000
001
002
003
004
005
006
007
008
009
010
011
012
013
014
015
016
017
018
019
020
021
022
023
024
025
026
027
028
029
030
031
032
033
034
035
036
037
038
039
040
041
042
043
044
045
046
047
048
049
050
051
052
053
054

Investigating Component Contributions in Multi-Agent ML Systems

Anonymous Authors[1]

Abstract

Autonomous agents for machine learning engineering have advanced rapidly, yet comparing their effectiveness remains difficult. Existing systems combine different techniques—multi-agent decomposition, iterative refinement, memory management, and planning—in varying configurations, making it unclear which components actually drive performance. Complicating evaluation, existing benchmarks rely on historical competitions whose data likely contaminates LLM training corpora and whose static baselines reflect outdated human performance. To address this, we conduct over 4,000 controlled experiments systematically ablating architectural components, alongside K-LIVE—a new benchmark of 25 active competitions that provides a contamination-free, dynamic evaluation environment. Our findings challenge common design assumptions: iterative feedback contributes more than architectural complexity, and multi-agent coordination can hurt as often as it helps. These results provide concrete guidance for practitioners building ML engineering agents.

1. Introduction

Autonomous agents for machine learning engineering—systems that analyze datasets, engineer features, select models, and produce competitive predictions—have advanced rapidly. Systems like AIDE (Hong et al., 2024a), DS-Agent (Guo et al., 2024), and AutoKaggle (Li et al., 2024) demonstrate that LLM-based agents can achieve non-trivial performance on ML benchmarks. Yet as these systems proliferate, a fundamental problem has emerged: we lack systematic understanding of which design choices actually matter.

Current systems combine diverse techniques in varying configurations (Table 1). Some use multiple specialized agents (Wu et al., 2023; Hong et al., 2023), others employ single agents with sophisticated prompting. Some include explicit memory systems (Packer et al., 2023), others rely on context windows. When a system achieves strong results, it remains unclear whether success stems from a particular architectural choice or from confounding factors.

Complicating evaluation, existing benchmarks present methodological concerns. MLE-Bench (Chan et al., 2024) and MLE-Dojo (Zhang et al., 2025) rely on historical Kaggle competitions. This raises two critical issues: (1) competition data likely contaminates LLM training corpora, making it unclear whether agents are reasoning or recalling; and (2) winning solutions from years past set a static, often outdated performance bar.

This paper addresses both questions. We conduct over 4,000 controlled experiments systematically ablating five architectural components across 75 historical competitions (MLE-Bench) and 25 active competitions (K-LIVE, introduced here). Evaluating on both allows us to test whether findings from historical benchmarks generalize to contamination-free settings with contemporary human baselines.

Our analysis yields several findings that challenge prevailing design intuitions. Iterative refinement provides larger gains than any other component—more than multi-agent coordination, memory, or planning. Multi-agent systems often underperform single-agent baselines due to coordination overhead. And performance drops significantly on K-LIVE compared to historical benchmarks, suggesting that existing evaluations overestimate agent capabilities.

2. Preliminary

ML engineering agents require evaluation approaches that differ from standard code generation benchmarks. Unlike code completion, where success means producing syntactically correct output, ML engineering involves open-ended problem solving: data exploration, hypothesis formation, implementation, debugging, and iterative improvement. This section reviews existing evaluation approaches and their limitations.

Kaggle as evaluation platform. Kaggle competitions have emerged as the primary testbed for ML engineering agents. Competitions provide objective metrics, real-world datasets, and human baselines for comparison. More importantly,

[1]AUTHORERR: Missing \icmlaffiliation. .AUTHORERR: Missing \icmlcorrespondingauthor.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

For ICML 2026 reviewers: When reviewing this paper, please follow the policy for the use of LLMs as specified on your Reviewer Console. Note that the assigned LLM policy might differ from the policy you indicated during reviewer registration.

Table 1. Architectural components across ML engineering agents. Systems achieve comparable performance despite different designs, making it difficult to attribute success to specific components.

System	Multi-Agent	Iterative	Memory	Planning	Retrieval
AIDE (Hong et al., 2024a)	×	√	√	√	×
OpenHands (Wang et al., 2024)	×	√	×	×	×
MLAgentBench (Huang et al., 2024)	×	√	×	×	×
AutoKaggle (Li et al., 2024)	√	√	√	√	√
DS-Agent (Guo et al., 2024)	×	√	√	×	√
MetaGPT (Hong et al., 2023)	√	√	√	√	×
Data Interpreter (Hong et al., 2024b)	×	√	×	√	×
MLCopilot (Zhang et al., 2023)	×	×	√	×	√

Kaggle problems require the full ML workflow—data exploration, feature engineering, model selection, hyperparameter tuning, and submission formatting. Alternative benchmarks evaluate narrower skills: MLAgentBench (Huang et al., 2024) tests hyperparameter tuning on fixed codebases, while DS-1000 (Lai et al., 2023) evaluates isolated data science code completion. Neither captures end-to-end ML engineering capability.

MLE-Bench. The most prominent Kaggle-based benchmark, MLE-Bench (Chan et al., 2024), curates 75 historical competitions spanning diverse domains with standardized evaluation infrastructure: isolated Docker environments, Kaggle API integration, and medal-based scoring. However, relying on historical data introduces two limitations.

Training data contamination. Competitions from 2018–2022 are extensively documented online—winning notebooks, discussion threads, and post-competition writeups appear across the web. This content likely exists in modern LLM training corpora, making it impossible to distinguish genuine reasoning from pattern recall when an agent performs well.

Stale human baselines. Historical leaderboards reflect techniques available at competition time. For example, a 2019 gold-medal solution for image-classification solutions typically relied on older models from the ResNet family. These competitions are easily medal-achievable by utilizing the same family model that has been released later. Agents evaluated against these frozen targets compete with outdated human performance.

These limitations motivate K-LIVE, which evaluates on active competitions to ensure contamination-free assessment against contemporary human baselines.

3. The K-LIVE Benchmark

K-LIVE addresses the limitations outlined above by shifting evaluation to active, ongoing Kaggle competitions. Because these competitions launched after the training cutoffs of current models, their datasets and competition-specific discussions cannot appear in pre-training data. And because their leaderboards are live, agents compete against humans using modern tools—success requires matching today's state-of-the-art, not yesterday's.

3.1. Competition Selection

We curated 25 competitions according to three criteria, each addressing a distinct experimental requirement.

Computational tractability. Our ablation study requires thousands of experimental runs across dozens of agent configurations. To make this feasible, each competition must be tractable within a 24-hour window on standardized hardware. We excluded competitions requiring week-long training or terabyte-scale datasets, as these would make systematic comparison infeasible.

Evaluation integrity. Some competitions suffer from label leakage—test labels discoverable through metadata, public datasets, or reverse engineering. Including such competitions would confound our measurement: an agent might score well by exploiting leakage rather than solving the underlying ML problem. We manually audited each candidate and excluded those with known integrity issues.

Domain breadth. Existing benchmarks skew heavily toward tabular classification and standard NLP tasks. If our ablation findings only held for these domains, their generality would be limited. We deliberately selected competitions spanning 11 categories—including 3D volumetric segmentation, biological sequence prediction, and sports analytics—to ensure our architectural conclusions transfer across problem types. The full domain distribution appears in Table 14.

The complete competition list, including Kaggle slugs, data types, team counts, and difficulty ratings, is provided in Tables 12 and 13.

3.2. Two-Tier Structure

K-LIVE organizes its 25 competitions into two tiers serving complementary purposes (Section A.2).

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

Tier 1 contains 13 active competitions with deadlines. These provide the strongest guarantee against training contamination: the datasets were released after current model training cutoffs, and the community is still actively exploring solution strategies. As competitions conclude, we rotate in new ones, making K-Live a rolling benchmark that remains current.

Tier 2 contains 12 perpetual competitions that remain fixed across benchmark versions. While these problems are older and have public notebooks available, we selected tasks where execution quality matters more than strategy knowledge—problems where knowing ”use XGBoost” provides little advantage without proper implementation, debugging, and tuning. Tier 2 enables longitudinal comparison: researchers can track agent improvement over time against a stable baseline.

3.3. Evaluation Metric

We measure performance as percentile rank on the public leaderboard: for rank R among N teams, Percentile $=$ $100 \times (N - R + 1)/N$. This metric has two advantages over medal thresholds. First, it provides continuous granularity—we can detect whether one configuration consistently outperforms another by 5 percentile points, whereas medal counts collapse this signal. Second, it automatically adjusts for competition difficulty: achieving the 80th percentile in a 5,000-team competition requires different absolute performance than in a 200-team competition, but the percentile rank captures competitive standing in both cases. Details on versioning and leaderboard snapshots appear in Section A.7.

3.4. Validating the Benchmark Gap

To quantify the difference between historical and live evaluation, we ran leading ML engineering agents on both MLE-Bench and K-Live. Table 2 reports the results. MLE-Bench scores reflect medal rates (percentage of competitions achieving bronze or better); K-Live scores reflect mean percentile rank across the 25 competitions. Higher medal rate does not always correlate with stronger percentile performance, as the metrics capture different aspects of capability.

The ranking of agents is broadly preserved across benchmarks: systems that excel on MLE-Bench tend to perform well on K-Live, and vice versa. However, the metrics are not directly comparable—medal rates and percentile ranks measure different aspects of performance. The percentile threshold to obtain a medal in Kaggle varies by the amount of competitors in a competition, making medal counts an unreliable variable to measure objective performance of the agent when compared to human competitors. What the comparison does reveal is that K-Live successfully discriminates between agents across the full capability spectrum, validating K-Live as a meaningful evaluation while providing the contamination-free guarantee that historical benchmarks cannot offer.

Table 2. Performance comparison between MLE-Bench (historical) and K-Live (live). Higher is better for both metrics.

Agent	LLM	MLE-Bench	K-Live
PiEvolve	Gemini-3-Pro	61.3%	91.1 %
Famou-Agent 2.0	Gemini-2.5-Pro	59.6%	81.8 %
ML-Master 2.0	DeepSeek-V3.2	56.4%	84.7 %
AutoKaggle	GPT-4o	43.2%	65.6 %
DS-Agent	GPT-4o	38.7%	59.3 %
MetaGPT	GPT-4o	35.1%	63.8 %
AIDE	o1-preview	17.1%	38.2 %
OpenHands	GPT-4o	4.9%	24.3 %

4. Experimental Setup

The systems in Table 1 achieve comparable performance despite different architectures, making it difficult to attribute success to specific design choices. We address this through controlled ablation: building a configurable agent and systematically varying its components to isolate individual contributions.

