomized selections and synthetic mutations are transparent and reproducible.

7.1. Threats to Causal Interpretation

The study uses paired candidate inputs, but different method labels do not correspond to independently trained foundation models. Therefore the main difference between retrieval-only and validated generation is a difference in a deterministic repair policy on the same mutant manifests. The chosen fault mix includes known incompatible specifiers and budget multipliers, which are easier to diagnose than semantic faults in arbitrary generated programs. The experiment is best interpreted as a component-level causal test of the validator and its selective invocation logic. It cannot justify quantitative comparisons to GPT-4, CodeT5+, Azure ML, or other systems that were not executed in the present software environment.

We also separate three notions of success. A generated file can be syntactically executable; a manifest can satisfy the package and resource oracle; and a cloud service can deploy and finish a distributed pipeline. We measure the first by subprocess exit status and the second by explicit constraint checks with real workload profiles. The third was never observed and is not substituted with either of the first two. This layered terminology avoids reading a local test-set rate as a cloud first-submission success rate.

7.2. Practical Steps Toward Cloud Integration

The current code suggests an incremental integration path. The fixed version catalog could be replaced by a repository-aware resolver over container lockfiles and transitive constraints. The memory regressor could ingest CPU and GPU telemetry keyed to platform instance families and use a recalibration set stratified by device type. The manifest generator could target a real orchestrator-specific schema, and the local success oracle could be replaced by a job submission with recorded scheduler status and memory peaks. Lastly, richer retrieval embeddings and a code language model could be inserted as upstream generators while preserving the validator interfaces and paired comparison protocol.

8. Conclusion

We present a measurable two-stage pre-validation design for retrieval-grounded ML pipeline configuration. A concrete

version-CSP checker and measured execution-based memory regressor repair controlled local faults, while a cheap retrieved-profile margin gates resource-predictor invocations. Four held-out workload-group folds yield 372/384 feasible configurations with 192 second-stage calls, matching exhaustive validation with 384; a matched random half-budget scheduler achieves only 281/384. The equally strong simple-margin and point-prediction ablations show that these results support selective budget allocation, not an independently established uncertainty benefit. The separate 41.7% held-out upper-bound coverage in one fold motivates explicit robustness testing under workload shift. Validating on real LLM outputs and cloud orchestrators remains the next research step.

References

[1] Sculley, D., et al. (2015). Hidden technical debt in machine learning systems. In Advances in Neural Information Processing Systems (Vol. 28, pp. 2503–2511). https://research.google/pubs/hidden-technical-debt-in-machine-learning-systems/

[2] Polyzotis, N., Roy, S., Whang, S. E., & Zinkevich, M. (2017). Data management challenges in production machine learning. In Proceedings of the ACM SIGMOD International Conference on Management of Data (pp. 1723–1726). https://doi.org/10.1145/3035918.3054782

[3] Baylor, D., et al. (2017). TFX: A TensorFlow-based production-scale machine learning platform. In Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (pp. 1387–1395). https://doi.org/10.1145/3097983.3098021

[4] Paleyes, A., Urma, R.-G., & Lawrence, N. D. (2023). Challenges in deploying machine learning: A survey of case studies. ACM Computing Surveys, 55(6). https://doi.org/10.1145/3533378

[5] Lewis, P., et al. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In Advances in Neural Information Processing Systems (Vol. 33, pp. 9459–9474). https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[6] Chen, J., Zhang, X., & Han, X. (2023). Explainable identification of tax-related financial reporting anomalies in U.S. public companies: Cross-fitted tax-consistency residuals and Owen-value Shapley propagation. World Journal of Economics and Business Research, 1(1), 33–45.

[7] Zhang, F., et al. (2023). RepoCoder: Repository-level code completion through iterative retrieval and generation. In Proceedings of EMNLP (pp. 2471–2484). https://doi.org/10.18653/v1/2023.emnlp-main.151

[8] Zhou, S., Alon, U., Xu, F. F., Wang, Z., Jiang, Z., & Neubig, G. (2023). DocPrompting: Generating code by retrieving the docs. In Proceedings of the International Conference on Learning Representations (ICLR). shturl.cc/nCJPRmwXG7DDrhA0eQTWT0odvp9BSm

[9] Wang, J., Ge, Y., Liu, J., & Jiang, B. (2024). Early warning of liquidity stress in U.S. banks with seasonally aligned frailty hazards and delayed-feedback conformal alarms. Journal of Trends in Finance and Economics, 1(1), 85–100.

