

000 PILOT-BENCH: PROBABILISTIC INTERACTION FOR 001 LLM OPERATIONS IN TOOL-DRIVEN SCENARIOS 002

003 **Anonymous authors**
004

005 Paper under double-blind review
006

007 ABSTRACT 008

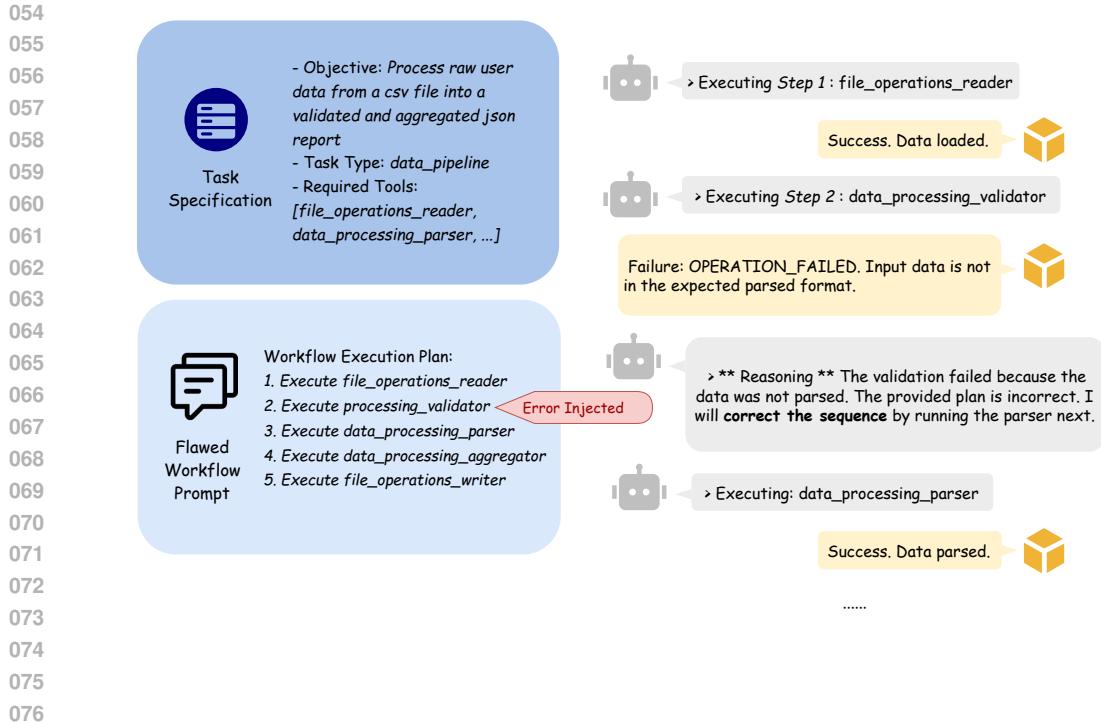
009 We introduce PILOT-Bench, a benchmark that evaluates LLM workflow execution
010 under simulated realistic conditions of instruction quality variability and tool ex-
011 ecution uncertainty. **Unlike existing benchmarks that encounter these challenges
012 incidentally, our work makes uncertainty the primary focus of systematic study.**
013 The benchmark incorporates three key aspects: (1) modeling of probabilistic tool
014 behaviors through parameterized error models that **simulate** real-world API failure
015 patterns, (2) **provision of MDP-derived workflows that maximize expected success
016 rates, and (3) systematic evaluation of model robustness through controlled pertur-
017 bations of workflow instruction quality.** Our construction pipeline generates 5,040
018 tasks from a tool library of 30 APIs. The evaluation conducted across widely used
019 large language models under conditions of probabilistic tool failures and varying
020 instruction quality reveals notable performance differences. Specifically, **MDP-
021 optimal workflow prompts achieve an average success rate of 62.1%, Chain-of-
022 Thought prompts yield an average success rate of 50.8%, and flawed workflow
023 prompts result in an average success rate of 54.3%.** Our benchmark is available at
024 <https://github.com/PilotBenchAnonymous/PilotBench>.
025

026 1 INTRODUCTION 027

028 Large language models (LLMs) have shown capabilities in reasoning, planning, and tool utilization
029 across diverse domains (Brown et al., 2020; Achiam et al., 2023). These models have demon-
030 strated proficiency in applications that combine natural language understanding with tool orches-
031 tration (Schick et al., 2023; Qin et al., 2023; Parisi et al., 2022). Recent advances in agent-based
032 systems have further expanded the scope of tool-use applications (Wang et al., 2023; Xi et al., 2023;
033 Sumers et al., 2023). Workflow execution represents one area where LLMs must navigate sequences
034 of tool interactions while maintaining coherence and achieving specified objectives, with applica-
035 tions ranging from software development (Fried et al., 2023; Nijkamp et al., 2022) to scientific
036 computing (Lewkowycz et al., 2022).
037

038 **When LLMs are deployed for workflow execution in production environments, they encounter three
039 primary categories of challenges: (1) Tool reliability issues, where APIs exhibit probabilistic fail-
040 ures including timeouts, service outages, validation errors, and resource limitations that require
041 adaptive error recovery strategies; and the model lacks higher-level information about tool reliabil-
042 ity, for example, an LLM may not know the specific cause of an API interruption. (2) Instruction
043 quality variations, where users provide instructions that may be incomplete, ambiguous, contra-
044 dictory, or contain logical inconsistencies due to limited domain knowledge or communication con-
045 straints; and (3) Complexity challenges, where processes involve intricate tool dependencies, dy-
046 namic multi-agent coordination, and other highly complex tasks, and the number of dialogue turns
047 or token limits restricts the model’s exploration behavior.**
048

049 Recent research has produced various tool-use benchmarks that evaluate LLMs’ capacity for API
050 interaction, multi-step reasoning, and workflow execution (Zhuang et al., 2023; Qin et al., 2023; Li
051 et al., 2023; Patil et al., 2023), with specialized benchmarks emerging for mathematics (Hendrycks
052 et al., 2021; Cobbe et al., 2021), code generation (Chen et al., 2021; Austin et al., 2021), and multi-
053 modal reasoning (Lu et al., 2022). These frameworks have made significant contributions to under-
standing LLM capabilities (Zhou et al., 2022; Wei et al., 2022), **establishing important baselines for
tool-use performance under controlled conditions.**



077 Figure 1: This figure demonstrates the PILOT-Bench setup. On the left, we provide task specifications, a tool library, and workflow prompts with injected errors. On the right, the LLM is expected to identify errors in the provided workflow, and execute with correction.

082 We introduce PILOT-Bench, a benchmark that systematically evaluates LLM workflow execution
083 capabilities under realistic deployment conditions **involving tool reliability issues and instruction**
084 **quality variations** (Liu et al., 2023; Jimenez et al., 2023). While existing benchmarks (Qin et al.,
085 2023; Li et al., 2023; Huang et al., 2023) may encounter probabilistic tool failures and instruction
086 variability as natural consequences of working with real-world APIs and diverse instruction gen-
087 eration, these works typically filter and curate APIs or instructions to minimize such issues. Our
088 work makes two distinct contributions: Firstly, we make these uncertainties the primary focus of
089 systematic study rather than attempting to filter them out. We deliberately introduce controlled
090 perturbations through systematically generated flawed workflow instructions and probabilistic er-
091 ror models. This creates environments that require adaptive strategies, allowing us to measure the
092 model’s ability to recover from API errors and adaptively replan workflows to complete tasks.

093 Secondly, we provide analytically derived optimal workflows through MDP optimization. Under re-
094 alistic deployment constraints, there exist three theoretical upper bounds for success rates: (i) 100%
095 may not be achievable due to round/retry limits; (ii) the upper bound achieved by a policy know-
096 ing tool call procedures and able to anticipate tool errors (e.g., knowing the random seed of every
097 tool operation); (iii) the upper bound achieved by a policy knowing tool call procedures but only
098 aware of the probability of tool failure. Our MDP formulation provides a tractable approximation
099 to bound (iii), computing workflows that maximize expected success rates by reasoning over known
100 reliability statistics.

101 PILOT-Bench consists of three primary components: task specifications spanning multiple types
102 and complexity levels, a tool registry modeling 30 APIs with probabilistic error behaviors and de-
103 pendency constraints, and multi-variant workflow guidance including both optimal execution plans
104 and perturbed variants. Each task is accompanied by four distinct prompt types—baseline, Chain-
105 of-Thought, MDP-optimal workflow prompt, and flawed workflow prompt—designed to simulate
106 low-quality prompts in real scenarios.

107 Our evaluation framework shows model families and size exhibit performance differences in current
108 LLMs when confronted with realistic workflow environments. Assessments suggest performance

108 changes when models encounter flawed instructions and probabilistic tool failures, with success
 109 rates that may correlate with both workflow instruction quality and tool reliability levels. These
 110 findings could inform approaches for improving LLM robustness in practical workflow execution
 111 scenarios.

112 The primary contributions of this work are multifold:
 113

- 114 **1. Benchmark Design and Automated Construction:** We present PILOT-Bench, an evalua-
 115 tion framework that [systematically](#) assesses LLM workflow execution capabilities under
 116 instruction variability and tool uncertainty [through automated task generation from struc-
 117 tured tool libraries](#).
- 118 **2. MDP-Based Workflow Generation Framework:** We develop a Markov Decision Pro-
 119 cess (MDP) framework that generates [theoretically](#) optimal execution workflows maximiz-
 120 ing expected success rates, along with seven types of [systematically perturbed variants](#),
 121 [enabling controlled evaluation of model robustness to instruction quality variations](#).
- 122 **3. Evaluation Findings:** Our evaluation reveals that models exhibit dramatically different
 123 robustness patterns when confronted with flawed instructions: GPT-4o-mini maintains rel-
 124 atively stable performance (optimal: 67.7%, flawed: 62.2%, a 5.5 percentage point drop),
 125 while Gemini-2.5-Flash shows substantial degradation (optimal: 60.1%, flawed: 20.0%,
 126 a 40.1 percentage point drop). We also observe emergent abilities in workflow execution
 127 where multi-step tool-use proficiency appears suddenly at certain parameter thresholds.
- 128 **4. Real-World API Integration:** We extend PILOT-Bench with a real-world task set that
 129 integrates live public APIs, serving as a complementary evaluation component that directly
 130 assesses model performance on actual API interactions. Real-world experiments reveal
 131 patterns consistent with the observations from the simulated API experiments.

133 The remainder of this paper is organized as follows. Section 2 details the core components, evalua-
 134 tion methodology, and benchmark statistics. Section 3 describes our automated construction pipeline
 135 and MDP-based workflow generation methods. Section 4 presents experimental results and analysis
 136 of current LLM capabilities. Section 5 reviews related benchmarks.

138 2 BENCHMARK SETUP

140 In this section, we present the setup of PILOT-Bench. We begin by describing the core components
 141 of the benchmark data and their organization, followed by the task types and evaluation methodology
 142 used to assess agent performance. A figure demonstrating the benchmark setup is provided in Figure
 143 1.

145 2.1 BENCHMARK DATA ORGANIZATION

147 The PILOT-Bench dataset is structured around three primary components: task specifications, tool
 148 registry, and reference workflows.

150 **Task Specifications.** The benchmark comprises 5,040 unique tasks organized across multiple di-
 151 mensions of type and complexity. Each task specification includes a natural language description,
 152 structured input/output requirements, a list of required tools, execution constraints, and comprehen-
 153 sive metadata. Detailed examples of task specifications are provided in Appendix B.1.

154 **Tool Registry.** The benchmark includes a comprehensive Tool Registry modeling 30 canonical
 155 software APIs with probabilistic behavior. Tools are systematically organized into six functional
 156 categories: data_processing, file_operations, network, computation, integration, and utility, with five
 157 tools per category ensuring balanced coverage.

158 Each tool definition specifies functional parameters with type constraints, structured return schemas,
 159 and explicit error models that enumerate possible failure modes. The system models five primary
 160 failure types: input validation failures (`INVALID_INPUT`), operational failures (`OPERATION_-
 161 FAILED`), timeout conditions (`TIMEOUT`), calculation errors (`CALCULATION_ERROR`), and re-
 source overflow conditions (`OVERFLOW`). These error types reflect common failure patterns in real-

162 world scenarios. Representative tool examples and detailed specifications are provided in Appendix
 163 B.1.