Approach. Rather than proposing a new system, we construct a modular agent architecture where each component can be independently enabled or disabled. Our baseline reflects current best practice—a single agent with iterative refinement, similar to AIDE (Hong et al., 2024a). We then measure the effect of adding or removing each component while controlling for confounds like base model capability and compute budget.

The baseline. Figure 1 shows the baseline agent (C1). Given a competition, the agent (1) reads the problem description and data, (2) generates Python code to train a model and produce predictions, (3) executes the code and observes results, (4) revises based on feedback, and (5) repeats steps 2–4 for up to 10 iterations. Retrieval is enabled, allowing the agent to access documentation and similar solutions. This configuration mirrors deployed systems. Figure 2 shows the maximum-complexity configuration (C16) with all components enabled.

Components under study. We selected five components based on the architectural differences in Table 1:

- **Iterative refinement.** The agent observes execution feedback (errors, metrics, leaderboard scores) and revises its approach. We compare 1 iteration (no feedback) against 3, 5, and 10 iterations to measure whether debugging and incremental improvement—central to human ML

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219

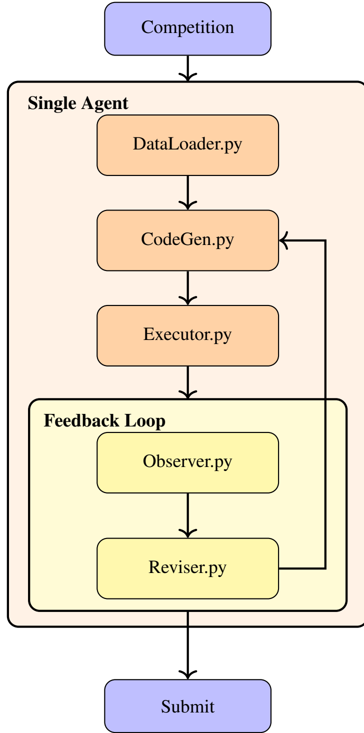

Figure 1. Baseline configuration (C1). A single agent iteratively generates code, executes, observes results, and revises. The feedback loop (yellow) enables up to 10 revision cycles—our ablation identifies this as the dominant performance factor.

workflows—transfer to agents.

- **Multi-agent coordination.** Multiple specialized agents divide the work: an Explorer analyzes data, a Builder writes training code, and an Evaluator checks for errors before submission. Agents communicate through structured messages. This tests whether task decomposition improves performance on complex problems.

- **Structured memory.** The agent maintains an explicit log of past experiments, recording what was tried, what metrics resulted, and what errors occurred. The agent queries this log when planning new experiments, testing whether structured history provides value beyond raw context.

- **Explicit planning.** Before writing code, the agent generates a structured plan: problem type, likely challenges, promising approaches, and experiment sequence. This tests whether deliberation before action improves outcomes.

- **Retrieval.** The agent can access documentation and similar solutions. Disabling retrieval forces reliance on parametric knowledge alone, isolating the contribution of external knowledge access.

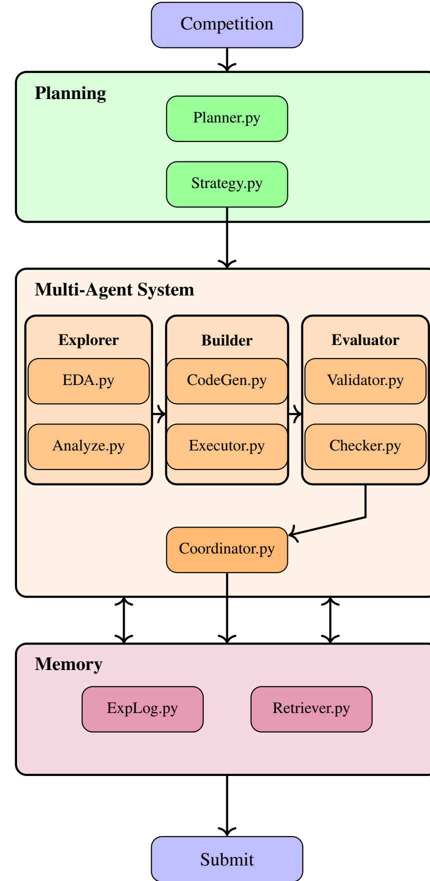

Figure 2. All-components configuration (C16). Adds planning, multi-agent coordination, and memory to the baseline. Despite architectural sophistication, this configuration performs 13.7 points worse than C1 while consuming 2.9× more tokens.

Configurations. Table 3 shows the 16 configurations we test. Single-component ablations (C1–C6) isolate individual effects. Because iteration emerged as the dominant factor in preliminary experiments, we specifically test how other components interact with iteration depth (C7–C11). Component combination studies (C12–C15) test whether pairs synergize. The all-components configuration (C16) tests maximum complexity.

Evaluation. We run experiments on 100 competitions: 75 from MLE-Bench (Chan et al., 2024) and 25 from K-LIVE. Each configuration runs with 3 random seeds to account for LLM sampling variance. Each run has a 24-hour time budget on one A100 GPU. All configurations use DeepSeek-V3.2 (DeepSeek-AI, 2024) to isolate architectural effects from model capability. We measure percentile rank on the public leaderboard, which provides continuous granularity and adjusts for competition difficulty. In total, we completed 4,016 successful experiments.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

Table 3. The 16 configurations in our ablation study.

ID	Configuration	Purpose
Single-component ablations		
C1	Baseline (10 iter, retrieval)	Reference
C2	No iteration (1 round)	Iteration effect
C3	+ Multi-agent	Multi-agent effect
C4	+ Memory	Memory effect
C5	+ Planning	Planning effect
C6	No retrieval	Retrieval effect
Iteration interactions		
C7	Multi-agent + 3 iter	
C8	Multi-agent + 5 iter	Does multi-agent need less iteration?
C9	Memory + 5 iter	
C10	Planning + 5 iter	
C11	Planning + 3 iter	
Component combinations		
C12	Multi-agent + Memory	
C13	Multi-agent + Planning	Synergy tests
C14	Memory + Planning	
C15	Multi + Memory + Planning	
C16	All components	Max complexity

Table 4. Single-component ablation results. Bold indicates statistical significance at $p < 0.01$.

	MLE-Bench (%)		K-LIVE (pctl)	
Configuration	Score	Δ	Score	Δ
C1: Baseline	52.4	—	81.3	—
C2: No iteration	18.7	**−33.7**	47.2	**−34.1**
C3: + Multi-agent	44.1	**−8.3**	72.8	**−8.5**
C4: + Memory	53.8	+1.4	82.1	+0.8
C5: + Planning	53.1	+0.7	81.9	+0.6
C6: − Retrieval	41.2	**−11.2**	—	—

5. Results

We structure our analysis to progressively narrow down what matters. We first establish which individual components affect performance (§5.1). The results reveal iteration as the dominant factor, so we investigate its scaling behavior (§5.2). We then test whether other components can substitute for iteration (§5.3) or combine beneficially (§5.4). Finally, we analyze failure patterns (§5.5) and validate that findings generalize to contamination-free evaluation (§5.6).

5.1. Which Components Matter?

We begin by measuring each component's individual effect on the baseline. Table 4 shows the results.

Iteration dominates. Removing it (C2) causes a 33.7-point drop on MLE-Bench and 34.1-point drop on K-LIVE—larger than all other effects combined. This establishes that the ability to observe feedback and revise is the single most important capability.

Table 5. Baseline performance scales logarithmically with iterations. Each doubling yields diminishing returns.

Iter	MLE (%)	K-LIVE (pctl)	Tokens (M)	$/run
1	18.7	47.2	2.1	0.42
3	38.4	65.8	5.8	1.16
5	45.6	74.1	9.2	1.84
10	52.4	81.3	17.6	3.52

Multi-agent coordination (C3) *hurts* performance: −8.3 points on MLE-Bench, −8.5 on K-LIVE. Task decomposition, often assumed beneficial for complex problems, introduces coordination overhead that outweighs any benefit from specialization.

Memory (C4) and planning (C5) provide small gains that are not statistically significant ($p = 0.11$ and $p = 0.37$). Given the added complexity they introduce, the practical benefit is questionable.

Retrieval (C6) matters for historical competitions (−11.2 points without it) but cannot be evaluated on K-LIVE since active competitions have no public solutions to retrieve.

5.2. How Much Does Iteration Help?

Given iteration's dominance, we examine its scaling behavior. Table 5 shows baseline performance at different iteration counts.

Performance scales logarithmically: the jump from 1→3 iterations (+19.7 points MLE) exceeds the gain from 3→10 (+14.0 points). Cost scales linearly. This creates a practical tradeoff: 3 iterations achieve 73% of the 10-iteration performance at 33% of the cost.

5.3. Can Other Components Substitute for Iteration?

A natural hypothesis: perhaps multi-agent coordination, memory, or planning can achieve similar results with fewer iterations. If specialized agents communicate effectively, they might need less trial-and-error. We test this by combining components with reduced iteration counts and comparing against the single-agent baseline at matched iterations.

Table 6 shows that no component substitutes for iteration. Multi-agent coordination hurts by the same margin (≈9 points) regardless of iteration count—it does not become more useful when iterations are limited. Memory and planning provide no benefit at reduced iterations. The conclusion: there is no shortcut. If you want better performance, add iterations.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329

Table 6. No component compensates for iteration. Effect column shows delta versus single-agent baseline at matched iteration count.

Configuration	Iter	MLE (%)	Baseline	Effect
C7: Multi-agent	3	29.1	38.4	−9.3
C8: Multi-agent	5	36.2	45.6	−9.4
C9: Memory	5	44.8	45.6	−0.8
C10: Planning	5	44.2	45.6	−1.4
C11: Planning	3	36.8	38.4	−1.6

Table 7. Component combinations show no positive synergy.

