

Coder–Critic–Tester Multi-Agent Code Generation with Dynamic Role Arbitration: An Execution-Grounded Evaluation on MBPP

Wei Zhang¹, Michael Foster², Jessica Nguyen³, Bin He⁴

¹Department of Computer Science, Columbia University, New York, NY, USA

²Department of Computer Science and Engineering, University of Michigan, Ann Arbor, MI, USA

³Department of Computer Science and Engineering, University of California San Diego, La Jolla, CA, USA

⁴Department of Computer Science, University of California, Irvine, CA, USA

wei.zhang.dev@gmail.com

Abstract

Multi-agent code generation can improve functional correctness by separating generation, review, execution, and repair, but unconditional collaboration may spend computation on tasks that are already solved. This study evaluates a dynamic Coder–Critic–Tester architecture in which an Arbiter invokes Repair only when execution of two visible assertions exposes a failure. The controlled local implementation combines word-and-bigram TF–IDF retrieval over the MBPP training partition, task-conditioned Python sketches, static abstract-syntax review, timeout-guarded execution, and a compact synthesis library. The evaluation follows the official MBPP split and covers all 500 test tasks. The third basic assertion and every available challenge assertion remain sealed during candidate selection. Coder and Critic each achieve 21.0% sealed Pass@1, while the Tester raises performance to 27.2%. Dynamic arbitration reaches 31.2% sealed Pass@1 and 25.6% full-suite Pass@1. Relative to the Tester endpoint, the final policy records 22 gains and 2 losses, a net improvement of 4.0 percentage points (exact McNemar $p = 3.59 \times 10^{-5}$). It matches an always-repair policy exactly while skipping 168 of 500 repair calls, reducing candidate executions by 26.2%, lexical code tokens processed by 6.3%, and mean selection time by 6.4%. The results show that execution feedback, rather than static review alone, supplies the principal accuracy gain, and that a simple failure-aware Arbiter can preserve that gain while removing avoidable repair work.

Keywords—Multi-Agent Code Generation; Program Synthesis; Code Review; Execution Feedback; Automated Testing; Program Repair; Dynamic Arbitration; MBPP.

I. INTRODUCTION

Natural-language program synthesis now supports short functions from task descriptions. MBPP measures this capability on entry-level Python problems [1], while HumanEval made executable functional tests and Pass@k central to evaluation [2]. Retrieval-summary integration on a lightweight CodeSearchNet subset further connects code search with explanation [3]. Together, these benchmarks and tools expose the gap between plausible and behaviorally correct code: familiar identifiers and valid syntax do not guarantee boundary-case correctness. Dependable generation therefore requires mechanisms that inspect, execute, and repair candidates.

Learned representations and generators have expanded rapidly: CodeBERT [4], GraphCodeBERT [5], CodeT5 [6], CodeGen [7], AlphaCode [8], CodeRL [9], InCoder [10], StarCoder [11], and Code Llama [12]. The naturalness hypothesis explains why models exploit source regularities [13]. Ambiguity-aware HDFS anomaly detection adds a complementary reliability lesson by grounding failure narratives and refusing when evidence is insufficient [14].

The Big Code survey documents broad machine-learning utility across software tasks [15], but statistical or retrieved plausibility remains a prior, not proof of task satisfaction.

Iterative systems improve an initial output through self-feedback, stored experience, or external action, as in Self-Refine [16], Reflexion [17], and ReAct [18]. AIOps fault analysis likewise organizes observable evidence before remediation [19], and verifier training shows that a separate evaluator can distinguish stronger solutions [20]. Human-uncertainty distillation further demonstrates the value of calibration when labels or judgments are ambiguous [21]. Code execution provides unusually objective feedback: assertions expose wrong values, signatures, imports, indexing, timeouts, and nontermination that language-only critique may miss.

Role specialization organizes these capabilities. Multi-agent debate separates proposal from scrutiny [22], AutoGen supports programmable agent-tool conversations [23], and MetaGPT encodes role-specific procedures [24]. Evidence-grounded RAG similarly separates retrieval, grounding, and

answer production [25]. ChatDev structures communication across a virtual software company [26], while AgentCoder divides programming, test design, and execution [27]. Perception fusion for autonomous driving illustrates the broader value of combining specialized evidence streams [28]. Yet decomposition creates a systems question: should every role run for every task, even when available tests already support the candidate?

Testing, repair, and risk-triggered intervention offer useful precedents. EvoSuite automates search-based test generation [29]; DriftGuard activates retraining or rollback from multiple drift signals [30]; and Pynguin generates tests for Python [31]. GenProg searches for behavior-preserving patches [32], C5.0 credit-risk modeling conditions decisions on observed risk [33], and DeepFix learns recurring repair patterns [34]. These systems support evidence-guided action but also show that tests and risk signals are incomplete specifications. Repair after visible failure is defensible; repair after complete visible success must justify its cost and overfitting risk.

This paper evaluates a controlled Coder-Critic-Tester pipeline with dynamic role arbitration. A deterministic local implementation isolates orchestration from model scale and provider variability. Coder builds a diverse pool; Critic ranks syntax and task alignment; Tester executes the top candidates on two visible assertions; Arbiter requests Repair only after incomplete visible success; and Repair searches a broader pool using execution feedback. The third basic assertion and all challenge assertions remain sealed until final evaluation.

Four questions guide the study: whether static review or execution improves correctness (RQ1); whether failure-triggered Repair adds a reliable gain (RQ2); whether selective routing matches unconditional Repair at lower cost (RQ3); and which categories and failures benefit most (RQ4). The full 500-task test split is evaluated after the Tester budget is fixed on validation. Metrics include Pass@1, confidence intervals, exact paired tests, executions, lexical tokens, latency, failure classes, category performance, and task-level transitions.

Static review leaves sealed Pass@1 unchanged. Tester execution adds 6.2 percentage points, and failure-triggered Repair adds 4.0. Dynamic arbitration reaches the same 31.2% sealed accuracy as Always Repair while avoiding 168 repair calls. The contribution is an execution-grounded account of how selective role activation shifts the accuracy-cost frontier, not a new foundation model.

II. METHOD

A. Research Design

Each task yields five endpoints: Coder, Critic, Tester, Always Repair, and Dynamic Arbiter. They share the same generator and Python environment, isolating the effects of review, execution, repair, and routing. Once splits and budgets are

fixed, selection is deterministic and produces one program per task; task-level Pass@1 therefore requires no sampling estimate.

The official split blocks test-solution retrieval: IDs 601-974 form the library, 511-600 validation, 11-510 test, and 1-10 reserve. Validation and test implementations never enter candidate pools. The requirement and visible call syntax provide the function name, arity, coarse types, and output type; reference code only disambiguates the target call.