164 **Reference Workflows.** Each task is accompanied by four distinct prompt types (detailed in Section
 165 3.3): baseline, Chain-of-Thought, MDP-optimal workflow prompt (optimal with respect to our MDP
 166 reward function), and flawed workflow prompt with systematic error injection. These variants enable
 167 evaluation of both instruction-following fidelity under high-quality instructions and robustness under
 168 flawed instruction quality.

170 **2.2 TASK EXECUTION AND EVALUATION**

172 **Execution Environment.** PILOT-Bench employs a simulated execution environment that provides
 173 realistic tool behavior. When agents invoke tools, a probabilistic simulator calculates success rates
 174 based on tool dependencies, execution history, and failure patterns, then generates appropriate suc-
 175 cess or failure responses. The simulator implements a base success rate of 0.8. Complete imple-
 176 mentation details are provided in Appendix B.3.

177 **Evaluation Methodology.** Task outcomes are categorized into three distinct levels of success. A
 178 task is considered `full_success` if all specified tools are executed correctly and in the proper
 179 sequence. `Partial_success` reflects substantial task completion that satisfies most requirements
 180 but does not achieve perfect execution. `Failure` indicates that task completion is insufficient due
 181 to critical tool failures or breakdowns in execution.

182 The evaluation framework relies on four primary assessment criteria. *Required Tools Coverage* mea-
 183 sures the proportion of tools successfully executed from the task’s required tool list, with `full_-`
 184 `success` requiring 100% coverage. *Sequence Correctness* assesses whether tools are executed in
 185 the exact order dictated by task dependencies. *Output Generation* verifies the successful execution
 186 of output-producing tools (e.g., writers, exporters, savers) as evidence of meaningful task comple-
 187 tion. *Explicit Completion Signals* examines the conversation history for indications from the LLM
 188 that the task has been completed. Achieving `full_success` requires success in all four of these
 189 criteria.

190 Additionally, two considerations are also applied. *Minimum Tool Execution*: This defines task-type-
 191 specific thresholds for considering partial success. A task may be rated as `partial_success`
 192 if it meets at least two conditions, such as exceeding minimum tool execution coverage while pro-
 193 ducing the expected outputs. *Termination Conditions*: The evaluation also accounts for premature
 194 task termination, such as when a task experiences an excessive number of consecutive tool failures
 195 or becomes trapped in a repetitive loop, which typically results in a `failure` rating. Complete
 196 implementation details are provided in Appendix B.4.

198 **2.3 BENCHMARK STATISTICS**

200 PILOT-Bench contains 5,040 unique tasks systematically distributed across multiple dimensions to
 201 ensure balanced coverage of workflow scenarios. The benchmark incorporates 30 canonical tools
 202 organized into six functional categories, providing a controlled yet diverse environment for eval-
 203 uating LLM workflow execution capabilities. Here we present some statistics about tasks and tools.
 204 For more detailed statistical information, please see Appendix B.1.

205 **Task Distribution.** We classify tasks using two approaches: task type classification and complex-
 206 ity classification. Task types include five categories: complex validation pipeline tasks, complex
 207 network integration tasks, basic file processing tasks, advanced computation pipelines, and simple
 208 data transformation tasks. Complexity classification spans three levels: easy, medium, and hard.
 209 Complex validation pipeline tasks (1,520 instances, 30.2%) form the largest category, followed by
 210 complex network integration tasks (1,360 instances, 27.0%), basic file processing tasks (1,200 in-
 211 stances, 23.8%), advanced computation pipelines (640 instances, 12.7%), and simple data trans-
 212 formation tasks (320 instances, 6.3%). Tasks span three complexity levels: easy (1,520 instances,
 213 30.2%), medium (2,880 instances, 57.1%), and hard (640 instances, 12.7%). The distributions for
 214 both classification approaches are shown in Figure 2a and 2b.

215 **Tool Library Composition.** We organize tools into six categories: computation, data processing,
 file operations, integration, network, and utility, with each category containing five tools, as shown in

216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
Figure 2c. We also conduct a statistical analysis of tool-required parameters, with results presented in Figure 2d. The most common required parameter is `options`, which is required by 25 tools.

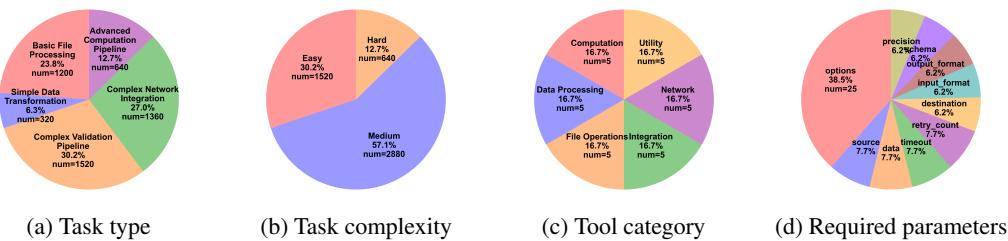


Figure 2: PILOT-Bench statistics showing (a) task type distribution, (b) task complexity distribution, (c) tool category distribution, and (d) required parameter distribution.

3 BENCHMARK CONSTRUCTION

PILOT-Bench employs an automated construction pipeline that generates diverse workflow tasks from a systematic tool library. The construction process consists of three main stages: tool library generation, task creation, and prompt generation. This automated approach enables scalable benchmark expansion while maintaining task quality and diversity.

3.1 TOOL LIBRARY GENERATION

The tool library is constructed using a two-layer categorization system that balances systematic coverage with realistic workflow dependencies. The foundation consists of a category-operation matrix spanning 6 functional categories (data_processing, file_operations, network, computation, integration, utility) and 5 operations each, yielding 30 distinct tool types. Each tool follows the `{category}_{operation}` naming convention and includes standardized parameter templates, return schemas, and error handling specifications.

The second layer introduces semantic operation types that group tools by their workflow roles: sources (readers, fetchers), processors (parsers, transformers, analyzers), aggregators, outputs (writers, posters), and utilities. This semantic grouping enables dependency modeling where processors depend on sources, aggregators depend on processors, and outputs depend on aggregators. More detailed information can be found in Appendix C.1.

3.2 TASK GENERATION METHODOLOGY

Tasks are automatically generated through semantic matching between predefined operation sequences and the tool library. The system defines five task types: `basic_file_processing`, `simple_data_transformation`, `complex_validation_pipeline`, `complex_network_integration`, and `advanced_computation_pipeline`. Each task type follows a standard operation sequence, such as `['read', 'validate', 'transform', 'aggregate', 'write']` for complex validation pipelines.

The generation process uses RAG-based semantic matching to map operation steps to appropriate tools. For example, a `'read'` operation can match to `file_operations_reader` or `network_fetcher` based on semantic similarity. Task variation is introduced through multiple tool choices per operation and different templates. An optional LLM enhancement phase improves task descriptions while preserving logical consistency, ensuring both natural language quality and structural integrity. More detailed information can be found in Appendix C.2.

3.3 WORKFLOW PROMPT GENERATION

Each task has four distinct prompt types to evaluate different aspects of agent capability. Table 1 provides a comprehensive comparison of these prompt variants. Baseline prompts contain essential task information including description, input/output specifications, and tool usage instructions. Chain-of-Thought (CoT) prompts enhance baseline prompts with explicit CoT instructions that encourage

270
271
272 Table 1: Comparison of prompt types in PILOT-Bench.
273
274
275
276
277
278
279
280
281
282
283
284
285

Prompt Type	Base Components	Additional Elements	Evaluation Focus
Baseline	Task description, Input/output specs, Tool instructions	None	Basic instruction following
Chain-of-Thought	Task description, Input/output specs, Tool instructions	Explicit reasoning instructions	Reasoning and planning capabilities
MDP-Optimal Workflow	Task description, Input/output specs, Tool instructions	Detailed execution plan	Workflow adherence
Flawed Workflow	Task description, Input/output specs, Tool instructions	Perturbed execution plan with systematic errors	Error detection and robustness

286
287 step-by-step analysis. MDP-Optimal workflow prompts incorporate detailed execution plans derived
288 from MDP formulations that account for tool dependencies and success probabilities.
289

290 The MDP-optimal workflow prompts are generated using a MDP framework that formally defines
291 optimality in terms of expected cumulative reward for tool sequence selection under uncertainty. Our
292 MDP formulation uses a composite state representation capturing tool execution states, progress
293 tracking, etc. The action space consists of structured tool invocations. The system implements a
294 two-phase adaptive reward strategy: an initial coverage-focused phase prioritizing tool discovery
295 and usage, followed by a sequence-optimized phase emphasizing execution order and efficiency.
296 Policy optimization employs Proximal Policy Optimization (PPO) with Transformer-based neural
297 networks, mixed-precision training, and curriculum learning across five difficulty stages. The trained
298 policy generates tool sequences that are optimal with respect to the learned reward function and
299 state transition probabilities, producing workflows with maximized expected success rates given the
300 MDP’s modeling assumptions. More details can be found in Appendix D. Flawed workflow prompts
301 are systematically generated by introducing controlled perturbations to MDP-optimal workflows
302 across seven categories: sequential ordering errors, tool misuse, parameter configuration errors,
303 missing critical steps, redundant operations, logic discontinuity, and semantic drift. More detailed
304 information can be found in Appendix C.3.

305 3.4 REAL-WORLD API INTEGRATION

306 To complement our simulated environment, we extend PILOT-Bench with a real-world task set
307 integrating live public APIs that directly assess practical deployment capabilities.
308

309 **API Selection and Characterization.** We source candidate APIs from the `public-apis` GitHub
310 repository¹. To identify APIs exhibiting probabilistic behaviors aligned with our benchmark design,
311 we conduct empirical reliability assessment: each candidate API is invoked 20 times to measure
312 success rate and response latency distributions. APIs are selected based on two criteria: (1) *Success
313 rate variability*—non-deterministic behavior with occasional failures (timeouts, rate limiting,
314 service interruptions), and (2) *Latency variability*—substantial variation in response times reflecting
315 real-world network dynamics. This yields APIs naturally exhibiting stochastic behaviors that mirror
316 our simulated error modes.

317 **Task Construction.** We design 8 sequential workflow tasks based on 23 selected APIs’ functional-
318 ities and output characteristics. Tasks are constructed by analyzing each API’s input re-
319 quirements, output schemas, and semantic capabilities to create realistic multi-step workflows. As
320 an example, the `content_creation_task` requires LLMs to sequentially call four real-world
321 APIs—fetching a random fact, a joke, a programming quote, and a stoic quote—then compile them
322 into a social media post draft.

323
324 ¹<https://github.com/public-apis/public-apis>

324 **Framework Integration.** We align real-world components with our existing infrastructure through:
 325 (1) constructing MCP-compliant tool registrations matching our simulated tool library schema (Sec-
 326 tion 3.1), including standardized parameters, return schemas, and error classifications, and (2) for-
 327 matting tasks to match our task specification structure (Section 3.2). This enables uniform workflow
 328 prompt generation and evaluation methodology across simulated and real-world components. The
 329 specific tools and tasks are updated to our benchmark repository².
 330

331 4 EXPERIMENTS

333 4.1 EXPERIMENTAL SETUP

335 We evaluate several models spanning proprietary LLMs (GPT-4o-mini, O3-0416-Global, etc.) and
 336 open-source LLMs (DeepSeek-V3, Qwen2.5-32B, etc.). Each model is tested with 4 prompt variants
 337 per task (baseline prompt, Chain-of-Thought prompt, MDP-optimal workflow prompt, and flawed
 338 workflow prompt). We use a base success rate $\rho_{\text{base}} = 0.8$ for probabilistic tool execution.