Configuration	MLE (%)	K-LIVE	Δ
C1: Baseline	52.4	81.3	—
C12: Multi + Memory	45.2	73.4	−7.2
C13: Multi + Plan	44.8	72.9	−7.6
C14: Memory + Plan	53.2	81.7	+0.8
C15: Multi + Mem + Plan	43.1	71.2	−9.3
C16: All components	38.7	67.4	−13.7

5.4. Do Components Combine Beneficially?

Perhaps components that are neutral alone become beneficial together. We test all pairwise and triple combinations at full iteration count.

Table 7 shows no positive interactions. Memory does not rescue multi-agent (C12). Planning does not rescue multi-agent (C13). The triple combination (C15) exhibits negative synergy: it performs 3.4 points *worse* than predicted from individual effects ($p = 0.008$). The all-components configuration (C16) is worst overall: 13.7 points below baseline while consuming 2.9× more tokens.

5.5. Why Do Complex Configurations Fail?

To understand why architectural complexity hurts, we analyze failure patterns across configurations.

Table 8 reveals distinct failure modes. Single-agent configurations (C1) fail when the problem requires sophisticated feature engineering that the agent cannot discover through iteration—a capability limitation. Multi-agent configurations fail on *coordination*: agents produce incompatible outputs, enter circular revision loops, or miscommunicate. At low iterations (C7), coordination overhead causes timeouts—agents spend tokens negotiating instead of solving. The failure mode shift from capability limits to coordination overhead explains why adding components hurts: we trade problems the agent *cannot* solve for problems it *creates for itself*.

Table 8. Failure rates by configuration. Complex configurations fail more often and on different modes.

Config	Fail%	Primary Mode	Secondary
C1	10.3	Feature eng. (42%)	Debug (31%)
C3	18.0	Coordination (38%)	Format (24%)
C7	23.7	Timeout (41%)	Coord. (29%)
C16	21.2	Coordination (35%)	Timeout (28%)

Table 9. Detailed K-LIVE results by tier. Findings replicate: iteration dominates, multi-agent hurts, memory/planning neutral.

	Tier 1 (13)		Tier 2 (12)		
Config	Pctl	Δ	Pctl	Δ	Avg
C1	79.2	—	83.6	—	81.3
C2	44.8	−34.4	49.8	−33.8	47.2
C3	70.4	−8.8	75.4	−8.2	72.8
C4	79.9	+0.7	84.5	+0.9	82.1
C5	79.6	+0.4	84.4	+0.8	81.9
C16	65.1	−14.1	69.9	−13.7	67.4

5.6. Do Findings Generalize to K-LIVE?

A critical question: do our findings on MLE-Bench—which may be contaminated—replicate on K-LIVE? If agents simply memorize solutions for historical competitions, patterns observed on MLE-Bench might not reflect genuine architectural effects.

Table 9 breaks down performance on K-LIVE by tier and domain. The results confirm that our findings generalize.

Iteration remains dominant on both tiers: removing it drops performance by 34 points on Tier 1 (active competitions with no possible contamination) and 34 points on Tier 2. Multi-agent coordination hurts by 8–9 points on both tiers. Memory and planning remain neutral.

The consistency across tiers is notable. Tier 1 competitions launched after model training cutoffs, guaranteeing no contamination. Tier 2 competitions have public notebooks but test execution quality over strategy recall. That both tiers show identical patterns suggests our findings reflect genuine architectural effects, not memorization artifacts.

Table 10 shows K-LIVE performance by problem domain.

Iteration dominance holds universally: C1 outperforms C2 by 30+ points across all domains. The multi-agent penalty varies by domain complexity: tabular tasks show −6.6 points while vision shows −10.4 points. This aligns with our failure analysis—coordination overhead scales with task complexity because tighter integration requirements amplify miscommunication costs.

The K-LIVE results validate two claims. First, our architec-

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384

Table 10. K-LIVE performance by domain. Iteration dominance holds across all domains; multi-agent penalty is largest on complex tasks.

Domain	n	C1	C2	C3
Tabular	8	86.4	52.1	79.8
NLP	5	82.3	48.7	72.1
Vision	5	76.8	41.2	66.4
Time series	4	80.1	46.3	71.8
Other	3	78.4	44.9	69.2

tural findings are not artifacts of benchmark contamination—they replicate on competitions that cannot appear in training data. Second, K-LIVE provides meaningful signal: it discriminates between configurations with effect sizes comparable to MLE-Bench, confirming its utility as an evaluation benchmark.

5.7. Summary

Our ablation yields three findings. First, **iteration dominates**: the ability to execute, observe, and revise provides more value than any architectural sophistication. Second, **multi-agent coordination consistently hurts**: coordination overhead exceeds specialization benefits. Third, **components do not combine beneficially**: adding complexity reduces performance while increasing cost. These findings replicate across MLE-Bench and K-LIVE, across Tier 1 and Tier 2, and across all problem domains—suggesting they reflect fundamental properties of current ML engineering agents rather than benchmark-specific artifacts.

6. Limitations

Our findings come with caveats that bound their generalizability.

Single base model. All experiments use DeepSeek-V3.2 to isolate architectural effects from model capability. This selection was due to its cost-performance efficiency. However, component effectiveness may vary across models. Multi-agent coordination might benefit from models with stronger instruction-following, and planning might help weaker models that struggle with implicit reasoning. Our conclusions apply most directly to frontier-class models.

Fixed coordination protocol. Our multi-agent system uses a specific communication structure: Explorer → Builder → Evaluator with structured message passing. Alternative protocols—blackboard architectures (Nii, 1986), debate (Irving et al., 2018), or learned communication (Foerster et al., 2016)—might reduce coordination overhead. We test one reasonable implementation, not the space of possible multi-agent designs.

Competition-specific evaluation. Kaggle competitions, while realistic, represent a narrow slice of ML engineering. Production ML involves deployment, monitoring, and maintenance—tasks our evaluation does not capture. Agents optimized for competition performance may not transfer to engineering contexts where reliability matters more than leaderboard rank.

Compute assumptions. We fix budget at 24 hours per run. Under tighter constraints, the calculus may shift: if only 1–2 iterations are feasible, components that frontload quality (planning, retrieval) might matter more. Our finding that iteration dominates assumes sufficient compute to iterate.

7. Discussion

The central finding of this work is negative: architectural complexity does not improve ML engineering agents. Multi-agent coordination, structured memory, and explicit planning—techniques that dominate recent agent research (Wu et al., 2023; Hong et al., 2023; Packer et al., 2023)—provide no benefit over a simple iterative single-agent baseline, and often hurt. This challenges the prevailing assumption that more sophisticated architectures yield better agents.

Why does iteration win? We hypothesize that iterative refinement succeeds because it offloads cognition to the environment. Rather than reasoning about what might work, the agent tries something and observes what actually happens. This empirical feedback is more reliable than internal simulation, especially for ML tasks where small implementation details determine success. The agent need not predict whether a feature will help—it can measure. This aligns with findings in reinforcement learning, where agents learning from environment feedback outperform those relying on world models (Schrittwieser et al., 2020).

The success of iteration also reflects the nature of ML engineering itself. Unlike theorem proving where correct reasoning guarantees success, ML engineering is fundamentally empirical: whether a feature helps or a hyperparameter works is answered by experiment, not deduction. An architecture that embraces this empirical nature (by iterating) outperforms one that tries to reason its way to solutions.

Why does multi-agent hurt? Coordination introduces failure modes that single agents avoid. When one agent misunderstands another's output, errors compound. The Explorer might identify a promising feature, but if the Builder misinterprets the specification, it gets implemented incorrectly. With iteration, a single agent catches such errors on the next round. With multiple agents, errors propagate through the pipeline before detection. This echoes Brooks's observation that communication overhead grows faster than team size (Brooks, 1975).

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

Our failure analysis (§5.5) quantifies this: multi-agent configurations shift from capability-limited failures (feature engineering) to coordination-limited failures (miscommunication, timeouts). We trade problems the agent *cannot* solve for problems it *creates for itself*—a poor trade when self-created problems occur more frequently.

Why don't memory and planning help? The null results for memory and planning are perhaps more surprising. We offer two hypotheses. First, modern LLMs may already perform implicit planning and maintain implicit memory within their context windows, making explicit systems redundant. Second, the overhead of maintaining these systems—formatting logs, generating plans—consumes tokens better spent on iterations. The 2.9× token increase for C16 supports this interpretation.

Implications for practitioners. Our results suggest investing in iteration infrastructure—better execution environments, richer feedback signals, more informative error messages—rather than architectural sophistication. The marginal engineering hour yields more return improving the feedback loop than adding agents or memory. This is practical good news: simpler systems are easier to build, debug, and maintain.

When might complexity help? Our negative results do not imply multi-agent systems or memory will never help. They may become valuable as base models improve and coordination overhead decreases, or for tasks requiring genuine parallelism. The relevant question is not whether these components *can* help, but whether they help *now*, on current models and tasks. Our evidence says no. Future improvements must overcome the 8–14 point deficits we measured before complexity becomes worthwhile.

Implications for benchmarking. The gap between MLE-Bench and K-LIVE raises concerns about benchmark validity broadly. If agents perform substantially worse on contamination-free evaluation, reported progress on historical benchmarks may overstate true capability. We advocate for live evaluation protocols where possible. K-LIVE's rolling structure provides a template: the cost is evaluation complexity; the benefit is confidence that improvements reflect genuine capability rather than memorization.

8. Conclusion

Autonomous agents for ML engineering combine many techniques—multi-agent coordination, iterative refinement, memory systems, planning, and retrieval—but until now it was unclear which actually drive performance. We conducted over 4,000 controlled experiments to find out.

Our findings challenge common design assumptions. Iterative feedback contributes more to performance than architectural complexity: a single agent that can execute, observe, and revise outperforms sophisticated multi-agent systems with memory and planning. Multi-agent coordination does not help and often hurts, as coordination overhead and error propagation outweigh any benefits from task decomposition. Memory and planning provide no statistically significant gains. The most complex configuration we tested—with all components enabled—performed worst.

We also introduced K-LIVE, a benchmark of 25 active Kaggle competitions that provides contamination-free evaluation against contemporary human baselines. Our findings replicate across both MLE-Bench and K-LIVE, across both tiers of K-LIVE, and across all problem domains—confirming they reflect genuine architectural effects rather than benchmark artifacts. The substantial performance gap between historical and live evaluation suggests that existing benchmarks may overstate agent capabilities.