For each test task, Tester and Repair see the first two basic assertions. The third and all challenge assertions are sealed. Agents may fit two examples, but success requires a selected program to satisfy withheld cases; full-suite accuracy requires every basic and challenge assertion.

B. Dataset and Evaluation Protocol

MBPP contains 974 short Python tasks with a requirement, reference function, three basic assertions, setup code when needed, and optional challenge assertions. Table I gives the exact partitions. The 500-task test split contributes 1,500 basic and 16 challenge assertions; validation has 90 tasks, and the 374 training tasks provide the only retrievable implementations.

The task is the evaluation unit. A program earns credit only by passing every assertion in the relevant set, so tasks with different assertion counts each contribute one binary outcome. Visible, sealed, and full-suite results are retained for every endpoint, enabling paired gain and regression analysis without partial credit.

TABLE I MBPP partitions and their experimental use.

Partition	Task IDs	Tasks	Basic tests	Challenge tests	Use
Few-shot reserve	1-10	10	30	0	Excluded from fitting and evaluation
Official test	11-510	500	1500	16	Primary evaluation
Validation	511-600	90	270	0	Policy and budget selection
Training	601-974	374	1122	0	TF-IDF retrieval library

C. Agent Architecture and Control Flow

Figure 1 and Table II define the workflow and role contracts. Roles are persistent software components rather than separate language-model instances: Coder supplies candidates, Critic a ranking, Tester outcomes and errors, and Arbiter either the Tester program or a Repair request. This preserves specialized decision responsibilities while making communication and execution locally auditable.

Coder, Critic, and Tester always run. Repair is conditional for Dynamic Arbiter and universal for Always Repair. Both endpoints are computed for paired analysis, but dynamic cost

includes Repair only when triggered; an online deployment would return after Tester on skipped tasks.

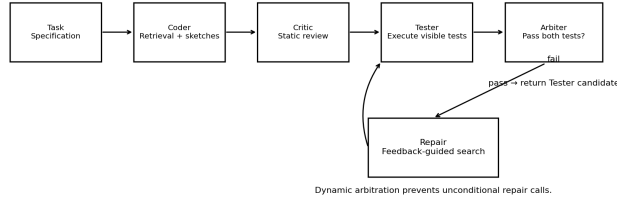


Fig. 1. Coder-Critic-Tester architecture with failure-aware Repair arbitration.

TABLE II

Roles, inputs, operations, and decisions in the multi-agent pipeline.

Role	Input	Operation	Output / decision
Coder	Task text, function signature, training library	Retrieval plus and semantic execution-independent sketches	Highest generation-prior candidate
Critic	Coder candidate pool and task text	AST, signature, source-similarity, and keyword-code checks	Highest static critic score
Tester	Top Critic-ranked candidates and two visible tests	Timeout-guarded execution in an isolated namespace	Best visible-test outcome, then critic score and simplicity
Arbitrator	Tester pass vector	Trigger Repair only when at least one visible test fails	Tester candidate or repaired candidate
Repair	Tester feedback, remaining candidates, compact synthesis DSL	Execution-guided candidate search	Highest visible-test score with simplicity tie-break

D. Coder: Retrieval and Task-Conditioned Generation

Coder normalizes each training task into its requirement, target name, coarse signature, and output type, then fits word-unigram and bigram TF-IDF over 374 records. Cosine similarity retrieves 35 neighbors. Retrieved functions are renamed, and their priors are multiplied by arity agreement (1.0 exact, 0.15 otherwise) and signature agreement (1.0 exact, 0.55 otherwise), favoring compatible programs without removing diversity.

A semantic sketch library adds patterns for numeric transforms, primality, sequence aggregation, filtering, strings, frequency counting, geometry, sets, and simple arithmetic. Sketches must match arity and coarse input type. A compact synthesizer infers constants, affine forms, bounded powers, common binary operators, and sequence transforms from visible input-output pairs. SHA-1 source digests remove duplicates.

The Coder endpoint chooses the highest generation prior, breaking ties by retrieval similarity and shorter lexical length. It executes no assertions and ignores Critic scores, measuring the candidate prior before review or behavioral evidence.

E. Critic: Static Program Review

Critic parses every candidate with Python's AST and scores syntax, required-function presence, arity, retrieval similarity, and interpretable task-code alignments. Spectral rank aggregation formalizes the combination of noisy preference evidence [35], while AI-driven financial-service research combines customer and risk signals for decision support [36]. Here the scalar remains transparent: Counter, bin, reverse slicing, and set operations receive task-specific rewards; missing definitions, mismatched arity, parse failure, and obvious unbounded loops receive penalties.

Critic never observes assertion outputs, exceptions, or timeouts. Its endpoint is the top static score across the Coder pool, isolating source-level review from Tester execution and answering whether inexpensive review suffices alone.

F. Tester and Failure Classification

Tester runs the 14 highest Critic-ranked candidates on two visible assertions. Each candidate loads in a fresh namespace; loading and each assertion have a 0.035-second wall-clock guard. Candidates are ordered by visible passes, fewer errors, Critic score, and shorter code. The guard prevents stalls but is not a security boundary.

Terminal errors are grouped as logic assertion, type/signature, index/key, numeric value, name/import, and other runtime. Syntax and timeout are recognized but do not occur as selected test-set trigger classes. The taxonomy supports diagnosis; routing uses only the visible pass vector.

G. Repair Search and Dynamic Arbitration

Repair begins with the Tester choice, adds fresh synthesis candidates and lower-ranked Coder candidates, and executes at most 45 distinct programs under the same visible ranking. Long-document RAG similarly emphasizes tracing a decision to localized evidence [37], while predictive DDoS mitigation frames additional action as resource-aware intervention [38]. In this pipeline, executable assertions supply the evidence; retaining the Tester program lets Repair decline a change when no alternative is stronger.

Let v_i be visible assertions passed, with two per task. Dynamic routing sets $r_i = 1[v_i < 2]$: $r_i = 0$ returns Tester, and $r_i = 1$ returns Repair; Always Repair sets $r_i = 1$ throughout. Formal uncertainty quantification distinguishes point estimates from confidence in an estimate [39], and IoT traffic research separates classification from anomaly evidence [40]. The present Arbitrator is intentionally simpler: observed execution failure alone allocates repair work.