339 The framework employs an interactive multi-turn execution environment with up to 10 con-
 340 versational turns per task. Key implementation details include: (1) **Interaction protocol and**
 341 **execution:** Models use `<tool_search>query</tool_search>`, `<tool_info>tool_`
 342 `</tool_info>`, and `<tool_call>tool_name</tool_call>` syntax; the system en-
 343 forces one-tool-per-turn execution with dependency management. (2) **Automated feedback and**
 344 **parameter handling:** The system generates feedback messages with execution results, error in-
 345 formation, and progress indicators; format detection reminders are provided when responses lack
 346 proper syntax. (3) **Error simulation:** Tool failures follow predefined error templates specific to
 347 each tool’s MCP specification, including modes such as `TIMEOUT`, `DEPENDENCY_ERROR`, and
 348 `INVALID_INPUT`. For details, see Appendix B. In addition, we provide testing interaction exam-
 349 ples in Appendix A.

350 Our experimental analysis of LLM workflow execution capabilities demonstrates:

- 351 • **Different robustness patterns to instruction quality:** Models exhibit dramatically dif-
 352 ferent robustness patterns when confronted with flawed instructions. For example, GPT-
 353 4o-mini maintains relatively stable performance across instruction quality variations (optimal:
 354 67.7%, flawed: 62.2%, a 5.5 percentage point drop), while Gemini-2.5-Flash shows
 355 substantial degradation (optimal: 60.1%, flawed: 20.0%, a 40.1 percentage point drop)
 356 (Section 4.3).
- 357 • **Emergent abilities of workflow execution:** Through experiments on the Qwen2.5 series,
 358 we observe emergent abilities in workflow execution, where the model’s multi-step work-
 359 flow execution ability appears suddenly at certain parameter thresholds rather than scaling
 360 smoothly (Section 4.4).
- 361 • **Real-world validation:** We extend our evaluation with a real-world test set integrating live
 362 public APIs. Real-world experiments reveal patterns consistent with the observations from
 363 the simulated API experiments (Section 4.5 and Appendix E.3).

365 4.2 OVERALL PERFORMANCE RESULTS

367 Table 2 presents the performance results using baseline, Chain-of-Thought, and MDP-optimal work-
 368 flow prompts. When the models are presented with the MDP-optimal prompt, success rates range
 369 from 56.8% to 67.7% across both proprietary and open-source models. Among proprietary models,
 370 GPT-4o-mini achieves a 67.7% success rate, followed by GPT-5-mini (60.7%) and Gemini-2.5-Flash
 371 (60.1%). Open-source models show competitive performance, with Llama-3.3-70B reaching 66.1%.

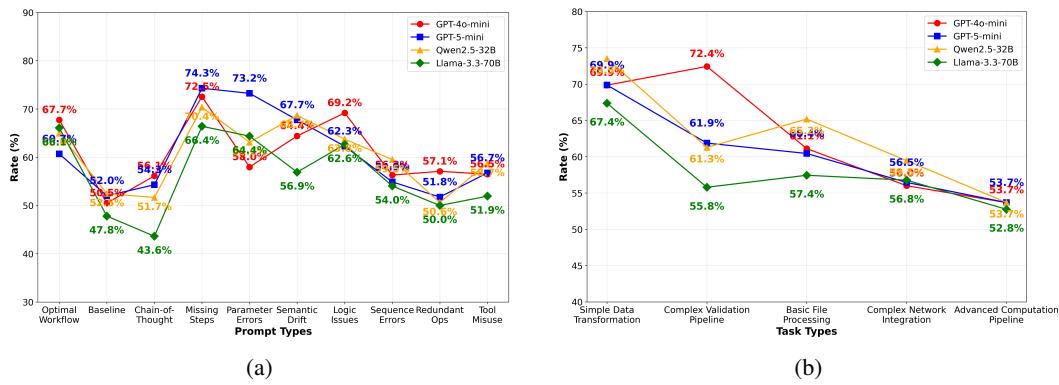
372 4.3 SENSITIVITY ANALYSIS

374 We conduct sensitivity analysis to examine model performance across two dimensions: instruction
 375 quality variation and task complexity progression. This analysis evaluates performance under the
 376 MDP-optimal workflow and flawed workflow prompts, and five task complexity levels.

377 ²<https://github.com/PilotBenchAnonymous/PilotBench>

378 Table 2: Performance: Baseline, Chain-of-Thought, and MDP-Optimal Workflow prompting.
379

380 381 382 383 384 385 386 387 388 389 390	Model	Baseline			Chain-of-Thought			Optimal Workflow		
		Full	Partial	Fail	Full	Partial	Fail	Full	Partial	Fail
GPT-4o-mini	50.5	46.5	3.0	56.1	43.9	0.0	67.7	31.2	1.1	
O3-0416-Global	52.7	44.9	2.4	48.9	45.6	5.6	58.5	35.1	6.4	
Gemini-2.5-Flash	54.3	44.5	1.1	51.1	44.7	4.2	60.1	36.7	3.3	
GPT-5-mini	52.0	46.0	2.0	54.3	45.7	0.0	60.7	35.5	3.8	
Llama-3.3-70B	47.8	42.5	9.6	43.6	43.6	12.7	66.1	30.9	3.0	
Qwen2.5-32B	52.5	43.8	3.7	51.7	45.0	3.3	65.0	31.9	3.1	
DeepSeek-V3	50.0	50.0	0.0	50.0	46.9	3.1	56.8	39.0	4.2	
Avg	51.4	45.5	3.1	50.8	45.1	4.1	62.1	34.3	3.6	

405 Figure 3: Performance analysis across representative models: (a) Prompt robustness analysis across
406 7 categories of flawed workflow prompts, and (b) model performance across different task types.
407
408

409 **Instructions Quality Sensitivity.** Table 3 shows that MDP-optimal workflow prompts achieves
410 higher average success rate (62.1%) compared to flawed workflow prompts (54.3%). Besides,
411 models exhibit dramatically different robustness patterns when confronted with flawed instructions.
412 Specifically, GPT-4o-mini maintains relatively stable performance with only a 5.5 percentage point
413 drop (optimal: 67.7%, flawed: 62.2%), suggesting implicit error correction capabilities. In
414 contrast, Gemini-2.5-Flash shows substantial performance degradation with a 40.1 percentage point
415 drop (optimal: 60.1%, flawed: 20.0%). This indicates that tolerance to flawed instructions repre-
416 sents a distinct capability dimension. To analyze the specific failure modes, we examine seven types
417 of systematic perturbations in Figure 3a, revealing that advanced models are particularly resilient to
418 ordering and parameter errors but more vulnerable to semantic drift.

419 **Task Complexity Sensitivity.** Figure 3b demonstrates consistent performance degradation pat-
420 terns as task complexity increases. Representative models show performance decline from simple
421 content analysis tasks to complex computation pipelines: GPT-4o-mini (72.4% to 53.7%), GPT-5-
422 mini (69.9% to 53.7%), Qwen2.5-32B (73.5% to 53.7%), and Llama-3.3-70B (67.4% to 52.8%).
423 This degradation reflects the increasing cognitive demands of workflow execution, where advanced
424 computation pipelines require more sophisticated reasoning about tool dependencies and execution
425 chains compared to basic file processing tasks.

426 4.4 SCALING ANALYSIS

427 We conducted additional experiments on the Qwen2.5 series to investigate the relationship between
428 model size and workflow execution capabilities. The Qwen2.5 series exhibits non-linear scaling
429 patterns (Table 4), with performance varying substantially: 0.5% (3B), 63.5% (7B), 65.0% (32B),
430 and 65.0% (72B).

The substantial jump from 3B to 7B (63.0 percentage point increase) suggests the emergence of basic tool comprehension capabilities around this scale, while performance plateaus from 32B to 72B indicate diminishing returns. This pattern reflects emergent abilities in workflow execution, where tool-use proficiency appears suddenly at certain parameter thresholds rather than scaling smoothly. This observation aligns with discussions such as (Berti et al., 2025).

Table 3: Performance comparison: Optimal vs. Flawed Workflow prompting.

Model	Optimal Workflow			Flawed Workflow		
	Full	Partial	Fail	Full	Partial	Fail
GPT-4o-mini	67.7	31.2	1.1	62.2	34.6	3.2
O3-0416-Global	58.5	35.1	6.4	53.8	39.2	7.0
Gemini-2.5-Flash	60.1	36.7	3.3	20.0	12.8	67.3
GPT-5-mini	60.7	35.5	3.8	63.5	36.1	0.4
Llama-3.3-70B	66.1	30.9	3.0	59.5	36.2	4.4
Qwen2.5-32B	65.0	31.9	3.1	62.9	35.9	1.2
DeepSeek-V3	56.8	39.0	4.2	58.4	39.7	1.9
Avg	62.1	34.3	3.6	54.3	33.5	12.2

Table 4: Qwen2.5 series scaling analysis with detailed performance metrics.

Model Size	Full Success Rate	Partial Success Rate	Failure Rate
Qwen2.5-3B	0.5%	0.5%	99.1%
Qwen2.5-7B	63.5%	30.2%	6.3%
Qwen2.5-32B	65.0%	31.9%	3.1%
Qwen2.5-72B	65.0%	32.0%	3.0%

4.5 RESULTS ON REAL-WORLD TASK SET

To assess the transferability of our simulation findings, we also evaluate the models on real-world tasks. Table 5 presents performance across 8 tasks using 23 live public APIs. Consistent with our simulation results, models exhibit different robustness patterns when confronted with flawed instructions. Specifically, GPT-4o-mini shows moderate degradation (optimal: 42.1%, flawed: 34.3%, a 7.8 percentage point drop), while Gemini-2.5-Flash exhibits more substantial degradation (optimal: 55.3%, flawed: 34.1%, a 21.2 percentage point drop). We present the specific distribution of API error rates and types occurred in our experiments in Section E.4.

Table 5: Real-word Task Set Model Performance

Model	Optimal Workflow			Flawed Workflow		
	Full	Partial	Fail	Full	Partial	Fail
GPT-4o-mini	42.1	38.6	19.3	34.3	34.3	31.4
Gemini-2.5-Flash	55.3	41.2	3.5	34.1	33.5	32.4
GPT-5-mini	47.6	45.2	7.1	36.6	36.3	27.0
Llama-3.3-70B	39.6	27.1	33.3	25.4	25.4	49.3
Qwen2.5-32B	55.3	44.7	0.0	42.3	38.8	18.9
DeepSeek-V3	40.2	39.3	20.5	36.0	35.4	28.6
Avg	46.7	39.4	13.9	34.8	34.0	31.3

5 RELATED WORK

Recent studies on large language models (LLMs) have produced various benchmarks and frameworks for evaluating tool-use reasoning, workflow execution, and LLM robustness in complex tasks.

486 Table 6: Comparison of workflow-oriented LLM benchmarks. Comparison involves 5 aspects:
 487 **Planning**: benchmark requires the LLM to design a sequence of tool calls. **Tool Choice**: benchmark
 488 measures correct selection of the appropriate tool/API. **Tool Call**: benchmark checks syntactic and
 489 semantic correctness of each invocation. **Multi-Step**: tasks demand two or more consecutive calls.
 490 **MCP Protocol**: tools are specified with the Model Context Protocol (or an equivalent structured
 491 schema).

Benchmark	Planning	Tool Choice	Tool Call	Multi-Step	MCP Protocol
ToolQA(Zhuang et al., 2023)	No	Yes	No	No	No
ToolBench(Qin et al., 2023)	Yes	Yes	Yes	Yes	No
MetaTool(Wang et al., 2024)	No	Yes	No	No	No
ToolAlpaca(Tang et al., 2023)	No	No	Yes	No	No
API-Bank(Li et al., 2023)	Yes	Yes	Yes	Yes	Yes
SOP-Bench (Nandi et al., 2025)	Yes	Yes	Yes	Yes	No
Multi-Mission Tool Bench (Yu et al., 2025)	Yes	Yes	Yes	Yes	No
TheAgentCompany (Xu et al., 2024)	Yes	Yes	No	Yes	No
Ours (this work)	Yes	Yes	Yes	Yes	Yes

503 For example, ToolQA (Zhuang et al., 2023) analyzes tool-use reasoning by differentiating between
 504 knowledge-based and tool-reliant questions but is limited to single tool invocations.