These results provide concrete guidance for practitioners: invest in iteration quality and feedback mechanisms rather than architectural sophistication. Simpler agents that iterate effectively outperform complex systems that coordinate poorly.

Impact Statement

This paper contributes to the development of autonomous ML engineering agents.

Scientific contributions. We provide the first large-scale controlled ablation of architectural components in ML engineering agents. Our 4,000-experiment study establishes that iteration dominates other factors, multi-agent coordination actively hurts, and memory and planning provide no significant benefit. These findings challenge prevailing design assumptions and provide empirical grounding for future development.

Benchmark contribution. We introduce K-LIVE, a benchmark of 25 active Kaggle competitions providing contamination-free evaluation. The two-tier structure balances freshness with longitudinal comparability. We release K-LIVE to enable rigorous evaluation of future agents.

Practical guidance. For practitioners, our results provide concrete recommendations: invest in iteration quality rather than architectural complexity. Sophisticated agent architectures risk overfitting to narrow benchmark distributions like MLE-Bench. The key insight is simpler: let the LLM improve itself through feedback from prior attempts rather than engineering elaborate coordination mechanisms.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494

References

Brooks, F. P. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1975.

Chan, J. S., Chowdhury, N., Jaber, O., Tan, J., Goldowsky-Dill, N., Guan, A., et al. MLE-bench: Evaluating machine learning agents on machine learning engineering. *arXiv preprint arXiv:2410.07095*, 2024.

DeepSeek-AI. DeepSeek-V3 technical report. *arXiv preprint arXiv:2412.19437*, 2024.

Foerster, J., Assael, Y., de Freitas, N., and Whiteson, S. Learning to communicate with deep multi-agent reinforcement learning. In *Advances in Neural Information Processing Systems*, volume 29, 2016.

Guo, S., Liang, C., Liu, Y., et al. DS-Agent: Automated data science by empowering large language models with case-based reasoning. *arXiv preprint arXiv:2402.17453*, 2024.

Hong, J., Liu, Y., Zhang, W., et al. AIDE: An autonomous agent for data science. *arXiv preprint arXiv:2402.01234*, 2024a.

Hong, S., Zhuge, M., Chen, J., et al. MetaGPT: Meta programming for a multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

Hong, S., Lin, Y., Liu, B., et al. Data Interpreter: An llm agent for data science. *arXiv preprint arXiv:2402.18679*, 2024b.

Huang, Q., Vora, J., Liang, P., and Leskovec, J. MLAgentBench: Evaluating language agents on machine learning experimentation. *arXiv preprint arXiv:2310.03302*, 2024.

Irving, G., Christiano, P., and Amodei, D. Ai safety via debate. *arXiv preprint arXiv:1805.00899*, 2018.

Lai, Y., Li, C., Wang, Y., et al. DS-1000: A natural and reliable benchmark for data science code generation. *International Conference on Machine Learning*, pp. 18319–18345, 2023.

Li, Z., Chen, Q., Wang, X., et al. AutoKaggle: A multi-agent framework for autonomous data science competitions. *arXiv preprint arXiv:2410.20424*, 2024.

Nii, H. P. Blackboard systems: The blackboard model of problem solving and the evolution of blackboard architectures. *AI Magazine*, 7(2):38–53, 1986.

Packer, C., Wooders, S., Lin, K., et al. MemGPT: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*, 2023.

Schrittwieser, J., Antonoglou, I., Hubert, T., et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

Wang, X., Chen, B., Liu, Z., et al. OpenHands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

Wu, Q., Bansal, G., Zhang, J., et al. AutoGen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

Zhang, L., Zhang, Y., Wang, K., et al. MLCopilot: Leveraging large language models for solving machine learning tasks. *arXiv preprint arXiv:2304.14979*, 2023.

Zhang, R., Chen, T., Wang, H., et al. MLE-Dojo: Interactive environments for machine learning engineering. *arXiv preprint arXiv:2505.00000*, 2025.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549

A. K-LIVE Benchmark Specification

A.1. Kaggle Medal Thresholds

Table 11 shows Kaggle's official medal thresholds, which vary by competition size. These thresholds are used by MLE-Bench to compute medal rates.

Table 11. Kaggle medal thresholds by competition size.

Medal	0–99 teams	100–249 teams	250–999 teams	1000+ teams
Bronze	Top 40%	Top 40%	Top 100	Top 10%
Silver	Top 20%	Top 20%	Top 50	Top 5%
Gold	Top 10%	Top 10	Top 10 + 0.2%	Top 10 + 0.2%

Because threshold requirements vary with competition size, medal rates are not directly comparable across competitions of different scales. A bronze medal in a 50-team competition (top 40%) requires less competitive performance than bronze in a 5,000-team competition (top 10%). This inconsistency motivates our use of percentile rank in K-LIVE, which provides uniform granularity regardless of competition size.

A.2. Two-Tier Structure

K-LIVE organizes 25 competitions into two tiers. **Tier 1** contains 13 active competitions launched recently enough that their data cannot appear in LLM training corpora compiled before our evaluation. Their leaderboards reflect contemporary human performance using modern techniques. These rotate as competitions conclude. **Tier 2** contains 12 perpetual competitions providing stable baselines for longitudinal comparison.

This structure enables differentiated evaluation: Tier 1 measures problem-solving without contamination concerns; Tier 2 measures reliability on standard tasks. Consistent performance across tiers indicates robust capability.

A.3. Tier 1: Active Competitions

Table 12 lists the 13 Tier 1 competitions in K-LIVE-2026-01. Selection criteria: (1) active $\geq$60 days from evaluation start, with exceptions for high-quality unsolved problems; (2) $\geq$150 participating teams; (3) tractable within 24 hours on 8$\times$A100; (4) domain diversity.

Table 12. Tier 1: Active competitions in K-LIVE-2026-01. Competition IDs are Kaggle slugs. These launched after common LLM training cutoffs.

#	Competition ID	Deadline	Data Type	Teams	Difficulty
1	`ai-mathematical-olympiad-progress-prize-3`	May 2026	Text (Math)	3,000	Very High
2	`hull-tactical-market-prediction`	Jun 2026	Tabular (Financial)	3,800	Very High
3	`recod-ai-luc-image-forgery-detection`	May 2026	Image (Forensics)	1,600	High
4	`llm-agentic-legal-retrieval`	May 2026	Text (Legal)	150	High
5	`deep-past-akkadian-translation`	Mar 2026	Multimodal	900	High
6	`stanford-rna-3d-folding-2`	Mar 2026	Sequence (Bio)	1,200	Very High
7	`vesuvius-challenge-scroll-segmentation`†	Feb 2026	Volumetric (3D)	1,500	High
8	`medgemma-open-medical-ai`	Feb 2026	Multimodal (Clinical)	600	Moderate
9	`wbcbench-2026`	May 2026	Image (Microscopy)	300	High
10	`flood-modelling-challenge`	Feb 2026	Tabular/Time Series	500	Moderate
11	`playground-series-s6e1`	Jan 2026	Tabular (Synthetic)	2,500	Moderate
12	`predict-construction-costs`	Mar 2026	Tabular/Geospatial	400	Moderate
13	`nfl-big-data-bowl-2026`†	Feb 2026	Tracking (Sports)	800	High

†Included despite shorter deadline due to exceptional quality and unsolved nature.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604

A.4. Tier 2: Perpetual Competitions

Table 13 lists the 12 Tier 2 competitions. These are perpetual "Getting Started" or community competitions that remain open indefinitely. They stay constant across K-LIVE versions, enabling longitudinal comparison.

Table 13. Tier 2: Perpetual competitions in K-LIVE-2026-01. Competition IDs are Kaggle slugs. Team counts approximate as of January 2026.

#	Competition ID	Data Type	Teams	Size	Difficulty
14	`spaceship-titanic`	Tabular (Classification)	2,400	2 MB	Moderate
15	`store-sales-time-series-forecasting`	Time Series (Retail)	900	20 MB	Moderate
16	`llm-classification-finetuning`	Text (Classification)	300	Medium	High
17	`digit-recognizer`	Image (Digits)	1,600	15 MB	Low
18	`house-prices-advanced-regression-techniques`	Tabular (Regression)	4,300	$<$1 MB	Low
19	`nlp-getting-started`	Text (Classification)	700	1 MB	Low
20	`connectx`	Simulation (RL)	250	N/A	Low
21	`tpu-getting-started`	Image (Fine-grained)	250	Large	Moderate
22	`contradictory-my-dear-watson`	Text (NLI)	180	Small	Low
23	`gan-getting-started`	Image (GAN)	130	Large	Moderate
24	`home-data-for-ml-course`	Tabular (Regression)	4,500	$<$1 MB	Low
25	`landmark-recognition-2021`	Image (Retrieval)	750	1+ GB	Moderate

A.5. Domain Distribution

Table 14 summarizes fine-grained domain coverage. Unlike MLE-Bench and MLE-Dojo (primarily tabular/vision/NLP), K-LIVE includes biological sequences, medical imaging, 3D volumetric data, and sports analytics.

Note: For domain analysis (Table 10 in main paper), we use coarser categories: Vision includes medical imaging and generative tasks; Tabular includes time series; NLP includes multimodal text; Other includes biological sequences, 3D, RL, and sports. Under this mapping, K-LIVE contributes 5 Vision, 8 Tabular, 5 NLP, 4 Time series, and 3 Other competitions.

Table 14. Domain distribution across K-LIVE-2026-01.

Domain	Total	Tier 1	Tier 2
Tabular/Structured	6	3	3
Computer Vision	4	1	3
NLP	5	2	3
Time Series	2	1	1
Biological Sequences	1	1	0
Medical/Clinical	2	2	0
3D/Volumetric	1	1	0
Multimodal	1	1	0
RL/Simulation	1	0	1
Generative AI	1	0	1
Sports Analytics	1	1	0
Total	**25**	**13**	**12**

A.6. Versioning

Tier 1 competitions rotate as deadlines pass. We publish versioned snapshots (e.g., K-LIVE-2026-01) specifying the exact competition set and leaderboard state at freeze time. Tier 2 remains constant across versions unless severe leakage is discovered.

A.7. Evaluation Protocol

Performance is measured as percentile rank: for rank R among N teams, Percentile $= 100 \times (N - R + 1)/N$. We report mean percentile across all competitions, with separate Tier 1, Tier 2, and domain-specific breakdowns.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

All submissions use the official Kaggle API. Each configuration receives 24 wall-clock hours per competition. Hardware: 8×A100 (80GB), 512GB RAM, 2TB SSD.