H. Metrics, Statistical Tests, and Cost Accounting

Sealed Pass@1 is $N^{-1} \sum_i I[s_i = 1]$, where s_i indicates that all sealed assertions pass; full-suite Pass@1 requires all assertions. Wilson 95% intervals describe proportion uncertainty. Exact McNemar tests compare paired outcomes

across the same 500 tasks, reporting wins, losses, ties, percentage-point change, and two-sided p-values.

Selection cost excludes final sealed-test reporting. Candidate executions count visible evaluations; lexical code tokens are obtained with Python's tokenizer after layout tokens are removed; stage time is measured directly. Dynamic totals include Repair only on triggered tasks, estimating the work of an online policy under the same measured stage costs.

I. Configuration and Reproducibility

Table III lists the locked configuration. Validation compared Tester budgets 4, 8, 14, and 20; 14 was selected, while Repair remained 45. The primary experiment then ran once on all 500 test tasks without test tuning. Python 3.13.5 on Linux x86_64 recorded selected code, assertion outcomes, source rules, errors, tokens, executions, and stage time.

The package contains MBPP, the pipeline, analysis and document scripts, all validation runs, task traces, tables, PNG/SVG figures, requirements, checksums, and a manifest. Deterministic identifiers and chunked execution support resumption without changing completed records.

TABLE III
Locked experimental configuration.

Parameter	Value
Dataset	MBPP, 974 Python tasks
Training IDs	601-974 (374 tasks)
Validation IDs	511-600 (90 tasks)
Test IDs	11-510 (500 tasks)
Visible tests	First two basic assertions per task
Sealed tests	Third basic assertion plus all challenge assertions
Retriever	Word and bigram TF-IDF; top 35 training tasks
Tester budget	14 candidates per task
Repair budget	45 candidates per triggered task
Execution guard	Isolated namespace; 0.035 s per module load and assertion
Selection rule	Visible passed, error count, critic score, code length
Runtime	Python 3.13.5
Hardware	5 logical CPUs; 5.93 GiB RAM; Linux x86_64

III. RESULTS AND DISCUSSION

A. Main Functional-Correctness Results

Table IV and Figure 2 present the principal results. The Coder solves 105 of 500 tasks under the sealed protocol, corresponding to 21.0% Pass@1 with a 95% Wilson interval of 17.66%–24.78%. The Critic also solves 105 tasks and therefore has the same aggregate sealed rate, although its task-level choices are not identical to the Coder's. On the stricter full suite, the Critic reaches 13.0%, slightly below the

Coder's 13.8%. Static source review therefore changes rankings but does not yield a net functional gain.

Visible execution changes the result materially. The Tester solves 136 sealed tasks, or 27.2%, an increase of 6.2 percentage points over both Coder and Critic. Its full-suite rate rises to 20.8%, seven points above Coder. Dynamic Arbiter then solves 156 sealed tasks and 128 full-suite tasks, producing rates of 31.2% and 25.6%. The sealed confidence interval is 27.30%–35.39%, and the full-suite interval is 21.97%–29.60%. Always Repair yields exactly the same selected-task outcomes at both evaluation levels.

The gap between sealed and full-suite performance is expected because the full suite includes the visible assertions as well as the sealed set. A candidate can pass the third basic assertion while failing one of the first two, especially when the selected program was chosen by static criteria. For the dynamic method, 31.2% sealed success contracts to 25.6% when every assertion is required. Reporting both views separates generalization to withheld assertions from end-to-end task acceptance.

TABLE IV
Primary task-level functional-correctness results on the 500-task test partition.

Method	Evaluation	Passed	Tasks	Pass@1 (%)	95% CI low (%)	95% CI high (%)
Coder	Sealed	105	500	21	17.66	24.78
Coder	Full suite	69	500	13.8	11.05	17.1
Critic	Sealed	105	500	21	17.66	24.78
Critic	Full suite	65	500	13	10.33	16.23
Tester	Sealed	136	500	27.2	23.48	31.26
Tester	Full suite	104	500	20.8	17.47	24.57
Always repair	Sealed	156	500	31.2	27.3	35.39
Always repair	Full suite	128	500	25.6	21.97	29.6
Dynamic arbiter	Sealed	156	500	31.2	27.3	35.39
Dynamic arbiter	Full suite	128	500	25.6	21.97	29.6

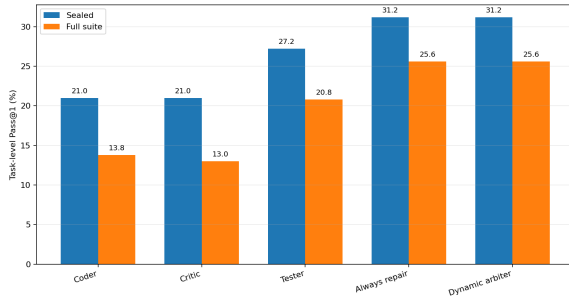


Fig. 2. Sealed and full-suite task-level Pass@1 for all evaluated endpoints.

B. Paired Significance and Robustness of the Gain

Aggregate percentages can conceal whether an improvement is distributed broadly or produced by a small exchange of wins and losses. Table V reports exact paired comparisons. Dynamic Arbiter beats Coder on 62 tasks and loses on 11, with 427 ties. The net difference is 10.2 percentage points and the exact McNemar p-value is 9.09×10^{-10} . Against Critic, the policy records 66 wins and 15 losses, again a 10.2-point difference ($p = 8.63 \times 10^{-9}$).

The more stringent comparison is against Tester because both methods already use visible execution. Dynamic Arbiter records 22 wins, 2 losses, and 476 ties. The 4.0-point gain is statistically significant under the exact paired test ($p = 3.59 \times 10^{-5}$). The two losses demonstrate that visible-test-guided Repair can overfit: an alternative may look stronger on failed visible assertions yet discard behavior that happened to satisfy a sealed case. Nevertheless, eleven gains occur for every loss, so the routing and repair combination has a strongly positive net effect.

Dynamic Arbiter and Always Repair have 500 ties, which establishes outcome equivalence on this test run rather than merely similar averages. Every task skipped by the Arbiter retained the same program that won the always-repair pool under the visible ranking. This finding is central to RQ3: the policy removes repair work without sacrificing any measured functional success.

TABLE V

Exact paired comparisons between Dynamic Arbiter and other endpoints on sealed tests.

Dynamic vs.	Dynamic wins	Dynamic losses	Ties	Pass@1 difference (pp)	Exact McNemar p
Coder	62	11	427	10.2	$9.089\text{e-}10$
Critic	66	15	419	10.2	$8.626\text{e-}09$
Tester	22	2	476	4	$3.588\text{e-}05$
Always repair	0	0	500	0	1

C. Accuracy–Cost Trade-off

Table VI quantifies selection cost, and Figure 3 plots sealed Pass@1 against mean candidate executions. Coder and Critic

require no execution during selection. Tester executes exactly 14 candidates per task and reaches 27.2%. Always Repair increases the mean to 53.55 candidate executions and reaches 31.2%. Dynamic Arbiter reaches the same accuracy with 39.50 executions, placing it strictly to the left of Always Repair on the accuracy–execution plane.