505 Some advanced benchmarks attempt to evaluate LLMs in more realistic scenarios. [ToolBench \(Qin et al., 2023\)](#) extends evaluations to a broader set of APIs and enable both single- and multi-tool tasks
 506 with curated APIs and instructions. MetaTool (Wang et al., 2024) uses meta-task augmentation
 507 to improve tool-use knowledge transfer. ToolAlpaca (Tang et al., 2023) studies generalization in
 508 smaller models, mainly focusing on simple tool use. Broader benchmarks such as API-Bank (Li
 509 et al., 2023), SOP-Bench (Nandi et al., 2025), and Multi-Mission Tool Bench (Yu et al., 2025)
 510 examine planning, procedural adherence, and adaptability, yet mostly emphasize dialog, industrial
 511 SOPs, or changing missions. TheAgentCompany (Xu et al., 2024) studies real-world digital tasks
 512 but does not deeply examine workflow quality and its effect on decision-making. We present a
 513 detailed comparison between our benchmark and others in Table 6.

516 6 CONCLUSION

518 We introduce PILOT-Bench, a benchmark that systematically evaluates LLM workflow execution
 519 capabilities under realistic deployment conditions involving tool reliability issues and instruction
 520 quality variations. While existing benchmarks may encounter these challenges incidentally, our
 521 work makes uncertainties the primary focus of systematic study, representing a complementary eval-
 522 uation dimension to existing capability-focused benchmarks.

523 Our benchmark employs an automated construction pipeline that generates 5,040 tasks from a tool
 524 library of 30 APIs, incorporating probabilistic error models and MDP-derived workflows that repre-
 525 sent theoretically optimal strategies maximizing expected success rates. Through systematic eval-
 526 uation across model families, we reveal that models exhibit dramatically different robustness patterns
 527 when confronted with flawed instructions. We also observe emergent abilities in workflow execu-
 528 tion where multi-step tool-use proficiency appears suddenly at certain parameter thresholds rather
 529 than scaling smoothly. These findings provide insights into LLM robustness under uncertainty and
 530 could inform approaches for improving model reliability in practical workflow execution scenar-
 531 os. Additionally, our real-world API integration validates these simulation findings, demonstrating
 532 consistent patterns across both controlled and live deployment environments. We believe this work
 533 opens new avenues for understanding and improving LLM performance in production settings where
 534 uncertainty is inevitable.

536 REPRODUCIBILITY STATEMENT

538 To ensure the reproducibility of our work, all components of the PILOT-Bench benchmark are pub-
 539 licly available. The complete tool library, the full set of 5,040 generated task specifications, and the
 source code for our automated construction pipeline are accessible at our anonymous GitHub repos-

540
541
542
543
544
545
546
547
548
549
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
itory: <https://github.com/PilotBenchAnonymous/PilotBench>. Furthermore, we have provided extensive details of our methodology in the appendices.

REFERENCES

- Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.
- Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Mich Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.
- Leonardo Berti, Flavio Giorgi, and Gjergji Kasneci. Emergent abilities in large language models: A survey. *arXiv preprint arXiv:2503.05788*, 2025.
- Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.
- Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. Incoder: A generative model for code infilling and synthesis. In *International Conference on Learning Representations*, 2023.
- Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.
- Yue Huang, Jiawen Shi, Yuan Li, Chenrui Fan, Siyuan Wu, Qihui Zhang, Yixin Liu, Pan Zhou, Yao Wan, Neil Zhenqiang Gong, et al. Metatool benchmark for large language models: Deciding whether to use tools and which to use. *arXiv preprint arXiv:2310.03128*, 2023.
- Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.
- Aitor Lewkowycz, Anders Andreassen, David Dohan, Ethan Dyer, Henryk Michalewski, Vinay Ramalesh, Ambrose Slone, Cem Anil, Imanol Schlag, Theo Gutman-Solo, et al. Solving quantitative reasoning problems with language models. *arXiv preprint arXiv:2206.14858*, 2022.
- Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. Api-bank: A comprehensive benchmark for tool-augmented llms. *arXiv preprint arXiv:2304.08244*, 2023.
- Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.
- Pan Lu, Swaroop Mishra, Tanglin Xia, Liang Qiu, Kai-Wei Chang, Song-Chun Zhu, Oyvind Tafjord, Peter Clark, and Ashwin Kalyan. Learn to explain: Multimodal reasoning via thought chains for science question answering. *Advances in Neural Information Processing Systems*, 35:2507–2521, 2022.
- Subhrangshu Nandi, Arghya Datta, Nikhil Vichare, Indranil Bhattacharya, Huzefa Raja, Jing Xu, Shayan Ray, Giuseppe Carenini, Abhi Srivastava, Aaron Chan, et al. Sop-bench: Complex industrial sops for evaluating llm agents. *arXiv preprint arXiv:2506.08119*, 2025.

- 594 Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese,
 595 and Caiming Xiong. Codegen: An open large language model for code with multi-turn program
 596 synthesis. *arXiv preprint arXiv:2203.13474*, 2022.
- 597
- 598 Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. *arXiv preprint*
 599 *arXiv:2205.12255*, 2022.
- 600 Shishir G Patil, Tianjun Zhang, Xin Wang, and Joseph E Gonzalez. Gorilla: Large language model
 601 connected with massive apis. *arXiv preprint arXiv:2305.15334*, 2023.
- 602
- 603 Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru
 604 Tang, Bill Qian, et al. Toolllm: Facilitating large language models to master 16000+ real-world
 605 apis. *arXiv preprint arXiv:2307.16789*, 2023.
- 606 Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro,
 607 Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can
 608 teach themselves to use tools. *Advances in Neural Information Processing Systems*, 36:68539–
 609 68551, 2023.
- 610 Theodore Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L Griffiths. Cognitive architec-
 611 tures for language agents. *arXiv preprint arXiv:2309.02427*, 2023.
- 612
- 613 Qiaoyu Tang, Ziliang Deng, Hongyu Lin, Xianpei Han, Qiao Liang, Boxi Cao, and Le Sun. Toolal-
 614 pac: Generalized tool learning for language models with 3000 simulated cases. *arXiv preprint*
 615 *arXiv:2306.05301*, 2023.
- 616 Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan,
 617 and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models.
 618 *arXiv preprint arXiv:2305.16291*, 2023.
- 619
- 620 Xiaohan Wang, Dian Li, Yilin Zhao, Hui Wang, et al. Metatool: Facilitating large language models
 621 to master tools with meta-task augmentation. *arXiv preprint arXiv:2407.12871*, 2024.
- 622 Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny
 623 Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in*
 624 *neural information processing systems*, 35:24824–24837, 2022.
- 625
- 626 Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe
 627 Wang, Senjie Jin, Enyu Zhou, et al. The rise and potential of large language model based agents:
 628 A survey. *arXiv preprint arXiv:2309.07864*, 2023.
- 629 Frank F Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z Wang,
 630 Xuhui Zhou, Zhitong Guo, Murong Cao, et al. Theagentcompany: benchmarking llm agents on
 631 consequential real world tasks. *arXiv preprint arXiv:2412.14161*, 2024.
- 632
- 633 Peijie Yu, Yifan Yang, Jinjian Li, Zelong Zhang, Haorui Wang, Xiao Feng, and Feng Zhang. Multi-
 634 mission tool bench: Assessing the robustness of llm based agents through related and dynamic
 635 missions. *arXiv preprint arXiv:2504.02623*, 2025.
- 636 Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuur-
 637 mans, Claire Cui, Olivier Bousquet, Quoc Le, et al. Least-to-most prompting enables complex
 638 reasoning in large language models. *arXiv preprint arXiv:2205.10625*, 2022.
- 639
- 640 Yuchen Zhuang, Yue Yu, Kuan Wang, Haotian Sun, and Chao Zhang. Toolqa: A dataset for llm
 641 question answering with external tools. *Advances in Neural Information Processing Systems*, 36:
 642 50117–50143, 2023.
- 643
- 644
- 645
- 646
- 647

648
649

A EXECUTION EXAMPLES

650
651
652
653

To provide concrete insight into the testing process, we present two actual execution examples from our evaluation logs for the gpt-4o model with our optimal prompt. These examples illustrate both successful and challenging scenarios that models encounter during advanced computation pipeline tasks.

654

655

A.1 SUCCESSFUL EXECUTION EXAMPLE

656

657
658

In the following example, we provide an MDP-optimal workflow prompt to the LLM. The basic file processing task documentary is presented as follows.

659

660

Task Type: content_analysis

661

Prompt Type: optimal

662

Task Instance:

664

Required Tools: ['file_operations_reader',
'data_processing_parser', 'data_processing_filter']

665

Description: Load server log files (raw text/JSON) and recent customer feedback forms from the database, parse and normalize timestamps and service identifiers, and convert both sources into a common CSV schema. Apply business rules to filter by production services, map error codes to severity levels, and discard debug-level noise. Aggregate response times, error counts, and complaint keywords by hourly and daily intervals, compute rolling averages and spike detection, and correlate spikes with customer complaints. Deliver a trend analysis CSV and a one-page summary highlighting recurring anomalies, peak degradation windows, and suggested next steps for the IT ops team.

666

667

The prompt that we provide to the LLM consists of the description of the task, a detailed guide on how to perform tool search and tool call, and a recommended optimal sequence for executing the tools, given as

668

Workflow Execution Plan:

669

670

1. Execute file_operations_reader
2. Execute data_processing_parser
3. Execute data_processing_filter

671

672

The execution flow in this example begins with Turn 1,

673

Assistant: <tool_search>file operations reader</tool_search>

674

This tool search request from the agent triggers the search engine to provide a set of tool search results:

675

676

file_operations_reader, file_operations_writer,
file_operations_converter, file_operations_scanner,
file_operations_compressor

677

678

As an example, the full information of file_operations_reader returned by the tool search operation is presented as follows.

679

680

1. file_operations_reader

681

Category: file_operations

```

702     Description: This tool is specifically designed to read and
703     retrieve data from files.
704     Relevance: 0.70
705     Parameters:
706         - source (string): Source location or identifier [REQUIRED]
707         - options (object): Additional options [OPTIONAL] default={}
708     Returns:
709         - success (boolean): Whether operation succeeded
710         - data (object): Retrieved or parsed data
711         - metadata (object): Operation metadata
712     Possible Errors:
713         - INVALID_INPUT: Input validation failed
714         - OPERATION_FAILED: Operation could not be completed
715         - TIMEOUT: Operation timed out
716         - FILE_NOT_FOUND: Specified file not found
717         - PERMISSION_DENIED: Insufficient permissions
718

```

The assistant then applies the tool call operation

```
<tool_call>file_operations_reader</tool_call>
```

followed by a success message indicating that the LLM should move to the next execution step:

```
User: file_operations_reader executed successfully.
```

```
Output: {
725     "status": "completed",
726     "tool": "file_operations_reader",
727     "execution_context": {
728         "step": 0,
729         "total_tools_executed": 0
730     }
731 }
```

Current progress: 1 tools executed. Continue with the next step.

After several turns of tool search and tool call, the final successful execution history consists of the following records:

```
Step 1: {'tool': 'file_operations_reader', 'success': True,
738 'output': "{'status': 'completed', 'tool':
739 'file_operations_reader', 'execution_context': {'step': 0,
740 'total_tools_executed': 0}}", 'error': None}
```

```
Step 2: {'tool': 'data_processing_parser', 'success': True,
743 'output': "{'status': 'completed', 'tool':
744 'data_processing_parser', 'execution_context': {'step': 1,
745 'total_tools_executed': 1}}", 'error': None}
```

```
Step 3: {'tool': 'data_processing_filter', 'success': True,
748 'output': "{'status': 'completed', 'tool':
749 'data_processing_filter', 'execution_context': {'step': 2,
750 'total_tools_executed': 2}}", 'error': None}
```

A.2 CHALLENGING EXECUTION EXAMPLE

To illustrate common execution difficulties, we also present a challenging scenario where typical failure patterns occur.