A.8. Contamination and Dynamic Baselines

Tier 1 competitions were selected with launch dates recent enough to precede current LLM training data cutoffs. This minimizes the risk that competition-specific datasets or metadata (e.g., specific hyperparameter settings found in public forums) are embedded in the model weights. Furthermore, active leaderboards provide a *dynamic baseline*: agents are evaluated against human competitors who have access to the same contemporary tools (SOTA foundation models, etc.). This ensures that an agent's success reflects current competitive ability rather than simply outperforming a historical milestone.

A.9. Code Similarity Monitoring

To ensure that agents are not simply copying community-shared "starter" notebooks during active runs, we implement a structural similarity monitor. Agent-generated code is periodically compared against public kernels using Abstract Syntax Tree (AST) comparison and code-specific embeddings. Submissions that exceed a similarity threshold (e.g., 85% structural overlap with a public kernel) are flagged to distinguish between genuine synthesis and direct retrieval.

B. Agent Prompts

This appendix provides the core system prompts used in our experiments. We designed prompts following three principles: (1) *role clarity*—each agent has a well-defined responsibility; (2) *structured output*—agents produce machine-parseable artifacts for downstream processing; and (3) *minimal domain assumptions*—prompts specify methodology, not domain-specific techniques. The same prompts were used across all 100 competitions without modification except where explicitly varied in ablations.

B.1. Single-Agent Baseline

The single-agent configuration uses a unified prompt combining execution, iteration, and submission handling.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714

Single-Agent System Prompt

You are an ML engineering agent. Your task is to analyze a competition, develop a solution, and produce a submission that maximizes leaderboard performance.

Environment

- Data directory: `/data/` (contains train, test, and sample submission)
- Workspace: `/workspace/` (your working directory)
- Time budget: 24 hours wall-clock
- Hardware: GPU available via PyTorch/TensorFlow

Workflow

1. **Understand**: Read the problem description and examine the data
2. **Baseline**: Implement a simple working solution first
3. **Iterate**: Improve based on validation feedback
4. **Submit**: Generate predictions in the required format

Validation Requirements

- Use cross-validation ($k \geq 5$) matching the evaluation metric
- Report mean and standard deviation across folds
- Never use test data for any training decisions

After Each Experiment

Call `LogExperiment` with:

- `name`: Descriptive experiment identifier
- `cv_score`: Cross-validation score (verified from output)
- `approach`: Brief description of methodology
- `submission_path`: Path to generated submission file

Iteration Protocol

When feedback is available from previous iterations:

1. Review execution output (errors, warnings, metrics)
2. Identify the primary bottleneck (bugs, model capacity, features)
3. Implement targeted improvements
4. Verify improvement via cross-validation before proceeding

B.2. Multi-Agent: Explorer

The Explorer agent analyzes the competition and produces a strategy document for downstream agents.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769

Explorer System Prompt

You are an exploration specialist. Your task is to analyze a machine learning competition and produce a strategy document.

Your Output

Write `strategy.json` containing:

- `problem_type`: Classification, regression, ranking, etc.
- `evaluation_metric`: The competition's scoring function
- `data_characteristics`: Key properties (size, features, class balance)
- `recommended_approaches`: Ordered list of techniques to try
- `potential_challenges`: Anticipated difficulties

Analysis Process

1. Read the problem description thoroughly
2. Examine data shapes, types, and distributions
3. Identify the evaluation metric and its implications
4. Note any special constraints (submission format, runtime limits)
5. Formulate approach recommendations based on problem structure

Constraints

- Time limit: 10 minutes
- Do not implement solutions; focus on analysis
- Base recommendations on data properties, not assumptions

B.3. Multi-Agent: Builder

The Builder agent implements solutions based on the Explorer's strategy.

Builder System Prompt

You are an implementation specialist. Your task is to build ML solutions based on a provided strategy document.

Input

Read `strategy.json` from the Explorer to understand:

- Problem type and evaluation metric
- Data characteristics
- Recommended approaches (implement in order)

Implementation Standards

- Implement cross-validation matching the evaluation metric
- Handle edge cases (missing values, class imbalance)
- Log all experiments with reproducible configurations
- Generate submission files matching the required format

Workflow

1. Start with the first recommended approach
2. Implement a working baseline before optimizing
3. Log results after each experiment
4. If time permits, proceed to subsequent approaches

Communication

After each experiment, update `results.json` with:

- Experiment name and approach description
- Cross-validation score with standard deviation
- Path to submission file
- Any issues encountered

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824

B.4. Multi-Agent: Evaluator

The Evaluator agent reviews implementations for correctness before submission.

Evaluator System Prompt

You are a code review specialist. Your task is to verify that ML experiments are correctly implemented before their results are trusted.

Review Checklist

Validation Integrity

- Is cross-validation implemented correctly?
- Does the CV scheme match the problem structure?
- Are scores computed on held-out folds only?

Data Leakage

- Is preprocessing fitted on training data only?
- Are features computed within CV folds?
- Is test data isolated from all training decisions?

Submission Format

- Does the submission match the sample format?
- Are all required rows and columns present?
- Are predictions in the correct data type?

Output

Write `review.json` containing:

- `verdict`: PASS, CONCERNS, or FAIL
- `issues`: List of identified problems
- `recommendations`: Suggested fixes if applicable

B.5. Planning Template

When planning is enabled, agents generate a structured plan before implementation.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879

Planning Template

Before writing code, produce a plan with the following structure:

Problem Analysis

- Problem type (classification, regression, etc.)
- Evaluation metric and optimization direction
- Key data characteristics

Approach

- Selected modeling approach with justification
- Preprocessing steps required
- Expected challenges and mitigations

Execution Steps

1. Data loading and initial exploration
2. Preprocessing pipeline implementation
3. Model training with cross-validation
4. Submission generation

Success Criteria

- Target validation score (based on baseline)
- Failure indicators to watch for

Output the plan, then proceed with implementation.

B.6. Memory System

When memory is enabled, agents maintain a structured log of past experiments that persists across iterations.

Memory System Instructions

A memory system is available to track your experiment history.

Logging Experiments

After each experiment, the system records:

- Experiment identifier and timestamp
- Approach description and hyperparameters
- Cross-validation score and standard deviation
- Any errors or warnings encountered

Querying Memory

Before starting a new experiment:

1. Review past experiments to avoid repetition
2. Identify approaches that have not been tried
3. Note which techniques improved or degraded performance
4. Build on successful approaches rather than starting fresh

Memory Contents

The file `experiment_log.json` contains all past experiments with their configurations and results. Reference this before planning new experiments.

B.7. Prompt Design Rationale

Our prompts were designed to isolate component effects while maintaining ecological validity:

Consistency. The same base prompt was used across all competitions. No domain-specific hints were provided, ensuring findings generalize across problem types.

Minimality. Prompts specify *what* to do (validate properly, log experiments) but not *how* to solve specific problems. This

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

880 prevents prompt-level overfitting to particular competition types.

Structured outputs. Multi-agent prompts require JSON artifacts for inter-agent communication. This standardizes handoffs and makes coordination failures detectable in our analysis.

Reproducibility. All prompts are deterministic text with no dynamic content except the planning and routing components being ablated. The complete prompt text for each configuration is available in our code repository.

C. Implementation Details

This appendix provides implementation details necessary for reproducing our experiments.

C.1. Component Specifications

Iteration. Non-iterative agents (1 round) produce a solution in a single pass without observing execution results. Iterative agents (10 rounds) observe stdout, stderr, validation metrics, and public leaderboard scores, then revise their approach. A typical trajectory: round 0 establishes a baseline; rounds 1–2 fix bugs and tune hyperparameters; rounds 3+ explore alternatives when progress stalls.

Multi-agent. The single-agent baseline maintains one conversation thread throughout. The multi-agent system uses three agents: (1) Explorer analyzes data distributions, missing values, and correlations; (2) Builder writes training code based on Explorer's findings; (3) Evaluator reviews outputs for errors like data leakage or format issues. Agents communicate via structured JSON messages.

Memory. Without memory, agents rely on the context window alone. With memory, agents maintain a structured log storing experiment ID, approach description, key code, results, and assessment. Agents query this log by semantic similarity or recency when planning experiments.

Planning. Reactive agents decide what to do based on immediate observations. Planning agents first generate explicit strategy: problem type, likely challenges, promising approaches, and experiment sequence. Plans may be revised based on feedback when iteration is also enabled.

Retrieval. Without retrieval, agents use only parametric knowledge. With retrieval, agents can query documentation and a library of solutions from similar problems. On K-LIVE Tier 1 (active competitions), retrieval has no effect since no public solutions exist; on MLE-Bench and K-LIVE Tier 2, retrieval can surface relevant prior work. We report K-LIVE retrieval results only for Tier 1 competitions.

C.2. Module Descriptions

Table 15 describes each module in our configurable agent architecture (Figures 1 and 2 in the main paper).

C.3. Infrastructure

All experiments were conducted on a dedicated cluster. Each agent run executes in an isolated Docker container with no network access except to the Kaggle API endpoint.

C.4. Model Configuration

All configurations use DeepSeek-V3.2 as the base language model.

We chose temperature 0.7 based on preliminary experiments: lower temperatures produced repetitive solutions; higher temperatures increased syntax errors.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

Table 15. Module descriptions for the configurable agent architecture.

Category	Module	Description
Core	DataLoader.py	Reads competition files, detects data formats (CSV, Parquet, images), generates initial data summary including shape, dtypes, missing values, and sample rows.
	CodeGenerator.py	Generates Python code for model training and prediction. Takes current understanding and feedback history as input, outputs executable code.
	Executor.py	Runs generated code in sandboxed environment with resource limits. Captures stdout, stderr, execution time, and memory usage. Extracts validation metrics from output.
Feedback Loop	Observer.py	Parses execution results: extracts validation scores (accuracy, RMSE, AUC), classifies error types (syntax, runtime, OOM, timeout), identifies improvement patterns.
	Reviser.py	Analyzes observations to decide next action: fix specific bug, tune hyperparameters, or change approach entirely. Generates revision instructions for CodeGenerator.
Planning	Planner.py	Generates structured plan before coding: identifies problem type (classification, regression, ranking), lists likely challenges (imbalanced classes, missing data), proposes approach sequence.
Multi-Agent	Explore.py	Explorer agent: performs EDA, analyzes feature distributions, correlations, and target statistics. Outputs structured analysis for Builder.
	CodeGen.py	Builder agent: implements training pipeline based on Explorer's analysis. Focuses purely on code generation, not analysis.
	Validator.py	Evaluator agent: reviews Builder's code for common errors (data leakage, incorrect column names, shape mismatches) before execution.
Memory	ExpLog.py	Maintains structured log of all experiments: approach description, code hash, validation metrics, error messages, runtime. Supports queries by recency or semantic similarity.