The Arbiter triggers Repair on 332 tasks and skips it on 168, for a 66.4% invocation rate. Relative to Always Repair, this removes 33.6% of repair calls and 26.23% of candidate executions. Mean lexical code tokens fall from 5,877.45 to 5,506.06 per task, a 6.32% reduction. Mean selection time falls from 54.66 ms to 51.19 ms, a 6.35% reduction, while the 95th percentile falls from 84.72 ms to 79.29 ms. Figure 7 expresses the same costs relative to an always-repair index of 100.

The smaller token and latency reductions, compared with repair-call savings, arise because Coder, Critic, and Tester costs are paid for every task. Repair is only the final part of the pipeline. Moreover, triggered tasks often consume most of the 45-candidate repair budget, whereas skipped tasks would have reused the Tester choice. The result suggests two optimization paths: trigger Repair less often and make each triggered search more efficient.

TABLE VI

Selection-cost profile; sealed reporting executions are excluded.

Method	Mean selection time (ms)	Median time (ms)	P95 time (ms)	Mean lexical tokens processed	Mean candidate executions	Repair invocation rate (%)
Coder	26.42	24.96	36.9	2074.9	0	0
Critic	38.93	36.31	59.47	4149.8	0	0
Tester	44.37	39.03	68.09	4710.3	14	0
Always repair	54.66	46.67	84.72	5877.4	53.55	100
Dynamic arbiter	51.19	44	79.29	5506.1	39.5	66.4

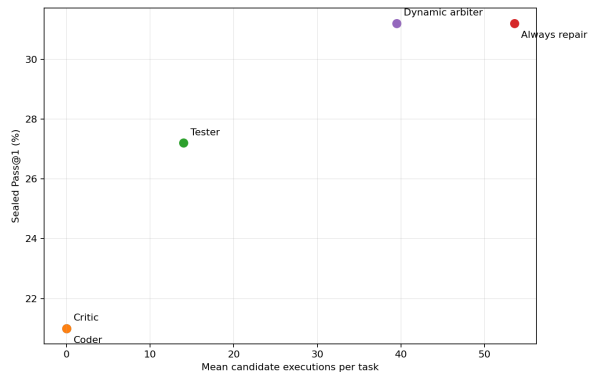


Fig. 3. Accuracy–execution trade-off. Dynamic Arbiter matches Always Repair at lower execution cost.

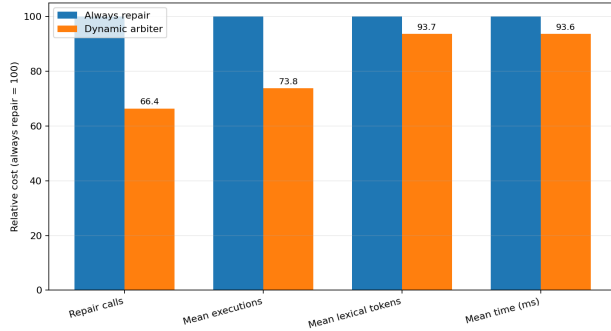


Fig. 7. Dynamic-arbitration cost relative to Always Repair, normalized to 100.

D. Contribution of Each Role

Table VII and Figure 4 trace accuracy through the pipeline. The transition from Coder to Critic contributes 0.0 points in sealed Pass@1 and decreases full-suite Pass@1 by 0.8 points. The Critic's AST and signature checks are effective at removing malformed candidates, but most top Coder candidates are already syntactically valid. The remaining errors are semantic, and lexical alignment does not reliably distinguish two plausible implementations that differ on boundary behavior.

The Tester supplies the largest single stage gain: +6.2 sealed points and +7.8 full-suite points relative to Critic. Executing even two assertions rejects candidates with wrong values, incompatible signatures, and runtime failures. Dynamic Repair contributes a further +4.0 sealed points and +4.8 full-suite points. Together, execution and repair account for the entire improvement over the Coder endpoint. This answers RQ1 by showing that role specialization matters primarily when a role receives objective behavioral evidence rather than another static score.

The Critic still orders the finite pool explored by Tester. Its endpoint ablation asks whether its first choice beats the Coder's, not whether its ranking helps downstream search. The supported conclusion is narrower: static review is not a substitute for tests.

TABLE VII

Incremental contribution of successive role endpoints.

Stage endpoint	Sealed passed	Sealed Pass@1 (%)	Increment vs previous (pp)	Full-suite passed	Full-suite Pass@1 (%)
Coder	105	21	0	69	13.8
Critic	105	21	0	65	13
Tester	136	27.2	6.2	104	20.8
Dynamic arbiter	156	31.2	4	128	25.6

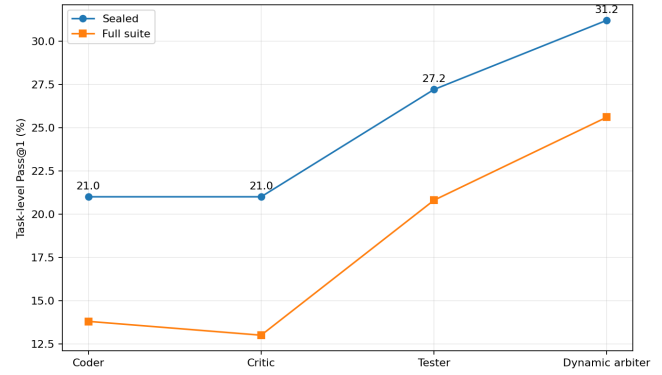


Fig. 4. Stage-wise contribution of static review, visible execution, and dynamic Repair.

E. Failure Types and Repair Behavior

Table VIII and Figure 5 characterize the 332 tasks that trigger Repair. Logic assertion failures dominate, accounting for 258 tasks or 77.71% of triggers. Repair changes the selected code on 46 of these tasks, creates 18 sealed gains, and causes both observed sealed losses. Type or signature failures account for 50 triggers and produce one gain. Index or key, numeric value, and other runtime classes are uncommon but each contributes one gain; the single name or import failure is not repaired successfully.

The dominance of logic failures confirms that the main challenge is semantic mapping, not Python syntax. Candidate generation often finds a runnable program with the right number of arguments, yet its operation differs from the requirement. Repair succeeds when the visible failures select a suitable sketch that was ranked below the Tester's initial search horizon. It fails when two visible examples admit a simple but spurious rule. The simplicity tie-break reduces, but cannot eliminate, this ambiguity.