The task instance is given as:

```

756 Task Instance:
757 -----
758 Required Tools: ['network_fetcher', 'data_processing_validator',
759 'data_processing_parser']
760
761 Description: Pull active customer records from the CRM API and
762 subscriber lists from the email marketing platform, and retrieve
763 shipping-rate quotes for each customer address via the shipping
764 rate API. Normalize and convert JSON/CSV fields, deduplicate by
765 email, and map CRM fields to email platform fields (first_name,
766 last_name, email, subscription_status, lifetime_value,
767 shipping_class). Apply business rules: prefer the record with the
768 latest CRM last_updated timestamp, set shipping_class = "fast" if
769 lowest quote < $10, and mark lifetime_value > $200 as VIP. Push
770 updates to both systems via their APIs and produce a CSV of
771 synchronized records plus a small sync report with counts and
772 errors.
773
774 The reference workflow:
775
776 1. Execute network_fetcher
777 2. Execute data_processing_validator
778     - Requires: data_processing_parser
779 3. Execute data_processing_parser
780
781 The execution results are provided as follows. For simplicity, we omit numerical details, keeping
782 only the tool search and tool call decisions made by the model, and the execution result records.
783
784 Turn 1:
785 Assistant: <tool_search>network fetcher</tool_search>
786 User: Tool Search Results: network_fetcher, network_poster,
787 network_monitor, network_router, utility_tracker (along with tool
788 information details)
789
790 Turn 2:
791 Assistant: <tool_call>network_fetcher</tool_call>
792 User: network_fetcher executed successfully.
793
794 Turn 3:
795 Assistant: <tool_search>data processing validator</tool_search>
796 User: Tool Search Results: data_processing_validator,
797 data_processing_validator, data_processing_filter,
798 data_processing_aggregator, data_processing_parser
799
800 Turn 4:
801 Assistant: <tool_call>data_processing_validator</tool_call>
802 User: data_processing_validator failed due to dependency issues.
803
804 - DEPENDENCY PROBLEMS:
805 Missing dependency: 'data_processing_parser' must be executed
806 first
807
808 - RECOMMENDATION:
809 Execute the required dependencies first, then retry this tool.
810
811 Turn 5:
812 Assistant: <tool_search>data processing parser</tool_search>
813 User: Tool Search Results: data_processing_parser,
814 data_processing_aggregator, data_processing_transformer,
815 data_processing_filter, data_processing_validator

```

```

810
811 Turn 6:
812 Assistant: <tool_call>data_processing_parser</tool_call>
813 User: data_processing_parser failed.
814 Error: TIMEOUT: Operation timed out (after 26 seconds)
815
816 Turn 7:
817 Assistant: <tool_search>data processing aggregator</tool_search>
818 User: Tool Search Results: data_processing_aggregator,
819 data_processing_filter,
820 data_processing_parser, data_processing_transformer,
821 data_processing_validator
822
823 Turn 8:
824 Assistant: <tool_call>data_processing_aggregator</tool_call>
825 User: data_processing_aggregator failed due to dependency issues.
826
827 - DEPENDENCY PROBLEMS:
828 Dependency 'data_processing_parser' failed - cannot proceed safely
829
830 - RECOMMENDATION:
831 Execute the required dependencies first, then retry this tool.
832
833 Turn 9:
834 Assistant: <tool_call>data_processing_parser</tool_call>
835 User: data_processing_parser failed.
836 Error: INVALID_INPUT: Input validation failed (expected: JSON
837 format)
838
839 ...
840
841 The agent's failure stemmed from two critical errors. First, in Turn 4, it overlooked the prerequisite
842 that data_processing_parser must precede data_processing_validator, leading to
843 an execution failure. Second, when data_processing_validator encountered a simulated
844 timeout in Turn 6, the agent attempted an improper fallback. It invoked data_processing_
845 aggregator, a tool for which it lacked the necessary information to correctly execute, causing
846 another execution error. These cascading failures ultimately caused the agent to exhaust its allotted
847 turns and fail the task.
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
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
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
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
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
1155
1156
1157
1158
1159
1160
1161
1162
1163
1164
1165
1166
1167
1168
1169
1170
1171
1172
1173
1174
1175
1176
1177
1178
1179
1180
1181
1182
1183
1184
1185
1186
1187
1188
1189
1190
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
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
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
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
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
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
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2178
2179
2180
2181
2182
2183
2184
2185
2186
2186
2187
2188
2189
2189
2190
2191
2192
2193
2194
2195
2196
2196
2197
2198
2199
2199
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2278
2279
2280
2281
2282
2283
2284
2285
2286
2286
2287
2288
2289
2289
2290
2291
2292
2293
2294
2295
2296
2296
2297
2298
2299
2299
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2378
2379
2380
2381
2382
2383
2384
2385
2386
2386
2387
2388
2389
2389
2390
2391
2392
2393
2394
2395
2396
2396
2397
2398
2399
2399
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2478
2479
2480
2481
2482
2483
2484
2485
2486
2486
2487
2488
2489
2489
2490
2491
2492
2493
2494
2495
2496
2496
2497
2498
2499
2499
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2578
2579
2580
2581
2582
2583
2584
2585
2586
2586
2587
2588
2589
2589
2590
2591
2592
2593
2594
2595
2596
2596
2597
2598
2599
2599
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2678
2679
2680
2681
2682
2683
2684
2685
2686
2686
2687
2688
2689
2689
2690
2691
2692
2693
2694
2695
2696
2696
2697
2698
2699
2699
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2778
2779
2780
2781
2782
2783
2784
2785
2786
2786
2787
2788
2789
2789
2790
2791
2792
2793
2794
2795
2796
2796
2797
2798
2799
2799
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2879
2880
2881
2882
2883
2884
2885
2886
2886
2887
2888
2889
2889
2890
2891
2892
2893
2894
2895
2896
2896
2897
2898
2899
2899
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2938
2939
2940
2941
2942
2943
2944
294
```

864 **Library Composition and Statistics.** The tasks within the library are categorized by type, com-
 865 plexity. Table 7 provides a detailed breakdown of these distributions.
 866

867 Table 7: Task library statistics.
 868

869 Category	870 Subcategory	871 Number of Tasks
871 Task Type	basic_file_processing	1200
	simple_data_transformation	320
	complex_validation_pipeline	1520
	complex_network_integration	1360
	advanced_computation_pipeline	640
876 Complexity	easy	1520
	medium	2880
	hard	640

879 The tasks are generated based on several templates defined through the generation process. Each
 880 **Task Type** corresponds to a specific generation logic that determines the complexity and nature of
 881 the task.
 882

883 **basic_file_processing:** These are fundamental tasks generated to represent simple data pro-
 884 cessing workflows. They typically involve a small number of tools (2-3) chosen randomly from the
 885 available tool library. The complexity is generally set to `easy`, focusing on straightforward input,
 886 processing, and output steps.

887 **simple_data_transformation:** This category includes tasks that use either one or two tools.
 888 Single-tool tasks are created for each tool category, often paired with optional validation or logging
 889 tools. Dual-tool tasks combine two randomly selected tools for a two-step process. These are also
 890 considered `easy` in complexity.

891 **complex_validation_pipeline:** These tasks simulate a multi-stage data validation pipeline.
 892 They are constructed by semantically selecting a chain of tools that follow a logical data flow: read-
 893 ing/parsing, transforming/validating, and writing/exporting. These tasks are assigned a `medium`
 894 complexity.

895 **complex_network_integration:** This task type is designed to simulate interactions with
 896 network APIs. The generator selects tools to create a sequence of fetching data from an endpoint,
 897 validating the response, and potentially posting data back. These are also of `medium` complexity.

898 **advanced_computation_pipeline:** These represent the most complex tasks. They are gen-
 899 erated by identifying longer, more intricate chains of tools based on their dependencies. The gen-
 900 erator attempts to find paths in the tool dependency graph, resulting in tasks that require a sequence of
 901 multiple tools to complete. These tasks are designated as `hard` complexity.
 902

903 **Task Parameters.** Each task in the library is defined by a set of parameters that describe its char-
 904 acteristics, requirements, and expected outcomes. Table 8 provides an overview of these parameters.
 905

906 Table 8: Description of task parameters.
 907

908 Parameter	909 Type	910 Description
instance_id	String	A unique identifier for the task instance.
task_type	String	The category of the task (e.g., <code>basic_file_processing</code> , <code>complex_validation_pipeline</code>).
description	String	A brief, human-readable description of the task.
inputs	Object	An object containing the input data and options for the task. This often includes a <code>source</code> file path and various processing options.
expected_outputs	Object	An object describing the expected outcome of the task, which typically includes a <code>success</code> boolean and may contain <code>metadata</code> about the results.

918
919

Table 8 continued from previous page.

920
921
922
923
924
925
926
927
928
929
930
931
932
933
934

Parameter	Type	Description
required_tools	Array	A list of strings, where each string is the name of a tool required to complete the task.
constraints	Object	An object specifying any constraints on the task execution, such as <code>timeout</code> in seconds and the maximum number of <code>max_retries</code> .
complexity	String	The complexity level of the task, categorized as <code>easy</code> , <code>medium</code> , or <code>hard</code> .
metadata	Object	An object containing metadata about the task generation, such as the <code>template</code> used, generation timestamp, and whether it was <code>LLM-generated</code> .
original_description	String	The original, more detailed description of the task before any enhancements or modifications.

935
936
937
938

B.1.2 EXAMPLE OF A TASK

939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971

To directly provide an idea of what a task might look like, we present an example of a “basic file processing task” in the task library.

The following task requires the LLM to execute a sequence of `file_operations_reader`, `data_processing_parser` and `data_processing_filter` tools to process the input data.

```
{
  "instance_id": "task_dee2d02d",
  "task_type": "basic_file_processing",
  "description": "This task retrieves data from a CSV file, parses it into a structured format, and filters it based on specified criteria. The final result will be a refined dataset that meets the filtering conditions.",
  "inputs": {
    "source": "data/input_file.csv",
    "options": {
      "filter": "true"
    }
  },
  "expected_outputs": {
    "success": true,
    "metadata": {
      "total_records": 100,
      "filtered_records": 80
    }
  },
  "required_tools": [
    "file_operations_reader",
    "data_processing_parser",
    "data_processing_filter"
  ],
  "constraints": {
    "timeout": 300,
    "max_retries": 3
  },
  "complexity": "easy",
  "metadata": {
    "template": "basic_file_processing",
    "generated_at": "2025-07-10T04:27:35.913857",
    "timeout": 300
  }
}
```

```

972         "semantic_generation": true,
973         "llm_generated": true,
974         "inputs_generated_from": "llm"
975     }
976 }
977

```

978 B.1.3 PARAMETERS AND STATISTICS OF TOOLS

979 The tool library consists of a set of 30 canonical tools, each designed to perform a specific operation.
980 These tools are categorized based on their functionality, such as data processing, file operations, and
981 network interactions. This section provides a detailed breakdown of the library’s composition and
982 the parameters used across the different tools.
983

984 **Tool Library Composition.** The tools are grouped into six distinct categories. The distribution
985 of tools across these categories is uniform, with each category containing five tools. This balanced
986 distribution ensures a wide range of capabilities within the library. Table 9 summarizes this distri-
987 bution.

988
989 Table 9: Distribution of tools by category.

990	991	Category	Number of Tools
992		computation	5
993		data_processing	5
994		file_operations	5
995		integration	5
996		network	5
997		utility	5

998
999 **Common Tool Parameters.** The tools in the library share a common set of parameters that define
1000 their inputs and behavior. These parameters allow for consistent interaction with the tools, regardless
1001 of their specific function. Table 10 provides a description of the most frequently used parameters.
1002

1003 Table 10: Common tool parameters and their usage.
1004

1005	Parameter	Data Type	Occurrences	Description
1006	options	object	25	A flexible object for passing additional options.
1007	source	string	5	The source location or identifier for data input.
1008	data	object	5	The data payload to be processed or sent by the tool.
1009	timeout	number	5	The timeout duration for the operation, in seconds.
1010	retry_count	number	5	The number of times to retry the operation upon failure.
1011	destination	string	4	The target location or identifier for the output.
1012	input_format	string	4	The format of the input data (e.g., JSON, CSV).
1013	output_format	string	4	The desired format for the output data.
1014	schema	object	4	The validation schema to check the data against.
1015	precision	number	4	The numerical precision for computational tasks.