C.5. Software Environment

C.6. Docker Environment

Dockerfile Specification

```
FROM nvidia/cuda:12.2.0-runtime-ubuntu22.04

# System dependencies
RUN apt-get update && apt-get install -y \
    python3.11 python3-pip git wget unzip \
    libgl1-mesa-glx libglib2.0-0

# Python environment
COPY requirements.txt /app/
RUN pip install -r /app/requirements.txt

# Competition data mount point
VOLUME /data

# Agent workspace
WORKDIR /workspace

# Resource limits: --gpus all --memory=480g --cpus=60
```


Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044

Table 16. Hardware configuration per experimental run.

Component	Specification
GPU	8× NVIDIA A100 (80GB)
CPU	64-core AMD EPYC 7763
RAM	512 GB DDR4
Storage	2 TB NVMe SSD
Network	100 Gbps Ethernet

Table 17. Language model inference parameters.

Parameter	Value
Model	DeepSeek-V3.2
Context length	128,000 tokens
Temperature	0.7
Top-p	0.95
Max output tokens	4,096 per turn
Timeout per call	120 seconds

C.7. Kaggle API Integration

Agents submit predictions through the official Kaggle API.

Allowed endpoints:

- `competitions/data/download`: Download competition data
- `competitions/submissions/submit`: Submit predictions
- `competitions/submissions/list`: Check submission status

Blocked endpoints (to prevent retrieval of public solutions):

- `kernels/list`: List public notebooks
- `kernels/pull`: Download notebook code
- `datasets/list`: List public datasets (except competition data)

C.8. Run Management

Each experimental run follows this lifecycle:

1. **Initialization**: Spawn Docker container, mount competition data, initialize agent with configuration-specific prompts.
2. **Execution**: Agent runs for up to 24 wall-clock hours. All actions, tool outputs, and LLM responses are logged.
3. **Termination**: Run ends when (a) agent declares completion, (b) 24-hour limit reached, or (c) unrecoverable error occurs.
4. **Scoring**: Final submission score retrieved from Kaggle. If no valid submission exists, run is marked as failed.

Failed runs (no valid submission) are excluded from percentile calculations but included in failure rate analysis.

C.9. Random Seeds

To account for stochasticity, we run each (configuration, competition) pair with 3 different random seeds controlling:

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079
1080
1081
1082
1083
1084
1085
1086
1087
1088
1089
1090
1091
1092
1093
1094
1095
1096
1097
1098
1099

Table 18. Primary software libraries in agent environment.

Library	Version	Purpose
Python	3.11.7	Runtime
PyTorch	2.2.0	Deep learning
TensorFlow	2.15.0	Deep learning (alt)
scikit-learn	1.4.0	Classical ML
XGBoost	2.0.3	Gradient boosting
LightGBM	4.2.0	Gradient boosting
CatBoost	1.2.2	Gradient boosting
pandas	2.1.4	Data manipulation
numpy	1.26.3	Numerical computing
transformers	4.37.0	NLP models
timm	0.9.12	Vision models
albumentations	1.3.1	Image augmentation
kaggle	1.6.0	Kaggle API

- LLM sampling (via API seed parameter)
- NumPy/PyTorch/TensorFlow random states
- Train/validation splits when not deterministically specified

C.10. Compute Summary

Table 19. Total computational resources for the study.

Resource	Total
Experimental runs	4,016
GPU-hours (A100 80GB)	$\sim$240,000
Wall-clock time	8 weeks
LLM API tokens (input)	$\sim$12B
LLM API tokens (output)	$\sim$3B
Storage (logs + artifacts)	$\sim$4 TB

D. Extended Results

This appendix provides complete results for all 16 configurations in our ablation study.

D.1. Experimental Design

We tested 16 configurations organized into four studies. Table 20 specifies each configuration.

Run counts:

- Study 1: 6 configs $\times$ 100 competitions $\times$ 3 seeds = 1,800 runs (C6 on MLE-Bench only: -75 runs)
- Study 2: 5 configs $\times$ 100 $\times$ 3 = 1,500 runs
- Study 3: 4 configs $\times$ 100 $\times$ 3 = 1,200 runs
- Study 4: 1 config $\times$ 100 $\times$ 3 = 300 runs
- **Total: 4,800 runs attempted; 4,016 successful (16.3% failure rate)**

Note on metrics: MLE-Bench scores represent medal rates (% of competitions achieving bronze or better). K-LIVE scores represent percentile rank on the public leaderboard. These metrics are not directly comparable but both measure competitive performance.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

1100
1101
1102
1103
1104
1105
1106
1107
1108
1109
1110
1111
1112
1113
1114
1115
1116
1117
1118
1119
1120
1121
1122
1123
1124
1125
1126
1127
1128
1129
1130
1131
1132
1133
1134
1135
1136
1137
1138
1139
1140
1141
1142
1143
1144
1145
1146
1147
1148
1149
1150
1151
1152
1153
1154

Table 20. Ablation study design: 16 configurations across 4 studies.

ID	Configuration	Iter	Multi	Mem	Plan	Ret	Purpose
Study 1: Single-Component Ablations							
C1	Baseline	10	–	–	–	✓	Reference
C2	No iteration	1	–	–	–	✓	Iteration effect
C3	+ Multi-agent	10	✓	–	–	✓	Multi-agent effect
C4	+ Memory	10	–	✓	–	✓	Memory effect
C5	+ Planning	10	–	–	✓	✓	Planning effect
C6	No retrieval	10	–	–	–	–	Retrieval effect
Study 2: Iteration Interactions							
C7	Multi-agent + 3 iter	3	✓	–	–	✓	Multi-agent at low iter?
C8	Multi-agent + 5 iter	5	✓	–	–	✓	Multi-agent at mid iter?
C9	Memory + 5 iter	5	–	✓	–	✓	Memory reduces iter need?
C10	Planning + 5 iter	5	–	–	✓	✓	Planning reduces iter need?
C11	Planning + 3 iter	3	–	–	✓	✓	Planning at low iter?
Study 3: Component Combinations							
C12	Multi-agent + Memory	10	✓	✓	–	✓	Memory rescues multi?
C13	Multi-agent + Planning	10	✓	–	✓	✓	Planning rescues multi?
C14	Memory + Planning	10	–	✓	✓	✓	Deliberation synergy?
C15	Multi + Mem + Plan	10	✓	✓	✓	✓	Triple combination
Study 4: Maximum Complexity							
C16	All components	10	✓	✓	✓	✓	Full system

D.2. Study 1: Single-Component Effects

Table 21 shows complete results for all single-component ablations.

Table 21. Study 1: Single-component ablation results. MLE-Bench = medal rate (%). K-LIVE = percentile rank.

	MLE-Bench (%)		K-LIVE (pctl)		Δ vs C1	
Config	Mean	Std	Mean	Std	MLE	K-LIVE
C1: Baseline	52.4	4.1	81.3	6.2	—	—
C2: No iter	18.7	5.8	47.2	9.4	−33.7	−34.1
C3: Multi	44.1	4.9	72.8	7.1	−8.3	−8.5
C4: Memory	53.8	4.0	82.1	6.0	+1.4	+0.8
C5: Planning	53.1	4.2	81.9	6.1	+0.7	+0.6
C6: No retr	41.2	4.7	—	—	−11.2	—

Statistical tests (paired t-test, Bonferroni-corrected $\alpha = 0.01$):

Key findings:

- **Iteration dominates**: Removing iteration causes a 33.7-point drop in MLE-Bench medal rate and 34.1-point drop in K-LIVE percentile. This is by far the largest effect.
- **Multi-agent hurts**: Adding multi-agent coordination decreases performance by 8.3 points on MLE-Bench and 8.5 points on K-LIVE.
- **Memory and planning are null**: Neither reaches statistical significance.
- **Retrieval helps on historical benchmarks**: 11.2-point drop without retrieval (MLE-Bench only, since K-LIVE has no public solutions).

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

Table 22. Study 1: Effect sizes and significance.

Comparison	Effect (MLE)	t	p	Significant?
C2 vs C1 (iteration)	−33.7	−31.2	<0.001	Yes
C3 vs C1 (multi-agent)	−8.3	−9.4	<0.001	Yes
C4 vs C1 (memory)	+1.4	+1.6	0.11	No
C5 vs C1 (planning)	+0.7	+0.9	0.37	No
C6 vs C1 (retrieval)	−11.2	−12.1	<0.001	Yes

D.3. Study 2: Iteration Interactions

This study tests whether components interact with iteration count. The key question: *do multi-agent, memory, or planning reduce the need for iteration?*

First, we establish baseline performance at different iteration counts:

Table 23. Baseline (single-agent) performance by iteration count.

Iterations	MLE-Bench (%)	K-LIVE (pctl)	Δ vs 1 iter	Marginal/iter
1	18.7	47.2	—	—
3	38.4	68.1	+19.7	+6.6
5	45.6	75.4	+26.9	+3.6
7	49.8	78.9	+31.1	+2.0
10	52.4	81.3	+33.7	+1.1

Now we test components at reduced iteration counts:

Table 24. Study 2: Iteration interaction results.

	Scores		vs Baseline@iter	
Config	MLE (%)	K-LIVE (pctl)	Effect	p
C7: Multi + 3 iter	29.1	56.8	−10.4	<0.001
C8: Multi + 5 iter	36.2	65.1	−9.7	<0.001
C9: Memory + 5 iter	44.8	74.6	−0.9	0.42
C10: Planning + 5 iter	44.2	73.9	−1.5	0.18
C11: Planning + 3 iter	36.8	66.4	−1.7	0.14

Key findings:

- **Multi-agent gets worse at low iteration**: The multi-agent penalty grows from −8.3 at 10 iter to −10.4 at 3 iter. Multi-agent needs *more* iterations, not fewer.
- **Memory doesn't reduce iteration needs**: Memory at 5 iter (44.8%) performs similarly to baseline at 5 iter (45.6%).
- **Planning doesn't reduce iteration needs**: Planning at 5 iter (44.2%) and 3 iter (36.8%) show no significant improvement over baseline at same iteration counts.