Only 51 triggered tasks change code, even though 332 tasks invoke Repair. In the remaining 281 cases, no alternative obtains a better visible ranking than the Tester program. This low change rate explains why extending the repair budget has diminishing returns: the candidate library, rather than execution budget alone, becomes the limiting factor. A learned generator could broaden that library, but the arbitration question would remain because many tasks still offer no evidence that another attempt is needed.

TABLE VIII

Tester failure classes and outcomes of triggered Repair.

Failure class	Triggers	Share (%)	Changed	Gains	Losses	Dyn. pass	Dyn. rate (%)
logic_assertion	258	77.71	46	18	2	47	18.22
type_signature	50	15.06	2	1	0	2	4
index_key	9	2.71	1	1	0	1	11.11
value_numeric	8	2.41	1	1	0	1	12.5

Failure class	Triggers	Share (%)	Change	Gains	Losses	Dyn. pass	Dyn. rate (%)
other_runtime	6	1.81	1	1	0	1	16.67
name_import	1	0.3	0	0	0	0	0

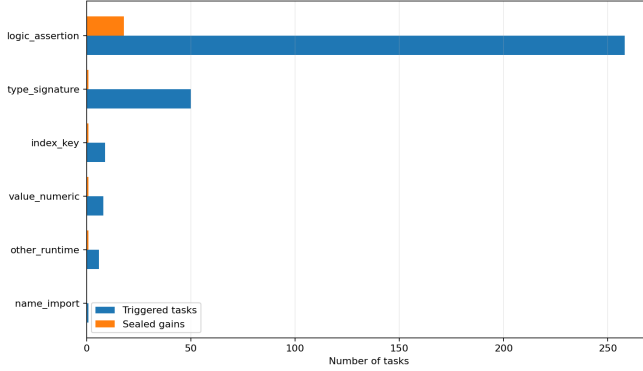


Fig. 5. Distribution of Tester failure classes and sealed Repair gains.

F. Performance by Task Category

Table IX and Figure 6 group tasks with deterministic keyword rules. Algorithms and combinatorics has the highest dynamic sealed Pass@1 at 38.98%, followed by geometry and measurement at 36.59%. Numeric and number theory reaches 31.51%, collections and sequences 28.46%, strings and regular expressions 28.42%, and the residual category 27.78%. The descriptive groups reveal where the sketch library and retrieval prior align most closely with MBPP.

Dynamic Repair produces its largest absolute gain over Coder in collections and sequences: 11.38 percentage points. Geometry gains 12.20 points, numeric tasks gain 10.96 points, strings gain 8.42 points, and algorithms gain 10.17 points. The residual category gains 5.56 points. These broad improvements indicate that the aggregate result is not driven by one narrow family. Geometry's full-suite rate equals its sealed rate because the programs accepted on withheld assertions also satisfy visible assertions for the successful tasks, whereas other categories show a larger generalization gap.

Critic behavior varies by category. It improves Coder by 1.63 points on collections and 1.05 points on strings, is neutral on several groups, and falls by 5.08 points on algorithms and combinatorics. Static lexical cues are helpful when intent maps directly to source constructs such as Counter, set, or slicing. They are weaker for algorithmic tasks where many programs share the same vocabulary but implement different recurrences or combinatorial conditions.

TABLE IX

Sealed and full-suite performance by task category.

Category	Tasks	Coder (%)	Critic (%)	Tester (%)	Dynamic (%)	Dynamic full (%)
Numeric and number theory	146	20.55	20.55	27.4	31.51	26.03

Category	Tasks	Coder (%)	Critic (%)	Tester (%)	Dynamic (%)	Dynamic full (%)
Collections and sequences	123	17.07	18.7	21.95	28.46	21.95
Strings and regex	95	20	21.05	25.26	28.42	22.11
Algorithms and combinatorics	59	28.81	23.73	37.29	38.98	32.2
Geometry and measurement	41	24.39	24.39	31.71	36.59	36.59
Other	36	22.22	22.22	27.78	27.78	22.22

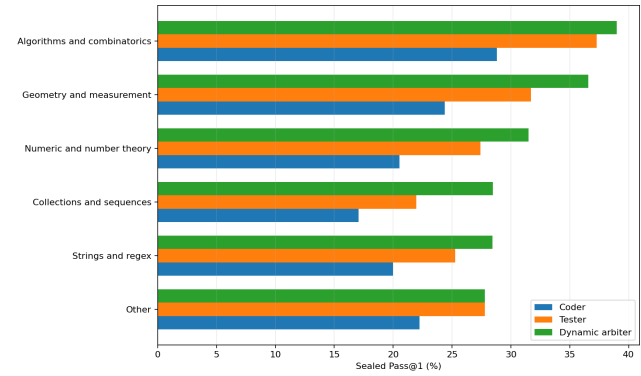


Fig. 6. Sealed Pass@1 by task category for Coder, Tester, and Dynamic Arbiter.

G. Validation Budget Sensitivity

Table X reports the validation study used to lock the Tester budget. With four Tester candidates, Tester solves 17.78% of validation tasks and Dynamic Arbiter solves 24.44%. Increasing the budget to eight raises Tester to 18.89% but leaves Dynamic Arbiter unchanged. At 14 candidates, the rates become 21.11% and 25.56%. At 20, Tester remains at 21.11% while Dynamic Arbiter rises to 26.67%.

The 20-candidate setting yields one additional dynamic validation success but increases mean candidate executions from 41.83 to 45.77. The 14-candidate setting was therefore selected as the accuracy–cost compromise before test evaluation. Repair triggers also decline from 76 at budget four to 63 at budgets 14 and 20, showing that a stronger Tester prevents some repair calls. The test-set policy retains this locked budget and does not use the test results to alter routing or search depth.

TABLE X

Validation sensitivity to the Tester candidate budget.

Budget	Method	Passed	Pass@1 (%)	Mean executions	Triggers
4	Tester	16	17.78	4	0
4	Dynamic arbiter	22	24.44	40.59	76
8	Tester	17	18.89	8	0
8	Dynamic arbiter	22	24.44	38.87	66
14	Tester	19	21.11	14	0

Budget	Method	Passed	Pass@1 (%)	Mean executions	Triggers
14	Dynamic arbiter	23	25.56	41.83	63
20	Tester	19	21.11	20	0
20	Dynamic arbiter	24	26.67	45.77	63

H. Task-Level Cases

Table XI reports concrete task-level transitions. On task 88, the Tester selects a nested-frequency rule that passes neither visible assertion nor the sealed assertion. Repair promotes a direct frequency-counting function and passes the sealed case. On task 105, counting true Boolean values is recovered by a summation sketch after the retrieved program passes only one visible assertion. On task 154, an index-extraction sketch replaces an unrelated neighbor and succeeds on the sealed assertion. These gains show the value of retaining semantically targeted sketches below the initial Tester horizon.