1016
1017 **Tool Construction Parameter Details.** Each tool is defined by a structured set of parameters, re-
1018 turn values, and potential errors. The following table, Table 11, provides a comprehensive overview
1019 of the fields that constitute a tool’s definition in the library.
1020

1021 Table 11: Detailed description of tool definition fields.
1022

1023	Field Name	Type	Description
1024	name	String	The unique name of the tool.

1026
1027

Table 11 continued from previous page

1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
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

Field Name	Type	Description
description	String	A human-readable summary of the tool’s purpose and functionality.
parameters	Array	An array of objects, where each object defines an input parameter for the tool. This includes the parameter’s name, type, description, and whether it is required.
returns	Array	An array of objects describing the possible return values from the tool. Each object specifies the name, type, and a description of the return value.
errors	Array	A list of possible errors that the tool might throw, including an error code and description for each.
dependencies	Array	A list of other tools that must be executed before this tool can run.
dependency_metadata	Object	Metadata related to the tool’s dependencies, including its execution level and order in a workflow.
metadata	Object	General metadata about the tool, such as its category, operation type, and creation timestamp.
canonical_name	String	The official, unique name for the tool, used to resolve any aliases.
aliases	Array	A list of alternative names that refer to this tool.
differentiation	Object	An object containing details that distinguish this tool from others, including its unique purpose, key differentiators, and usage keywords.
original_description	String	The initial, unenhanced description of the tool.
differentiation_enhanced	Boolean	A flag indicating whether the tool’s differentiation information has been enhanced by an LLM.

B.1.4 EXAMPLE OF A TOOL

We also provide an example to demonstrate what a tool might look like. Recall that a tool named `file_operations_reader` is required in executing the task example. Here we demonstrate its registry in the tool library.

```

"file_operations_reader": {
    "name": "file_operations_reader",
    "description": "This tool is specifically designed to read and
    retrieve data from files.",
    "parameters": [
        {
            "name": "source",
            "type": "string",
            "description": "Source location or identifier",
            "required": true,
            "default": null,
            "constraints": {}
        },
        {
            "name": "options",
            "type": "object",
            "description": "Additional options",
            "required": false,
            "default": {},
            "constraints": {}
        }
    ],
    "returns": [

```

```

1080      {
1081          "name": "success",
1082          "type": "boolean",
1083          "description": "Whether operation succeeded"
1084      },
1085      {
1086          "name": "data",
1087          "type": "object",
1088          "description": "Retrieved or parsed data"
1089      },
1090      {
1091          "name": "metadata",
1092          "type": "object",
1093          "description": "Operation metadata"
1094      },
1095      "errors": [
1096          {
1097              "code": "INVALID_INPUT",
1098              "description": "Input validation failed"
1099          },
1100          {
1101              "code": "OPERATION_FAILED",
1102              "description": "Operation could not be completed"
1103          },
1104          {
1105              "code": "TIMEOUT",
1106              "description": "Operation timed out"
1107          },
1108          {
1109              "code": "FILE_NOT_FOUND",
1110              "description": "Specified file not found"
1111          },
1112          {
1113              "code": "PERMISSION_DENIED",
1114              "description": "Insufficient permissions"
1115          },
1116          "dependencies": [],
1117          "dependency_metadata": {
1118              "level": 0,
1119              "execution_order": 0,
1120              "category": "file_operations"
1121          },
1122          "metadata": {
1123              "category": "file_operations",
1124              "operation": "reader",
1125              "version": "1.0.0",
1126              "created_at": "2025-06-27T17:38:31.057340",
1127              "dependency_level": 0,
1128              "execution_order": 0
1129          },
1130          "canonical_name": "file_operations_reader",
1131      },
1132
1133

```

In summary, the registration of a tool in the tool library, also often referred to as MCP (Model Context Protocol), is presented as a standardized interface system that provides structured access to external tools and services. Rather than a custom protocol implementation, it uses generated JSON schemas and client libraries that define how AI agents can interact with various tools. Specifically,

1134 the schemas specify (1) the *input parameters* (given as `parameters` in the following) that an agent
 1135 needs to provide to the tool, and (2) the possible outputs of executing the tool, including *return*
 1136 *values* (given as `returns`) and *error messages* (given as `errors`). The schema also specifies
 1137 other information, including the dependencies of the tool. If a tool has other dependencies, all
 1138 dependencies must be successfully executed before the tool can be called.

1139 Provided with the task to complete, and the tool library for usage, the LLM is required to perform
 1140 tool calls, and check their return values to ensure task completion.

1142 B.2 PROMPTS

1144 We provide the LLMs with multiple types of task-related *prompts*, carrying necessary information
 1145 regarding the task. The LLMs are prompted to identify and execute tools to complete the task. The
 1146 prompts of PILOT-BENCH are classified into 4 categories:

- 1148 1. Baseline prompts: Only contains basic information, such as task description, input and
 1149 output, tool call instructions, etc.
- 1150 2. CoT (Chain of Thought) prompts: Baseline prompts enhanced with CoT reasoning instruc-
 1151 tions.
- 1152 3. MDP-optimal workflow prompts: Baseline prompts enhanced with a detailed workflow
 1153 execution plan.
- 1154 4. Flawed workflow prompts: Optimal prompts with various flaws injected to the workflow
 1155 execution plan description.

1157 B.2.1 AN EXAMPLE OF BASELINE PROMPT

1158 Execute a simple task.

1161 Task: Leverage innovative multi-tool workflows to metamorphose
 1162 ambiguous input into strategic insights, enhancing operational
 1163 efficacy and fostering data-driven decisions, thus unlocking
 1164 untapped business potential through streamlined transformation.

1165 Input Data:

1166 - `input_data`: numeric array with 5 values

1168 Expected Output:

1169 - `processed_data`: processed data with success status

1171 Tool Search Available:

1172 You have access to a comprehensive tool library with specialized
 1173 tools for various operations.

1174 To find relevant tools, use the search command: `<tool_search>your`
 1175 `search query</tool_search>`

1176 Examples of tool searches:

1177 - `<tool_search>file reader writer</tool_search>`
 1178 - `<tool_search>data validation parser</tool_search>`
 1179 - `<tool_search>network api fetch</tool_search>`

1181 After finding the tools you need, execute them using:

1182 `<tool_call>tool_name</tool_call>`

1184 Instructions:

- 1185 1. Analyze the task requirements
- 1186 2. Search for appropriate tools based on what you need to do
- 1187 3. Execute the tools in the correct order
- 1188 4. Complete the task and indicate when finished

```

1188
1189 Use appropriate tools to complete the task.
1190
1191 B.2.2 COT REASONING INSTRUCTIONS
1192
1193 **Think step by step about which tools to use and why.**
1194
1195 Please:
1196 1. First explain your reasoning about which tools to use
1197 2. Then execute the tools in the order you determined
1198 3. Format tool calls as: <tool_call>tool_name</tool_call>
1199
1200 Begin with "Reasoning:" followed by your thought process.
1201
1202 Use appropriate tools to complete the task.
1203
1204 The instructions are directly appended to the baseline prompt to form a CoT prompt.
1205
1206 B.2.3 AN EXAMPLE OF MDP-OPTIMAL WORKFLOW PROMPT
1207
1208 We provide and example of a workflow execution plan with summary:
1209
1210 ## Workflow Execution Plan
1211
1212 1. Execute data_processing_transformer
1213   - Reason: Semantic match for 'full_description' operation
1214   - Requires: data_processing_parser
1215 2. Execute data_processing_filter
1216   - Reason: Semantic match for 'full_description' operation
1217 3. Execute data_processing_parser
1218   - Reason: Semantic match for 'full_description' operation
1219 4. Execute computation_analyzer
1220   - Reason: Semantic match for 'full_description' operation
1221   - Requires: data_processing_parser, data_processing_aggregator
1222 5. Execute file_operations_scanner
1223   - Reason: Semantic match for 'full_description' operation
1224 6. Execute file_operations_reader
1225   - Reason: Semantic match for 'full_description' operation
1226 7. Execute data_processing_validator
1227   - Reason: Semantic match for 'full_description' operation
1228   - Requires: data_processing_parser
1229 8. Execute computation_calculator
1230   - Reason: Semantic match for 'full_description' operation
1231   - Requires: data_processing_parser, network_validator
1232 9. Execute network_monitor
1233   - Reason: Semantic match for 'full_description' operation
1234 10. Execute data_processing_aggregator
1235   - Reason: Semantic match for 'full_description' operation
1236   - Requires: data_processing_parser
1237
1238 ### Analysis:
1239   - Critical tools identified: data_processing_filter,
1240     data_processing_transformer, data_processing_parser
1241
1242 ### Execution Strategy:
1243 1. Follow the recommended sequence for optimal results
1244 2. Use alternatives if primary tools fail

```

1242 3. Pay special attention to critical tools
 1243

1244 Use appropriate tools to complete the task.
 1245

1246 The workflow execution plan is attached directly to the baseline prompt to form an MDP-optimal
 1247 workflow prompt. In section D, we will explain how the MDP-optimal workflow prompt is con-
 1248 structed.
 1249

1250 B.2.4 FLAWED WORKFLOW PROMPT
 1251

1252 There are several ways to perturb an MDP-optimal workflow execution plan to make it flawed. For
 1253 example, we can introduce flaws to the order of the tool execution sequence. We can also introduce
 1254 missing steps or redundancies. An ideally robust LLM is expected to discover and correct the flaws.

1255 To test the capability of the LLM, we include Sequential Ordering Errors, where tools are executed
 1256 in an incorrect sequence, and Tool Misuse Errors, which involve selecting an inappropriate tool
 1257 that appears suitable. Parameter Configuration Errors occur from incorrect or omitted parameters,
 1258 while Missing Critical Steps involves the strategic removal of essential operations. We also observe
 1259 Redundant Operations, where unnecessary tools are added; Logic Discontinuity, where a tool’s
 1260 output is incompatible with the next tool’s input; and Semantic Drift, a gradual deviation from the
 1261 intended workflow caused by replacing tools with functionally different but semantically similar
 1262 alternatives. We refer to C.3 for details of the flaws.
 1263

1264 B.3 TOOL RESULT SIMULATOR
 1265

1266 Once the LLM performs a tool call, a simulator provides a return message that is either a success or
 1267 a type of failure. For simplicity, we create a unified simulator for every tool, which calculates the
 1268 success rate of a specific tool execution, then randomly samples a success or a failure.

1269 The specific simulator implementation is decomposed as follows:
 1270

1271 **Success rate calculator.** The success rate calculator assigns a success rate $p \in [0, 1]$ for each
 1272 specific tool execution. Specifically, the calculator first assigns a base success rate of $p_0 = 0.8$.
 1273 Then, dependencies of the current tool is checked. For each dependency that has not been previ-
 1274 ously called, a penalty of 0.5 is multiplied to the success rate. For each dependency that has been
 1275 previously called but not successfully executed, a penalty of 0.7 is multiplied to the success rate.
 1276 Further, failure histories of the current task decreases the success rate of the current tool execution.
 1277 Each history failure record induces a penalty of 0.9 multiplied to the success rate. In summary, we
 1278 have
 1279

$$1280 \quad \rho_{\text{success}} = \rho_{\text{base}} \cdot \prod 0.5^{N_u} \cdot 0.7^{N_f} \cdot 0.9^{N_h}$$

1281 where N_u is the number of dependencies not called, N_f is the number of dependencies (called but
 1282 not successfully executed, and N_h is the number of historical execution failures.
 1283

1284 **Result simulator.** Given the success rate p , the result simulator first randomly simulates a “suc-
 1285 cess” or “failure”. If the result is “success”, the simulator creates the success-related return message
 1286 provided in the tool’s documentation. If the result is “failure”, the simulator randomly selects a type
 1287 of failure (e.g., TIMEOUT, EXECUTION_FAILURE) from the tool’s documentation.
 1288

1289 B.4 TASK RESULT EVALUATION
 1290

1291 The evaluation of task outcomes is performed by the `InteractiveExecutor`, which determines
 1292 the success of a task based on a detailed analysis of the execution history. This section outlines the
 1293 different levels of task success and the criteria used for their assessment.
 1294

1296 B.4.1 LEVELS OF TASK SUCCESS
12971298 **full_success:** This level indicates that the task was completed perfectly. It requires all of the
1299 specified tools to be executed successfully and in the correct sequence.1300 **partial_success:** This level represents a task that was completed to a significant extent but
1301 did not meet all the criteria for full success. This could mean, for example, that a majority of the
1302 required tools were executed, or that a valid output was generated despite some tools failing.1303 **failure:** This level is assigned when the task could not be meaningfully completed. This typ-
1304 ically occurs if a critical number of tools fail, no output is generated for essential tasks, or the
1305 execution gets stuck in a loop.1306
1307 B.4.2 EVALUATION CRITERIA
13081309 The executor evaluates the success of a task by assessing several key factors from the execution
1310 state. The final success level is determined by a combination of these criteria:

- **Required Tools Coverage:** This is the most critical metric. The evaluator checks what percentage of the tools listed in the task’s `required_tools` list were executed successfully. Full success requires 100% coverage.
- **Sequence Correctness:** For a task to be considered a `full_success`, the required tools must not only be executed but also be called in the precise order specified in the task definition.
- **Output Generation:** The evaluator checks whether any tool that is expected to produce an output (e.g., tools with names like ‘writer’, ‘exporter’, ‘saver’) was successfully executed. The generation of an output is an indicator of full success and partial success, especially for pipeline-oriented tasks.
- **Explicit Completion Signal:** The conversation history is scanned for signals from the language model indicating that it considers the task complete (e.g., “task completed”, “finished executing”).
- **Minimum Tool Execution:** For each task type, there is a minimum number of successfully executed tools required to be considered for `partial_success`. For instance, an `advanced_computation_pipeline` requires more successful tool calls than a `simple_data_transformation`.
- **Termination Conditions:** The evaluation also considers reasons for premature termination, such as an excessive number of consecutive tool failures or getting stuck in a repetitive loop, which would typically lead to a `failure` rating.