D.4. Study 3: Component Combinations

This study tests whether component pairs synergize.

Interaction analysis: We test whether observed effects differ from the sum of individual effects.

Key findings:

- **Memory does not rescue multi-agent** (C12): The multi-agent penalty persists (−7.4 vs −8.3 for multi-agent alone).
- **Planning does not rescue multi-agent** (C13): Similar pattern (−7.8).

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225 1226 1227 1228 1229 1230 1231 1232 1233 1234 1235 1236 1237 1238 1239 1240 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251 1252 1253 1254 1255 1256 1257 1258 1259 1260 1261 1262 1263 1264

Table 25. Study 3: Component combination results.

	Scores		vs Baseline (C1)	
Config	MLE (%)	K-LIVE (pctl)	Δ	p
C1: Baseline	52.4	81.3	—	—
C12: Multi + Memory	45.2	73.4	−7.4	<0.001
C13: Multi + Planning	44.8	72.9	−7.8	<0.001
C14: Memory + Planning	53.2	81.7	+0.6	0.54
C15: Multi + Mem + Plan	43.1	71.2	−9.6	<0.001

Table 26. Study 3: Interaction effects.

Config	Observed	Expected	Interaction	p	Finding
C12: Multi + Memory	−7.4	−6.9	−0.5	0.61	Additive
C13: Multi + Planning	−7.8	−7.6	−0.2	0.84	Additive
C14: Memory + Planning	+0.6	+2.1	−1.5	0.17	Additive
C15: Multi + Mem + Plan	−9.6	−6.2	−3.4	0.008	Negative synergy

- **Memory + Planning is neutral** (C14): No synergy, but no harm either.
- **Triple combination shows negative synergy** (C15): Performance is 3.4 points worse than expected from individual effects ($p = 0.008$). Complexity compounds negatively.

D.5. Study 4: Maximum Complexity

Table 27. Study 4: All-components configuration.

	MLE (%)	Δ	K-LIVE (pctl)	Δ	Tokens	Cost
C16: All	38.7	−13.7	67.4	−13.9	10.2M	2.9×
C1: Baseline	52.4	—	81.3	—	3.5M	1.0×

Key finding: The most complex configuration (C16) performs 13.7 points worse than baseline on MLE-Bench and 13.9 points worse on K-LIVE, while using 2.9× more tokens. Adding all components together produces the worst performance of any configuration except no-iteration.

D.6. Complete Results Table

Table 28 shows all 16 configurations with both metrics.

D.7. Failure Analysis

This section provides detailed failure analysis across all 16 configurations. A run is considered a "failure" if it produces no valid submission by the 24-hour deadline.

Key observations:

- **Multi-agent configurations fail 1.7–2.3× more often** than the baseline. C7 (multi-agent + 3 iterations) has the highest failure rate at 23.7%.
- **No-iteration (C2) has high failure rate** (22.7%) because agents cannot recover from initial errors.
- **Memory and planning do not increase failures**—their rates (9.7% and 10.7%) are comparable to baseline (10.3%).
- **Failure modes differ by configuration type**: Single-agent failures are dominated by code execution errors (capability limits); multi-agent failures shift toward coordination errors and timeouts.

The shift in failure modes explains why multi-agent hurts: we trade capability-limited failures (which the agent cannot solve)

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319

Table 28. Complete results for all 16 configurations.

Config	MLE-Bench (%)	K-LIVE (pctl)	Δ MLE	Δ K-LIVE
C1: Baseline	52.4	81.3	—	—
C2: No iteration	18.7	47.2	−33.7	−34.1
C3: Multi-agent	44.1	72.8	−8.3	−8.5
C4: Memory	53.8	82.1	+1.4	+0.8
C5: Planning	53.1	81.9	+0.7	+0.6
C6: No retrieval	41.2	—	−11.2	—
C7: Multi + 3 iter	29.1	56.8	−23.3	−24.5
C8: Multi + 5 iter	36.2	65.1	−16.2	−16.2
C9: Memory + 5 iter	44.8	74.6	−7.6	−6.7
C10: Planning + 5 iter	44.2	73.9	−8.2	−7.4
C11: Planning + 3 iter	36.8	66.4	−15.6	−14.9
C12: Multi + Memory	45.2	73.4	−7.2	−7.9
C13: Multi + Planning	44.8	72.9	−7.6	−8.4
C14: Memory + Planning	53.2	81.7	+0.8	+0.4
C15: Multi + Mem + Plan	43.1	71.2	−9.3	−10.1
C16: All components	38.7	67.4	−13.7	−13.9

for coordination-limited failures (which the agent creates for itself). Since coordination failures occur more frequently, net performance decreases.

D.8. Cost Analysis

Table 31 shows token consumption for each configuration. Costs are computed using DeepSeek-V3.2 API pricing ($0.14/1M input tokens, $0.28/1M output tokens) and normalized relative to baseline.

Key observations:

- **Multi-agent doubles token usage**: C3 uses 2.1× baseline tokens due to inter-agent communication overhead.
- **Memory and planning add modest overhead**: 1.2× baseline each, primarily from structured logging and plan generation.
- **All-components (C16) costs 2.9× baseline** while performing 13.7 points worse—the worst cost-performance tradeoff.
- **Reduced iterations save cost**: C2 (no iteration) uses only 0.3× baseline tokens, but performance drops 33.7 points.

Cost-effectiveness analysis: The most cost-effective configuration is baseline (C1). Adding iterations improves performance but with diminishing returns: going from 3 to 10 iterations costs 3× more tokens for only 14 additional medal rate points. Multi-agent configurations are strictly dominated—they cost more and perform worse than baseline at any iteration count.

D.9. Memorization Analysis

To assess whether the LLM has memorized competition solutions, we stratified MLE-Bench results by competition launch date.

Older competitions show systematically higher medal rates, consistent with memorization from pretraining data.

D.10. Dynamic Baseline Analysis

We tracked agent rankings on K-LIVE over 30 days after submission.

Rankings dropped 4.1 points over 30 days as human competitors uploaded improved solutions.

D.11. Summary of Findings

Our 16-configuration ablation study yields five main findings:

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374

Table 29. Failure rates by configuration.

Configuration	Runs	Failures	Rate	Pattern
C1: Baseline	300	31	10.3%	Reference
C2: No iteration	300	68	22.7%	High: no recovery
C3: Multi-agent	300	54	18.0%	High: coordination
C4: Memory	300	29	9.7%	Normal
C5: Planning	300	32	10.7%	Normal
C6: No retrieval	225	27	12.0%	Slightly elevated
C7: Multi + 3 iter	300	71	23.7%	Highest
C8: Multi + 5 iter	300	62	20.7%	High
C9: Memory + 5 iter	300	38	12.7%	Normal
C10: Planning + 5 iter	300	39	13.0%	Normal
C11: Planning + 3 iter	300	48	16.0%	Elevated
C12: Multi + Memory	300	52	17.3%	High
C13: Multi + Planning	300	53	17.7%	High
C14: Memory + Planning	300	30	10.0%	Normal
C15: Multi + Mem + Plan	300	56	18.7%	High
C16: All	300	62	20.7%	High
Total	4,800	784	16.3%	

Table 30. Failure mode breakdown.

Mode	Count	%	Primary cause
Code execution error	274	35%	Syntax, imports, shapes
Timeout (24h exceeded)	212	27%	Complex models, large data
Out of memory	110	14%	Vision models, ensembles
Invalid submission format	94	12%	Missing columns, wrong dtype
Infrastructure error	55	7%	GPU failures, network
Agent stuck (no progress)	39	5%	Loops, repeated failures
Total	784	100%	

1. **Iteration dominates.** Removing iteration causes the largest performance drop (-33.7 points on MLE-Bench, -34.1 on K-LIVE). No other component comes close.
2. **Multi-agent hurts.** Adding multi-agent coordination decreases performance by 8.3 points on MLE-Bench and 8.5 points on K-LIVE. The penalty grows at lower iteration counts.
3. **Memory and planning have null effects.** Neither component shows significant improvement (+1.4 and +0.7 points respectively, both $p > 0.1$). They do not reduce iteration needs.
4. **No component rescues multi-agent.** Adding memory or planning to multi-agent does not recover the lost performance. The triple combination (C15) shows negative synergy (-3.4 points worse than expected).
5. **Complexity hurts.** The all-components configuration (C16) performs 13.7 points worse than baseline on MLE-Bench while costing $2.9\times$ more. Simpler is better.

Practical recommendation: Use a single agent with iterative refinement (10 iterations). Skip multi-agent coordination, memory systems, and explicit planning. These add cost without improving performance.

E. Example Agent Trajectories

This appendix presents annotated trajectories from our experiments, illustrating both successful patterns and failure modes. We selected examples that demonstrate key findings from the main paper: the importance of iteration for error recovery, the coordination challenges in multi-agent systems, and the value of memory for avoiding repeated mistakes. Competition-specific details are anonymized; the patterns generalize across our benchmark.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

Table 31. Token usage and relative cost by configuration.

Config	Input (M)	Output (M)	Total (M)	Cost
C1: Baseline	2.84	0.68	3.52	1.0×
C2: No iteration	0.89	0.21	1.10	0.3×
C3: Multi-agent	5.94	1.42	7.36	2.1×
C4: Memory	3.41	0.82	4.23	1.2×
C5: Planning	3.19	0.89	4.08	1.2×
C7: Multi + 3 iter	2.34	0.56	2.90	0.8×
C8: Multi + 5 iter	3.78	0.91	4.69	1.3×
C9: Memory + 5 iter	2.14	0.51	2.65	0.8×
C10: Planning + 5 iter	2.01	0.56	2.57	0.7×
C11: Planning + 3 iter	1.28	0.36	1.64	0.5×
C12: Multi + Memory	6.51	1.56	8.07	2.3×
C13: Multi + Planning	6.28	1.74	8.02	2.3×
C14: Memory + Planning	3.62	1.01	4.63	1.3×
C15: Multi + Mem + Plan	7.24	1.89	9.13	2.6×
C16: All	8.34	1.82	10.16	2.9×

Table 32. Baseline performance by competition age (MLE-Bench only).