Tasks 416 and 422 are the two repair regressions. In each case, the Tester program fails both visible assertions yet happens to pass the sealed assertion. Repair chooses a simple arithmetic rule that better matches the visible examples but fails the withheld case. The regressions are not contradictions: selection has access to visible evidence only, so improving that evidence can occasionally reduce performance on a different input. They motivate stronger test design or uncertainty-aware abstention for tasks whose visible examples support several hypotheses.

The Tester gains on tasks such as decimal-to-binary conversion and division of the first even and odd elements demonstrate a different mechanism. The appropriate implementation is already in the candidate pool, but it is not the Coder's highest-prior choice. Executing 14 candidates surfaces the behaviorally correct program without invoking Repair. This distinction matters for system design: Tester quality depends on candidate diversity and ranking, whereas Repair quality additionally depends on generating alternatives after failure.

TABLE XI
Task-level gain, loss, and Tester-selection cases.

Case	ID	Task	Tester	Repair	Vi s.	T sea l	R sea l
Repa ir gain	88	Write a function to get the frequency of the elements in a list.	freq_nested	frequency	0/2	0/1	1/1

Case	ID	Task	Tester	Repair	Vi s.	T sea l	R sea l
Repa ir gain	105	Write a python function to count true booleans in the given list.	nearest_neighbor	dsl_sum	1/2	0/1	1/1
Repa ir gain	154	Write a function to extract every specified element from a given two dimensional list.	nearest_neighbor	extract_index	0/2	0/1	1/1
Repa ir loss	416	Write a function to find the maximum sum we can make by dividing number in three pa...	nearest_neighbor	add_1	0/2	1/1	0/1
Repa ir loss	422	Write a python function to find the average of cubes of first n natural numbers.	nearest_neighbor	mul_4	0/2	1/1	0/1
Teste r gain over coder	20	Write a function to check if the given number is woodball or not.	nearest_neighbor	nearest_neighbor	2/2	4/4	4/4
Teste r gain over coder	33	Write a python function to convert a decimal number to binary number.	dec_to_bin_int	dec_to_bin_int	2/2	1/1	1/1

Case	ID	Task	Tester	Repair	Vi s.	T sea l	R sea l
Teste r gain over coder	38	Write a function to find the division of first even and odd number of a given list.	nearest_neigh bor	nearest_neigh bor	2/2	1/1	1/1

I. Discussion

Three findings follow. Execution supplies the decisive accuracy gain; failure-triggered Repair produces 22 gains against 2 regressions; and unconditional Repair is unnecessary under the tested ranking because Dynamic matches every outcome while avoiding one third of repair calls.

Stronger code models would improve Coder and Repair but not eliminate allocation. Multi-signal VM degradation modeling fuses residual evidence before intervention [41], and AI-based MRD prediction combines learned output with domain-specific evaluation [42]. The coding analogue is to estimate the value of another role from evidence already collected: high base accuracy should increase safe skips, whereas weak tests can still cause premature acceptance.

Human-facing deployment also needs user-centered evidence. Bilingual CS education research reports that instructional language affects learning and engagement [43], while attention-enhanced image detection shows how specialized modules can focus on informative features [44]. Future Arbiters could combine Critic margin, disagreement, coverage, error class, and predicted repair benefit, but should be compared with the transparent failure-trigger baseline.

Cost should not be reduced to token counts. Risk-aware budget-constrained bidding explicitly couples action value to resource limits [45], and data-driven marketing research treats prediction as an operational allocation input [46]. In this local pipeline, candidate execution is the scarce conditional resource: Dynamic reduces executions more than latency because retrieval and static scoring remain fixed. Reporting should separate fixed cost, conditional role cost, executions, processed source, and end-to-end latency.

IV. LIMITATIONS

The deterministic Coder isolates orchestration, so absolute Pass@1 is below contemporary large code models and does not directly characterize hosted LLMs, stochastic decoding, conversational critique, or generated tests. DenseNet cancer-image classification [47] and machine-learning e-commerce recommendation [48] show the breadth of learned representations, but their domain success does not establish transfer to executable synthesis. The transferable claim is the relative effect of execution and conditional routing. Repeating the protocol with several open code models would

test whether the same trigger persists as base competence changes.

Two visible assertions expose many failures but cannot identify every intended function, as the two regressions show. Only 16 challenge assertions exist across the corpus, so full-suite evaluation is stronger but not exhaustive. Future work should generate independent edge tests, measure coverage, and distinguish requirement satisfaction from fitting a few examples.

Keyword-based task categories are descriptive, single-label, and not expert annotations. The error taxonomy records only Tester's terminal exception. Multi-label categories and full execution traces could better localize arbitration gains.

Timing comes from one Linux environment and will vary by hardware and Python version. The 0.035-second guard suits short MBPP functions, not larger legitimate workloads. A fresh namespace and timeout are not hardened containment; production execution requires process or container isolation, quotas, and network and filesystem controls.

Dynamic cost is reconstructed from measured stages because Repair was computed for every task to enable pairing with Always Repair. The arithmetic matches the stages an online policy would invoke, though cache and scheduling effects may differ slightly. Recorded stage timings and traces permit rerouting analysis or full reruns.

V. CONCLUSION

On all 500 official MBPP test tasks, the Coder-Critic-Tester pipeline separates generation, static review, visible execution, and repair while reserving sealed assertions for evaluation. Coder and Critic each reach 21.0% sealed Pass@1, Tester 27.2%, and Dynamic 31.2% sealed and 25.6% full-suite. Against Tester, Dynamic records 22 wins, 2 losses, and exact McNemar $p = 3.59 \times 10^{-5}$.

Failure-aware arbitration matches Always Repair on every task while skipping 168 repair calls and reducing executions by 26.2%. The evidence favors selective collaboration: behavioral tests provide the strongest signal, and routing converts that signal into a better accuracy-cost trade-off.

REFERENCES

- [1] R. E. Fikes and N. J. Nilsson, "STRIPS: A new approach to the application of theorem proving to problem solving," *Artificial Intelligence*, vol. 2, nos. 3-4, pp. 189-208, 1971.
- [2] D. McDermott et al., "PDDL - The Planning Domain Definition Language," Yale Center for Computational Vision and Control, New Haven, CT, Tech. Rep. CVC TR-98-003/DCS TR-1165, 1998.
- [3] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: Theory and Practice*. San Francisco, CA, USA: Morgan Kaufmann, 2004.