1332 A task is rated as a `partial_success` if it meets at least two of the conditions (e.g., has over
1333 Minimum Tool Execution tool coverage and generates an output). If the conditions for either
1334 full or partial success are not met, the task is marked as a `failure`.1336 C BENCHMARK CONSTRUCTION
13371339 In this section, we introduce how the tasks are automatically constructed. Specifically, this includes
1340 the generation of the *task documentation*, *tool library* and *prompts*.1341 C.1 TASK AND TOOL CONSTRUCTION
13421343 The generation framework employs a *category-mediated* architecture where tool categories serve as
1344 the primary interface between tool capabilities and task requirements, ensuring that generated tasks
1345 are both reasonable and logically correspond to the underlying tool ecosystem.1347 C.1.1 TOOL GENERATION METHODOLOGY
13481349 We first generate the tool library, which is independent of tasks, and generated via a LLM-free rule-
based process. This process highly relies on categories, where the basic layer of category consists

1350 of a category-operation matrix consisting of 30 distinct tool types. Then, we introduce semantic
 1351 operations and semantics matching that re-categorizes these tools, in order to (1) introduce aliases
 1352 to flexibly create more tools and (2) facilitate and interface with task creation.
 1353

1354 **Layer 1: Category-Operation Matrix (Tool Definition Layer).** The system defines 6 categories,
 1355 each with 6 specific operations, creating a 6×5 matrix of 30 distinct tool types. The categories and
 1356 their operations are given as
 1357

- data_processing: parser, transformer, validator, aggregator, filter
- file_operations: reader, writer, scanner, compressor, converter
- network: fetcher, poster, monitor, validator, router
- computation: calculator, analyzer, optimizer, simulator, predictor
- integration: connector, authenticator, mapper, queue, scheduler
- utility: logger, cache, notifier, tracker, helper

1366 Within the matrix, each category-operation pair (e.g., data_processing_parser, network_fetcher) generates
 1367 a unique tool with:
 1368

- Specific parameter templates
- Specific return value templates
- Specific error handling templates
- Category-specific behavior patterns

1374 The category-operation matrix-based templates generate the following information of tools:
 1375

- **Naming Convention.** Tools follow a systematic {category}-{operation} pattern, ensuring consistent identification and categorization.
- **Parameter Assignment.** The system uses rule-based logic to assign appropriate parameters based on operation semantics. For example, reading operations (e.g., reader, parser, scanner) receive source parameters, and transformation (e.g., transformer, converter, mapper) operations receive input_format and output_format specification parameters.
- **Return Value Generation.** Return values are systematically assigned based on operation type. For example, operations for receiving data, e.g., reader, fetcher, scanner, returns data values. Other than the operation-specific return values, all tools provide success indicators and metadata.

1387 Another part of return values is the error information, where we also create general errors such as INVALID_INPUT, and operation-specific errors such as OVERFLOW for computation operations.
 1388
 1389

1390 **Layer 2: Semantic Operation Types (Workflow/Dependency Layer).** The system then regroups
 1391 the 30 operations into 5 semantic operation types based on their role in data processing workflows:
 1392

```
1393 operation_types = {
1394     'sources': ['reader', 'fetcher', 'scanner', 'authenticator'],
1395     'processors': ['parser', 'transformer', 'validator',
1396                   'analyzer', 'calculator'],
1397     'aggregators': ['aggregator', 'combiner', 'merger'],  # Note:
1398     # some are aliased
1399     'outputs': ['writer', 'poster', 'exporter', 'notifier'],  #
1400     # Note: some are aliased
1401     'utilities': ['logger', 'cache', 'tracker', 'monitor']
1402 }
1403
```

1403 We have the following remarks:

1404
 1405 1. **Cross-category semantic grouping.** Operations from different categories can belong to
 1406 the same semantic type. For example:
 1407

- sources includes: reader (file_operations), fetcher (network), scanner (file_operations), authenticator (integration)
- processors includes: parser (data_processing), transformer (data_processing), validator (data_processing/network), analyzer (computation), calculator (computation)

1408
 1409 2. **Some operations have aliases.** The semantic layer introduces some operations not explicitly in the original 30:

- combiner and merger (aliases for aggregator)
- exporter (alias for writer or converter)

1410
 1411 3. **Workflow-based logic.** This semantic grouping enables dependency rules like:

```
1412 dependency_rules = {
  1413   'processors': {
    1414     'transformer': ['parser', 'reader'], # transformers
    1415     depend on parsers and readers
    1416     'validator': ['parser', 'transformer'],
    1417     'analyzer': ['parser', 'aggregator'],
    1418     'calculator': ['parser', 'validator']
    1419   },
    1420   # ... more rules
  1421 }
```

1422
 1423 Based on the dependency rules, the prerequisites of each tool is configured, facilitating
 1424 construction of complicated workflows with dependencies.

1425
 1426 **How the Two Layers Work Together.** We now summarize how the two layers of categorization
 1427 work together to generate tool information, and facilitate task construction. First, in the *tool generation phase*, we use Layer 1 (category + operation) to create specific tools. Each tool gets a unique
 1428 name and a category-specific parameter/return template. Next, we move to the *dependency resolution phase*, where we use Layer 2 (semantic operation types) to determine logical dependencies.

1429
 1430 Lastly, the *task construction phase*, which will be introduced shortly, combines both layers to build
 1431 workflow-aware task templates. This phase uses semantic types to understand data flow patterns,
 1432 and uses specific category-operation tools as the actual implementation.

1433 C.2 TASK GENERATION METHODOLOGY

1434
 1435 This section describes a systematic approach for generating computational tasks based on a library
 1436 of available tools, ensuring that generated tasks are both reasonable and logically correspond to the
 1437 capabilities of the underlying tool ecosystem. The framework employs a multi-layered architecture
 1438 that combines predefined task templates, semantic tool matching, and large language model (LLM)
 1439 augmentation to create diverse, executable task instances.

1440
 1441 **Foundational Task Categories** The system first defines five types of tasks, including basic file
 1442 processing, simple data transformation, complex validation pipeline, complex network integration,
 1443 and advanced computation pipeline.

1444
 1445 Then, for each category of tasks, we define a standard operation sequence, given as

```
1446 operation_sequences = {
  1447   'basic_file_processing': ['input', 'process'],
  1448   'simple_data_transformation': ['read', 'process', 'output'],
  1449   'complex_validation_pipeline': ['read', 'validate',
  1450     'transform', 'aggregate', 'write'],
  1451   'complex_network_integration': ['fetch', 'parse', 'validate',
  1452     'transform', 'post'],
  1453   'advanced_computation_pipeline': ['read', 'validate',
  1454     'transform', 'compute', 'aggregate', 'write'],
  1455 }
```

1458 The task generation process begins with a structured template system that defines the fundamental
 1459 characteristics of different task types. Each task template serves as a blueprint that specifies the
 1460 following core components:
 1461

- 1462 • **Task requirements:** A TaskRequirement object that defines the minimum number of
 1463 tools needed, required tool operations (such as "reader", "transformer", "writer"), and com-
 1464 plexity constraints. This ensures that generated tasks align with available tool capabilities
 1465 and maintain logical coherence.
- 1466 • **Template structure:** Each template includes a task type identifier, descriptive require-
 1467 ments, and objectives that guide the generation process. The template acts as a constraint
 1468 mechanism, ensuring that only feasible task combinations are considered.
- 1469 • **Tool category analysis:** The system performs automated analysis of the available tool
 1470 library to identify tool categories and their associated operations. This analysis informs
 1471 template creation, ensuring that templates are grounded in actual tool availability rather
 1472 than abstract specifications.

1473 **Semantic Tool Matching.** To connect tasks to the pre-generated tool system, the framework in-
 1474 corporates semantic matching to select appropriate tools for task generation. For example, when
 1475 read exists in the operation sequence of a task, RAG semantic matching would lead to operations
 1476 such as reader, fetcher and scanner in the tool *semantic operation types*. This further leads
 1477 to retrieving specific tools, such as file_operations_reader. To create variations in one task
 1478 category, the specific tool matching with a task operation has multiple choices (e.g., read can corre-
 1479 spond to file_operations_reader or network_fetcher). For each task, only one specific
 1480 tool is selected for each operation step. Other sources of variation come from the choice of different
 1481 templates.

1482 **Task Construction Example.** We now provide a simple example of how a task might be con-
 1483 structed.

1484 Step 1: Choice of tools

```
1485 complex_validation_pipeline = ['read', 'validate', 'transform',  

  1486 'aggregate', 'write']  

  1487  

  1488 # semantics search result:  

  1489 'read' -> choose 'file_operations_reader'  

  1490 'validate' -> choose 'data_processing_validator'  

  1491 'transform' -> choose 'data_processing_transformer'  

  1492 'aggregate' -> choose 'data_processing_aggregator'  

  1493 'write' -> choose 'file_operations_writer'  

  1494  

  1495 # final required_tools = [a list of tools after dependency-based  

  1496 sorting]
```

1497 Step 2: Task construction

```
1498  

  1499 # 1. Task ID  

  1500 instance_id = f"task_{uuid.uuid4().hex[:8]}"  

  1501  

  1502 # 2. Task description (template-based)  

  1503 description = f"... #{i+1}"  

  1504  

  1505  

  1506 # 3. Input and output (based on template and tools)  

  1507 requirements = [  

  1508     TaskRequirement("processing_config", "Configuration for  

  1509     processing"),  

  1510     ...  

  1511 ]
```

```

1512     ]
1513
1514     objectives = [
1515         TaskObjective(
1516             "...",
1517             [f"Execute {tool}" for tool in selected_tools],
1518             "output"
1519         )
1520     ]
1521
1522     # 4. Difficulty (based on task type and tools)
1523     complexity = "easy" if len(tools) <= 2 else "medium"
1524
1525 LLM Enhancement. After the basic RAG tool-selection results of a task, we can also use LLM
1526 APIs to create an instance of a task.
1527
1528 Specifically, we provide to the LLM related RAG semantic search result together with the following
1529 prompt:
1530
1531     prompt = f"""
1532 You are an expert workflow designer. Based on RAG search results,
1533 design {task_desc}.
1534
1535 Available tools (sorted by relevance):
1536 {json.dumps(tools_info, indent=2)}
1537
1538 Requirements:
1539 - Task type: {task_type}
1540 - Complexity: {complexity}
1541 - Select 3-6 most appropriate tools based on:
1542     1. RAG relevance scores
1543     2. Logical workflow sequence
1544     3. Tool categories and operations
1545     4. Input/output compatibility between tools
1546
1547 Design a complete task instance with:
1548     1. A clear description of what the task accomplishes
1549     2. A logical sequence of tools that work together
1550     3. Realistic input data that matches the first tool's parameters
1551     4. Expected output that matches the final tool's returns
1552 """
1553
1554 The LLM enhancement provides a more natural and detailed task description, and ensures consistency (e.g., among choices of tools and description) within the task setting.
1555
1556 C.3 WORKFLOW PROMPT GENERATION
1557
1558 Each task has several corresponding prompts that guide the tested LLM to complete the task. The
1559 baseline prompts and CoT prompts can be directly transformed from the task profile in the generated
1560 task library. The MDP-optimal workflow prompt is generated through MDP training (details
1561 provided in Section D), while the flawed workflow prompt is created by inserting deliberate flaws.
1562
1563 We present a comprehensive framework for systematically generating flawed workflows from optimal
1564 sequences. Our approach introduces seven distinct categories of workflow defects with varying
1565 severity levels, enhanced by semantic similarity detection through RAG techniques.
1566
1567     • Sequential Ordering Errors (Order Flaws). Sequential ordering errors occur when tools
1568     are executed in incorrect or flawed sequences, violating logical dependencies or temporal
1569     constraints.