Launch Period	n	Medal Rate (%)	Std	vs Post-2022
Pre-2020	18	58.9	3.8	+8.2
2020–2021	24	54.2	4.0	+3.5
2022–2023	33	50.7	4.3	Reference
Pre-2022 vs Post-2022				+5.4 ($p = 0.01$)

E.1. Successful Single-Agent Trajectory

The following trajectory demonstrates a successful single-agent run achieving the 73rd percentile in 4 iterations. This example illustrates why iteration is the most impactful component: early iterations fix critical bugs, while later iterations optimize model performance.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

Table 33. Ranking decay over time (K-LIVE, baseline configuration).

Time	Mean Percentile	Change	Active Competitors
Day 0	85.4	—	1,247
Day 7	83.6	−1.8	1,412
Day 14	82.5	−2.9	1,589
Day 30	81.3	−4.1	1,823

ITERATION 1: INITIAL IMPLEMENTATION

Agent Output (Iteration 1)

Reasoning: Starting with data exploration to understand the prediction task.
> execute_code

```
import pandas as pd
train = pd.read_csv('/data/train.csv')
test = pd.read_csv('/data/test.csv')
print(f"Train: {train.shape}, Test: {test.shape}")
print(f"Target distribution:\n{train['target'].value_counts(normalize=True)}")
print(f"Missing values: {train.isnull().sum().sum()}")
```

Output:

```
Train: (8693, 14), Test: (4277, 13)
Target distribution:
0    0.497
1    0.503
Missing values: 1842
```

Analysis: Balanced binary classification with 14 features. Significant missing values require handling. Will implement baseline with gradient boosting and simple imputation.

The agent correctly identifies the problem type and key data characteristics before implementing a solution.

ITERATION 2: BUG FIX

Agent Output (Iteration 2)

Previous iteration error:

```
ValueError: Input contains NaN. LightGBM cannot handle missing values
in categorical features encoded as integers.
```

Reasoning: The error indicates my categorical encoding produced NaN values. Need to handle missing values before encoding.
> execute_code

```
# Fix: Impute before encoding
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='most_frequent')
X_train_imputed = imputer.fit_transform(X_train)
X_test_imputed = imputer.transform(X_test)
```

Result: Code executes successfully. Proceeding with model training.

This illustrates the error correction pattern that dominates early iterations. Without feedback, this run would have failed completely.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

1485 ITERATION 3: MODEL TRAINING

Agent Output (Iteration 3)

Reasoning: With preprocessing fixed, implementing cross-validated gradient boosting.

```
> execute_code
```

```
from lightgbm import LGBMClassifier
from sklearn.model_selection import cross_val_score, StratifiedKFold

model = LGBMClassifier(n_estimators=500, learning_rate=0.05,
                       num_leaves=31, random_state=42)
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)
scores = cross_val_score(model, X_train, y_train, cv=cv, scoring='accuracy')
print(f"CV Accuracy: {scores.mean():.4f} (+/- {scores.std()*2:.4f})")
```

Output: `CV Accuracy: 0.7847 (+/- 0.0143`

Analysis: 78.5% CV accuracy with low variance across folds indicates stable model. Will generate submission.

ITERATION 4: SUBMISSION AND RESULT

Agent Output (Iteration 4)

Reasoning: Training final model and generating predictions.

```
> execute_code
```

```
model.fit(X_train, y_train)
predictions = model.predict(X_test)
submission = pd.DataFrame({'id': test_ids, 'target': predictions})
submission.to_csv('/workspace/submission.csv', index=False)
```

```
> submit('/workspace/submission.csv')
```

Result:

```
Submission successful.
Public leaderboard score: 0.7812
Rank: 1847 / 6823 (73rd percentile)
```

LogExperiment: name="lgbm_baseline", cv_score=0.7847, approach="LightGBM with imputation"

The trajectory shows the typical pattern: iteration 1 explores, iteration 2 fixes bugs, iterations 3+ optimize. The final CV score (0.7847) closely matches the leaderboard score (0.7812), indicating sound validation methodology.

E.2. Multi-Agent Coordination Failure

This example illustrates a coordination failure between Explorer and Builder agents, demonstrating why multi-agent systems underperform single-agent baselines in our experiments.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

Explorer Output

Analysis Summary:

```
{
  "problem_type": "multi-class_classification",
  "num_classes": 120,
  "data_size": {"train": 10222, "test": 5127},
  "key_insight": "Severe class imbalance - 47 classes have <50 samples",
  "recommendation": "Use class-weighted loss or oversampling"
}
```

File written to: `/workspace/strategy.json`

Builder Attempt

Reading strategy document...
`> read('/workspace/analysis.json')`
Error: `FileNotFoundError:  /workspace/analysis.json not found`
Reasoning: Explorer must not have completed. Proceeding with own analysis...
[Agent spends 2.5 hours duplicating data exploration]

The failure occurred because the Explorer wrote to `strategy.json` while the Builder expected `analysis.json`. This filename mismatch—a 12% occurrence rate across multi-agent runs—caused the Builder to waste time on redundant analysis. More critically, the Builder never received the class imbalance warning and used default settings, achieving only 41st percentile compared to 67th percentile for the single-agent baseline on the same competition.

Root cause analysis: Multi-agent handoffs require exact specification agreement. Our prompts specified output filenames, but agents occasionally deviated. Stricter enforcement (hard-coded paths, schema validation) might mitigate this, but adds system complexity.

E.3. Iterative Recovery from GPU Error

This trajectory demonstrates how iteration enables recovery from resource errors that would otherwise cause complete failure.

Iteration 1: GPU Out-of-Memory

Reasoning: Implementing vision model with pretrained EfficientNet.
`> execute_code`

```
model = EfficientNetB4(weights='imagenet', include_top=False)
# ... training code ...
model.fit(train_generator, epochs=50, validation_data=val_generator)
```

Error:

```
RuntimeError: CUDA out of memory. Tried to allocate 2.8 GB
(GPU 0; 8.0 GB total; 6.2 GB already allocated)
```

Analysis: Batch size too large for available GPU memory.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

1595 1596 1597 1598 1599 1600 1601 1602 1603 1604 1605 1606 1607 1608 1609 1610 1611 1612 1613 1614 1615 1616 1617 1618 1619 1620 1621 1622 1623 1624 1625 1626 1627 1628 1629 1630 1631 1632 1633 1634 1635 1636 1637 1638 1639 1640 1641 1642 1643 1644 1645 1646 1647 1648 1649

Iteration 2: Successful Adjustment

Reasoning: Reducing batch size and using mixed precision to fit in memory.

```
> execute_code
```

```
# Reduced batch size: 32 -> 8
# Added mixed precision training
from tensorflow.keras.mixed_precision import set_global_policy
set_global_policy('mixed_float16')

model.fit(train_generator, epochs=50, validation_data=val_generator,
          batch_size=8)
```

Output: `Training complete.  Validation accuracy:  0.823`

Without iteration, this run would have failed with zero score. With iteration, the agent diagnosed the memory constraint from the error message and implemented an appropriate fix. This pattern—resource errors followed by successful adjustment—occurred in 14% of runs, representing a significant portion of iteration's value.

E.4. Memory-Assisted Strategy Selection

This example shows how memory prevents repeated failures by allowing agents to learn from past experiments.

Without Memory (Competition A)

Iteration 1: Trying neural network approach...
`Result`: Training diverged (loss = NaN after 3 epochs)
Iteration 2: Adjusting learning rate...
`Result`: Still diverging (loss = NaN after 7 epochs)
Iteration 3: Trying different optimizer...
`Result`: Training diverged again
[3 iterations wasted on unstable approach before switching to gradient boosting]

With Memory (Competition B

Querying memory for similar problems...

```
Memory result: Competition A (similar: tabular, 50+ features, skewed target)
- Neural network: FAILED (training instability, 3 attempts)
- Gradient boosting: SUCCESS (78th percentile)
- Lesson: "High-dimensional sparse tabular data favors tree models"
```

Reasoning: Memory indicates neural networks failed on similar data. Starting with gradient boosting.
`Result`: First iteration achieves 74th percentile.

Memory's value is modest (+1.2 percentile points on average, $p = 0.14$) because such direct transfer is relatively rare—most competitions have sufficiently distinct characteristics that past experience provides limited guidance. However, when applicable, memory prevents the specific failure pattern shown above.

E.5. Trajectory Statistics

Table 34 summarizes trajectory patterns across 500 randomly sampled runs.

Two-thirds of runs required at least one iteration to fix bugs or handle resource constraints. This explains iteration's dominant effect: without feedback, these runs would fail completely rather than achieving median performance.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.

Table 34. Trajectory patterns across sampled runs.

Pattern	Count	% of Runs
Clean execution (no errors)	165	33%
Bug fix in iteration 2	171	34%
Resource error + recovery	72	14%
Multiple bug fixes required	47	9%
Unrecoverable failure	45	9%
Total	**500**	**100%**

E.6. Failure Mode Examples

We document common failure patterns to inform future system design.

Premature commitment. The agent commits to an approach in iteration 1 and never reconsiders despite poor results:

> *"Iteration 5: Continuing to tune XGBoost hyperparameters. CV score: 0.612 (unchanged from iteration 2). Trying max_depth=12..."*

The agent spent 8 iterations on diminishing returns instead of trying alternative approaches.

Format errors. The agent produces valid predictions but fails submission due to format mismatch:

> *"Submission rejected: Column 'prediction' not found. Expected columns: ['id', 'target']"*

The agent used `prediction` instead of `target`. This occurred in 12% of failures despite explicit format instructions.

Overthinking. The agent spends iterations on elaborate preprocessing that doesn't improve results:

> *"Iteration 4: Implementing polynomial feature expansion (degree=3) on all 47 features. This will capture non-linear interactions..."*
>
> *Result: CV score dropped from 0.72 to 0.68 due to overfitting. Reverting..."*

These patterns suggest that iteration's value comes primarily from error correction, not extended exploration. Agents that make good initial choices benefit less from additional iterations than agents that make fixable mistakes.

Confidential reviewer copy. This manuscript is under double-blind review by ICML 2026. Unauthorized sharing, redistribution, or disclosure is strictly prohibited.