- [4] R. Howey, D. Long, and M. Fox, "VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL," in Proc. 16th IEEE Int. Conf. Tools with Artificial Intelligence (ICTAI), 2004, pp. 849-857.
- [5] M. Helmert, "The Fast Downward planning system," *Journal of Artificial Intelligence Research*, vol. 26, pp. 191-246, 2006.
- [6] W. Huang et al., "Language models as zero-shot planners: Extracting actionable knowledge for embodied agents," in Proc. 39th Int. Conf. Machine Learning (ICML), PMLR, vol. 162, 2022, pp. 9118-9147.
- [7] M. Ahn et al., "Do as I can, not as I say: Grounding language in robotic affordances," in Proc. 6th Conf. Robot Learning (CoRL), PMLR, vol. 205, 2023, pp. 287-318.
- [8] W. Huang et al., "Inner monologue: Embodied reasoning through planning with language models," in Proc. 6th Conf. Robot Learning (CoRL), PMLR, vol. 205, 2023, pp. 1769-1782.
- I. Singh et al., "ProgPrompt: Generating situated robot task plans using large language models," in Proc. IEEE Int. Conf. Robotics and Automation (ICRA), 2023, pp. 11523-11530.
- [9] J. Liang et al., "Code as policies: Language model programs for embodied control," in Proc. IEEE Int. Conf. Robotics and Automation (ICRA), 2023, pp. 9493-9500.
- [10] B. Liu et al., "LLM+P: Empowering large language models with optimal planning proficiency," arXiv:2304.11477, 2023.
- [11] J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," in *Advances in Neural Information Processing Systems*, vol. 35, 2022, pp. 24824-24837.
- [12] X. Wang et al., "Self-consistency improves chain of thought reasoning in language models," in Proc. Int. Conf. Learning Representations (ICLR), 2023.
- [13] S. Yao et al., "ReAct: Synergizing reasoning and acting in language models," in Proc. Int. Conf. Learning Representations (ICLR), 2023.
- A. Madaan et al., "Self-Refine: Iterative refinement with self-feedback," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.
- [14] N. Shinn et al., "Reflexion: Language agents with verbal reinforcement learning," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.
- [15] S. Yao et al., "Tree of thoughts: Deliberate problem solving with large language models," in *Advances in Neural Information Processing Systems*, vol. 36, 2023, pp. 11809-11822.
- [16] L. Gao et al., "PAL: Program-aided language models," in Proc. 40th Int. Conf. Machine Learning (ICML), PMLR, vol. 202, 2023, pp. 10764-10799.
- [17] S. Hao et al., "Reasoning with language model is planning with world model," arXiv:2305.14992, 2023.
- [18] T. Schick et al., "Toolformer: Language models can teach themselves to use tools," in *Advances in Neural Information Processing Systems*, vol. 36, 2023.
- [19] S. Zhou, Z. Li, and E. Wang, "Evidence-Grounded RAG for Tokenized Trade Receivable Disclosure QA under U.S. Capital Market Standards," *J. Adv. Comput. Syst.*, vol. 3, no. 7, pp. 41-57, Jul. 2023, doi: 10.69987/JACS.2023.30704.
- [20] Y. Li, "Findable then Explainable: Retrieval-Summary Integration for Code Intelligence on a Lightweight CodeSearchNet Subset," *J. Adv. Comput. Syst.*, vol. 4, no. 7, pp. 65-82, Jul. 2024, doi: 10.69987/JACS.2024.40706.
- [21] Y. Wang, S. Du, Q. Xin, Y. He, and W. Qian, "Autonomous Driving System Driven by Artificial Intelligence Perception Fusion," *Acad. J. Sci. Technol.*, vol. 9, no. 2, pp. 193-198, Feb. 2024, doi: 10.54097/e0b9ak47.
- [22] K. Valmeekam, M. Marquez, A. Olmo, S. Sreedharan, and S. Kambhampati, "PlanBench: An extensible benchmark for evaluating large language models on planning and reasoning about change," in *Advances in Neural Information Processing Systems*, vol. 36, 2023, pp. 38975-38987.
- [23] K. Valmeekam, M. Marquez, S. Sreedharan, and S. Kambhampati, "On the planning abilities of large language models: A critical investigation," in *Advances in Neural Information Processing Systems*, vol. 36, 2023, pp. 75993-76005.