[10] Wang, Y., Le, H., Gotmare, A., Bui, N., Li, J., & Hoi, S. (2023). CodeT5+: Open code large language models for code understanding and generation. In Proceedings of EMNLP (pp. 1069–1088). https://doi.org/10.18653/v1/2023.emnlp-main.68

[11] 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. In Advances in Neural Information Processing Systems (Vol. 35). https://doi.org/10.52202/068431-1549

[12] Asai, A., Wu, Z., Wang, Y., Sil, A., & Hajishirzi, H. (2024). Self-RAG: Learning to retrieve, generate, and critique through self-reflection. In Proceedings of the International Conference on Learning Representations (ICLR). https://proceedings.iclr.cc/paper_files/paper/2024/hash/25f7be9694d7b32d5cc670927b8091e1-Abstract-Conference.html

[13] Schelter, S., Lange, D., Schmidt, P., Celikel, M., Biessmann, F., & Grafberger, A. (2018). Automating large-scale data quality verification. Proceedings of the VLDB Endowment, 11(12), 1781–1794. https://doi.org/10.14778/3229863.3229867

[14] Kuchnik, M., Klimovic, A., Simsa, J., Smith, V., & Amvrosiadis, G. (2022). Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines. Proceedings of Machine Learning and Systems, 4. https://proceedings.mlsys.org/paper_files/paper/2022/hash/d0e90e9a9310570dfa643aa3b2da6e89-Abstract.html

[15] Husain, H., Wu, H.-H., Gazit, T., Allamanis, M., & Brockschmidt, M. (2019). CodeSearchNet challenge: Evaluating the state of semantic code search (Tech. Rep. MSR-TR-2019-45). Microsoft Research. https://arxiv.org/abs/1909.09436

[16] Robertson, S., & Zaragoza, H. (2009). The probabilistic relevance framework: BM25 and beyond. Foundations and Trends in Information Retrieval, 4(1–2), 1–174. https://doi.org/10.1561/1500000019

[17] Zhou, F. (2026). Moment-constrained side-force allocation for wind-assisted merchant ships: Hydrodynamic performance of retractable anti-leeway fins and flap rudders. Innovation and Technology Studies, 3(1), 27–41.

[18] Karpukhin, V., et al. (2020). Dense passage retrieval for open-domain question answering. In Proceedings of EMNLP (pp. 6769–6781). https://doi.org/10.18653/v1/2020.emnlp-main.550

[19] 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

[20] Friedman, J. H. (2001). Greedy function approximation: A gradient boosting machine. Annals of Statistics, 29(5), 1189–1232. https://doi.org/10.1214/aos/1013203451

[21] Vovk, V., Gammerman, A., & Shafer, G. (2005). Algorithmic learning in a random world. Springer. https://doi.org/10.1007/b106715

[22] Angelopoulos, A. N., & Bates, S. (2023). Conformal prediction: A gentle introduction. Foundations and Trends in Machine Learning, 16(4), 494–591. https://doi.org/10.1561/2200000101

[23] Geifman, Y., & El-Yaniv, R. (2017). Selective classification for deep neural networks. In Advances in Neural Information Processing Systems (Vol. 30). https://proceedings.neurips.cc/paper/2017/hash/4a8423d5e91fda00bb7e46540e2b0cf1-Abstract.html

[24] Niculescu-Mizil, A., & Caruana, R. (2005). Predicting good probabilities with supervised learning. In Proceedings of the International Conference on Machine Learning (pp. 625–632). https://doi.org/10.1145/1102351.1102430

[25] Teng, D. (2025). TRIAD: Deadline- and reliability-aware scheduling for post-earthquake structural assessment workflows. World Journal of Engineering Research, 3(6), 55–66.

[26] Pedregosa, F., et al. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825–2830. https://www.jmlr.org/papers/v12/pedregosa11a.html

[27] Zhao, W., Chen, T., Wang, B., & Shen, Z. (2026). PIPE-RAG: An Execution-Guided Retrieval and Reuse Framework for Reliable Machine Learning Pipeline Code Generation. WEP Transactions on Computer Science and Intelligent Systems Research, 13, 286–297.

[28] Shang, W., Yang, J.-S., Ding, G., & Yang, S. (2026). ArchMigrate-Swift: A Compiler-in-the-Loop Retrieval-Augmented Multi-Agent Framework for Objective-C-to-Swift Migration. WEP Transactions on Computer Science and Intelligent Systems Research, 13, 275–285.