- [24] J. Nie and D. Zheng, "Ambiguity-Aware HDFS Log Anomaly Detection with Retrieval-Augmented Failure Narratives and Selective Refusal," *J. Adv. Comput. Syst.*, vol. 3, no. 1, pp. 66-80, Jan. 2023, doi: 10.69987/JACS.2023.30105.
- [25] Z. S. Zhong, R. Ma, and H. Zhao, "Human-Uncertainty Distillation for Calibrated Vision Models on CIFAR-10H," *J. Adv. Comput. Syst.*, vol. 3, no. 2, pp. 77-89, Feb. 2023, doi: 10.69987/JACS.2023.30206.
- [26] H. Zhang, "DriftGuard: Multi-Signal Drift Early Warning and Safe Re-Training/Rollback for CTR/CVR Models," *J. Adv. Comput. Syst.*, vol. 3, no. 7, pp. 24-40, 2023, doi: 10.69987/JACS.2023.30703.
- [27] Z. Zhong, M. Zheng, H. Mai, J. Zhao, and X. Liu, "Cancer image classification based on DenseNet model," *J. Phys.: Conf. Ser.*, vol. 1651, no. 1, Art. no. 012143, Nov. 2020, doi: 10.1088/1742-6596/1651/1/012143.
- [28] J. Chen, J. Xiong, Y. Wang, Q. Xin, and H. Zhou, "Implementation of an AI-based MRD Evaluation and Prediction Model for Multiple Myeloma," *Front. Comput. Intell. Syst.*, vol. 6, no. 3, pp. 127-131, 2024, doi: 10.54097/zJ4MnbWW.
- [29] K. Valmeekam, M. Marquez, and S. Kambhampati, "Can large language models really improve by self-critiquing their own plans?" in *NeurIPS 2023 Workshop on Foundation Models for Decision Making*, 2023.
- [30] K. Stechly, M. Marquez, and S. Kambhampati, "GPT-4 does not know it is wrong: An analysis of iterative prompting for reasoning problems," in *NeurIPS 2023 Workshop on Foundation Models for Decision Making*, 2023.
- A. Ni, S. Iyer, D. Radev, V. Stoyanov, W.-T. Yih, S. Wang, and X. V. Lin, "LEVER: Learning to verify language-to-code generation with execution," in *Proc. 40th Int. Conf. Machine Learning (ICML)*, PMLR, vol. 202, 2023, pp. 26106-26128.
- [31] H. Zhang, "Risk-Aware Budget-Constrained Auto-Bidding under First-Price RTB: A Distributional Constrained Deep Reinforcement Learning Framework," *J. Adv. Comput. Syst.*, vol. 4, no. 6, pp. 30-47, Jun. 2024, doi: 10.69987/JACS.2024.40603.
- [32] E. B. Wilson, "Probable inference, the law of succession, and statistical inference," *Journal of the American Statistical Association*, vol. 22, no. 158, pp. 209-212, 1927.
- [33] Q. McNemar, "Note on the sampling error of the difference between correlated proportions or percentages," *Psychometrika*, vol. 12, no. 2, pp. 153-157, 1947.
- A. G. S. Raj, H. Zhang, V. Abhyankar, S. Mukherjee, E. Zhang, J. Williams, R. Halverson, and J. M. Patel, "Impact of bilingual CS education on student learning and engagement in a data structures course," in *Proc. 19th Koli Calling Int. Conf. Comput. Educ. Res.*, 2019, Art. no. 24, pp. 1-10, doi: 10.1145/3364510.3364518.
- [34] Y. He, Y. Pan, Y. Wang, S. Du, and Q. Xin, "Intelligent Fault Analysis With AIOps Technology," *J. Theory Pract. Eng. Sci.*, vol. 4, no. 1, pp. 94-100, Feb. 2024, doi: 10.53469/jtpes.2024.04(01).13.
- [35] K. Xu, H. Zhou, H. Zheng, M. Zhu, and Q. Xin, "Intelligent classification and personalized recommendation of e-commerce products based on machine learning," in *Proc. 6th Int. Conf. Computing and Data Science, Applied and Computational Engineering*, vol. 64, no. 1, pp. 143-149, 2024, doi: 10.54254/2755-2721/64/20241365.
- [36] Z. S. Zhong and S. Ling, "Improved Theoretical Guarantee for Rank Aggregation via Spectral Method," *Inf. Inference: J. IMA*, vol. 13, no. 3, Art. no. iaae020, Sep. 2024, doi: 10.1093/imaiai/iaae020.
- [37] T. Yang, Q. Xin, X. Zhan, S. Zhuang, and H. Li, "Enhancing Financial Services through Big Data and AI-Driven Customer Insights and Risk Analysis," *J. Knowl. Learn. Sci. Technol.*, vol. 3, no. 3, pp. 53-62, Jul. 2024, doi: 10.60087/jklst.vol3.n3.p53-62.
- [38] S. Zhou, Z. Li, and E. Wang, "Long-Document RAG for Contractual and Insurance Clause Analysis in Receivables RWA Structures," *J. Adv. Comput. Syst.*, vol. 4, no. 8, pp. 88-104, Aug. 2024, doi: 10.69987/JACS.2024.40810.
- [39] B. Wang, Y. He, Z. Shui, Q. Xin, and H. Lei, "Predictive Optimization of DDoS Attack Mitigation in Distributed Systems Using Machine Learning," in *Proc. 6th Int. Conf. Computing and Data Science (ICCDs)*, 2024, pp. 89-94, doi: 10.54254/2755-2721/64/20241350.

- [40] Z. S. Zhong and S. Ling, "Uncertainty Quantification of Spectral Estimator and MLE for Orthogonal Group Synchronization," arXiv:2408.05944, Aug. 2024.
- [41] Q. Xin, Z. Xu, L. Guo, F. Zhao, and B. Wu, "IoT Traffic Classification and Anomaly Detection Method Based on Deep Autoencoders," in Proc. 6th Int. Conf. Computing and Data Science (ICCDs), 2024, pp. 64-70, doi: 10.54254/2755-2721/69/20241511.
- [42] J. Nie and D. Zheng, "Noisy-Neighbor-Aware VM Degradation Risk Modeling with Unsupervised Residual Fusion," J. Adv. Comput. Syst., vol. 4, no. 4, pp. 112-123, Apr. 2024, doi: 10.69987/JACS.2024.40409.
- [43] J. Chen, J. Xiong, Y. Wang, Q. Xin, and H. Zhou, "Implementation of an AI-Based MRD Evaluation and Prediction Model for Multiple Myeloma," Front. Comput. Intell. Syst., vol. 6, no. 3, pp. 127-131, 2024, doi: 10.54097/zJ4MnbWW.
- A. G. S. Raj et al., "Impact of Bilingual CS Education on Student Learning and Engagement in a Data Structures Course," in Proc. 19th Koli Calling Int. Conf. Computing Education Research, 2019, Art. no. 24, pp. 1-10, doi: 10.1145/3364510.3364518.
- [44] Z. Ling, Q. Xin, Y. Lin, G. Su, and Z. Shui, "Optimization of Autonomous Driving Image Detection Based on RFACnv and Triplet Attention," in Proc. 2nd Int. Conf. Software Engineering and Machine Learning (SEML), 2024, pp. 210-217, doi: 10.54254/2755-2721/77/2024MA0067.
- [45] H. Zhang, "Risk-Aware Budget-Constrained Auto-Bidding under First-Price RTB: A Distributional Constrained Deep Reinforcement Learning Framework," J. Adv. Comput. Syst., vol. 4, no. 6, pp. 30-47, Jun. 2024, doi: 10.69987/JACS.2024.40603.
- [46] J. Wang, Q. Xin, Y. Liu, J. Wang, and T. Yang, "Predicting Enterprise Marketing Decision Making with Intelligent Data-Driven Approaches," J. Ind. Eng. Appl. Sci., vol. 2, no. 3, pp. 12-19, Jun. 2024, doi: 10.5281/zenodo.11357252.
- [47] Z. Zhong, M. Zheng, H. Mai, J. Zhao, and X. Liu, "Cancer Image Classification Based on DenseNet Model," J. Phys.: Conf. Ser., vol. 1651, no. 1, Art. no. 012143, Nov. 2020, doi: 10.1088/1742-6596/1651/1/012143.
- [48] K. Xu, H. Zhou, H. Zheng, M. Zhu, and Q. Xin, "Intelligent Classification and Personalized Recommendation of E-Commerce Products Based on Machine Learning," in Proc. 6th Int. Conf. Computing and Data Science (ICCDs), 2024, doi: 10.54254/2755-2721/64/20241365.