```

- 1566 To inject this error into the workflow, we use the *swap method*, i.e., creating random permutations of adjacent tool pairs. We also use the *dependency violation method*, which repositions dependent tools before their prerequisites, to inject more logical flaws.
- 1567
- 1568
- 1569
- 1570 • **Tool Misuse Errors** Tool misuse represents the selection of inappropriate tools that appear suitable but lack the required functionality for the specific context.
- 1571 To inject this error, we use the *semantic similarity method*, leveraging RAG-based semantic search to identify tools with similar descriptions but different functionalities. We also use *category mismatch method*, which replaces tools with alternatives from entirely different functional categories, simulating gross misunderstanding of tool capabilities.
- 1572
- 1573
- 1574
- 1575
- 1576
- 1577
- 1578 • **Parameter Configuration Errors.** Parameter errors involve incorrect specification or omission of required tool parameters.
- 1579 To inject this error, we use the *missing parameters method*, which systematically removes required parameters based on tool specifications. We also use the *type mismatch method*, which introduces parameters with incorrect data types or value ranges. Parameter error injection follows the formula $P_{error} = \alpha \times P_{required} + \beta \times P_{optional}$, where α and β are severity-dependent coefficients.
- 1580
- 1581
- 1582
- 1583
- 1584
- 1585 • **Missing Critical Steps.** Workflow incompleteness through strategic removal of essential tools or validation steps.
- 1586 To inject this error, we use the *middle step removal method*, which eliminates intermediate processing steps while preserving workflow endpoints. We also use the *validation removal method*, which systematically removes quality assurance and verification tools.
- 1587
- 1588
- 1589
- 1590 • **Redundant Operations.** Introduction of unnecessary or duplicated operations that increase computational overhead without adding value.
- 1591 To inject this error, we use the *duplication method*, which repeats existing tools within the sequence. We also use the *unnecessary addition method*, which inserts tools that provide no functional benefit to the task.
- 1592
- 1593
- 1594 • **Logic Discontinuity.** Breaks in logical flow where tool outputs become incompatible with subsequent tool inputs.
- 1595 To inject this error, we use the *format mismatch method*, which introduces incompatible data format transitions between tools. We also use the *unrelated insertion method*, which adds tools unrelated to the primary task objective.
- 1596
- 1597
- 1598 • **Semantic Drift (RAG-Enhanced).** Advanced error patterns enabled by semantic understanding, representing gradual deviation from intended workflow semantics.
- 1599 To inject this error, we use the *semantic mismatch method*, which replaces tools with semantically similar but functionally inappropriate alternatives. We also use the *semantic drift method*, which performs progressive replacement where each subsequent tool is selected based on the previous replacement, creating cumulative semantic deviation. The drift function is defined as $T_i = \arg \max_{t \in T} (S(t, T_{i-1}) \times (1 - F(t, T_{original})))$, where S represents semantic similarity and F represents functional equivalence.
- 1600
- 1601
- 1602
- 1603
- 1604
- 1605
- 1606

1607 D MDP-BASED WORKFLOW GENERATION TRAINING MECHANISM

1608 This section provides a comprehensive analysis of the Markov Decision Process (MDP) training framework used for MDP-optimal workflow generation in our system.

1612 D.1 MDP STATE SPACE FORMULATION

1614 The MDP state space \mathcal{S} is defined by a composite representation that captures both task semantics
1615 and execution dynamics. Each state $s_t \in \mathcal{S}$ is represented as:

$$1616 \quad s_t = \langle \tau, \omega; \psi, \phi, \xi \rangle$$

1617 where τ and ω are task-invariant components that remain constant throughout execution, while ψ ,
1618 ϕ , and ξ capture the dynamic execution state.

1620 **Static Task Components** (τ, ω): The task identification $\tau = \{\text{task_id, task_type, task_objective}\}$
 1621 and semantic feature vector $\omega \in \mathbb{R}^{20}$ encode task requirements (input/output needs, domain type,
 1622 complexity level) as normalized features. These components are extracted once at initialization and
 1623 provide consistent task context for all subsequent decisions.

1624

1625 **Tool Execution States Component** (ψ)

1626

$$\psi = \{\sigma_i : i \in \mathcal{T}\} \text{ where } \sigma_i \in \Sigma$$

1627

1628 The tool execution status set Σ contains:

1629

$$\Sigma = \{\text{NOT_ATTEMPTED, QUEUED, RUNNING, SUCCESS,} \\ \text{FAILED, TIMEOUT, DEPENDENCY_FAILED}\}$$

1630

1631 where \mathcal{T} represents the available tool set and each σ_i tracks the execution status of tool i .

1632

1633 **Progress and Workflow Component** (ϕ)

1634

$$\phi = \{p, k, \mathcal{M}_{\text{achieved}}, \mathcal{M}_{\text{expected}}, \mathbf{e}_{\text{seq}}\}$$

1635

1636 where:

1637

- 1640 • $p \in [0, 1]$ denotes overall task progress
- 1641 • $k \in \mathbb{N}$ represents the current workflow step
- 1642 • $\mathcal{M}_{\text{achieved}} \subseteq \mathcal{M}_{\text{expected}}$ are milestone sets
- 1643 • $\mathbf{e}_{\text{seq}} = [\iota_1, \iota_2, \dots, \iota_k]$ is the tool execution sequence

1644

1645 **RAG-Enhanced Context Component** (ξ) This component integrates retrieval-augmented gener-
 1646 ation capabilities:

1647

$$\xi = \{\mathcal{R}_{\text{rag}}, \mathbf{E}_{\text{cache}}, \mathbf{C}_{\text{semantic}}, \mathbf{H}_{\text{selection}}, \mathcal{T}_{\text{candidates}}\}$$

1648

1649 where:

1650

- 1651 • \mathcal{R}_{rag} : RAG search results mapping semantic operations to tool-score pairs
- 1652 • $\mathbf{E}_{\text{cache}}$: Pre-computed embedding similarity scores for all tools
- 1653 • $\mathbf{C}_{\text{semantic}}$: Semantic confidence scores per tool
- 1654 • $\mathbf{H}_{\text{selection}}$: Tool selection history with metadata
- 1655 • $\mathcal{T}_{\text{candidates}}$: Capability-type to candidate-tools mapping for fallback selection

1656

1657 **State Encoding for Neural Networks** The composite state is encoded into a fixed-size vector
 1658 $\mathbf{s}_t \in \mathbb{R}^d$ through concatenation and normalization:

1659

$$\mathbf{s}_t = \text{Concat}(\text{encode}(\psi), \text{encode}(\phi), \text{encode}(\xi), \omega)$$

1660

1661 where $\text{encode}(\cdot)$ transforms discrete components into continuous representations using one-hot
 1662 encoding, progress normalization, and embedding lookups.

1663

1664 This multi-faceted state representation enables the MDP to capture both structural workflow depen-
 1665 dencies and semantic task requirements, facilitating more informed action selection in complex tool
 1666 orchestration scenarios.

1667

1668 D.2 MDP ACTION SPACE DEFINITION

1669

1670 The action space \mathcal{A} consists of structured actions that operate on tool orchestration and workflow
 1671 management. Each action $a \in \mathcal{A}$ is represented as a composite tuple:

1672

$$1673 a = \langle \alpha, \iota, \beta, \theta, c \rangle$$

1674 where the component α refers to an action (with respect to the tool), t is the specific choice of target
 1675 tool, and β is the consequent relevant information. Specifically, we introduce the components as
 1676 follows:
 1677

1678 **Action Type Component (α)** The action type α belongs to a finite set of predefined operations:
 1679

$$1681 \quad \alpha \in \mathcal{A}_{\text{type}} = \{\text{INVOC_TOOL}, \text{VALIDATE_OUTPUT}, \text{RETRY_TOOL}, \\ 1682 \quad \text{RECOVER_ERROR}, \text{CHECK_DEPENDENCIES}, \\ 1683 \quad \text{CREATE_CHECKPOINT}, \text{RESTORE_CHECKPOINT}, \\ 1684 \quad \text{NO_OP}, \text{PARALLEL_EXECUTE}\}$$

1686 **Tool Target Component (ι)** The tool identifier $\iota \in \mathcal{T} \cup \{\emptyset\}$ specifies the target tool for execution,
 1687 where \mathcal{T} represents the available tool set and \emptyset indicates no specific tool target.
 1688

1689 **Semantic Enhancement Component (β)** The RAG-enhanced component $\beta = \langle s_{\text{sem}}, \mathcal{S}_{\text{src}}, \mathcal{T}_{\text{alt}} \rangle$
 1690 contains:
 1691

- 1692 • $s_{\text{sem}} \in [0, 1]$: Semantic relevance score from embedding-based search
- 1693 • $\mathcal{S}_{\text{src}} \in \{\text{rule, embedding, hybrid, pattern}\}$: Information source type
- 1694 • $\mathcal{T}_{\text{alt}} \subseteq \mathcal{T}$: Alternative tool candidates
 1695

1696 **Parameters Component (θ)** Action parameters θ contain execution-specific configuration as a
 1697 key-value mapping, enabling flexible parameterization for different action types.
 1698

1699 **Confidence Component (c)** The confidence score $c \in [0, 1]$ represents the system's belief in the
 1700 action's appropriateness for the current state, computed through multi-factor assessment.
 1701

1702 D.2.1 ACTION FILTERING MECHANISM

1703 Valid actions at state s_t are determined through a multi-stage filtering process $\mathcal{F} : \mathcal{S} \rightarrow 2^{\mathcal{A}}$:

$$1706 \quad \mathcal{A}_{\text{valid}}(s_t) = \mathcal{F}_{\text{semantic}} \circ \mathcal{F}_{\text{dependency}} \circ \mathcal{F}_{\text{constraint}}(\mathcal{A})$$

1707 where:

1709 **Constraint Filter ($\mathcal{F}_{\text{constraint}}$):** Removes actions violating basic execution constraints:

$$1710 \quad \mathcal{F}_{\text{constraint}}(\mathcal{A}) = \{a \in \mathcal{A} : \text{status}(a, \iota) \neq \text{SUCCESS} \wedge \\ 1711 \quad \text{retries}(a, \iota) < 3 \wedge \\ 1712 \quad \text{parallel_safe}(a, \iota) \vee |\text{running_tools}(s_t)| = 0\}$$

1714 Here a, ι represents the choice of tool ι of action a .

1716 **Dependency Filter ($\mathcal{F}_{\text{dependency}}$):** Ensures prerequisite tools are successfully executed:

$$1717 \quad \mathcal{F}_{\text{dependency}}(\mathcal{A}') = \{a \in \mathcal{A}' : \forall d \in \text{deps}(a, \iota), \psi_d = \text{SUCCESS}\}$$

1719 **Semantic Filter ($\mathcal{F}_{\text{semantic}}$):** Applies multi-source confidence scoring:

$$1720 \quad \mathcal{F}_{\text{semantic}}(\mathcal{A}'') = \{a \in \mathcal{A}'' : c_{\text{composite}}(s_t, a) > \tau_{\text{threshold}}\}$$

1722 The composite confidence combines multiple information sources:

$$1724 \quad c_{\text{composite}}(s_t, a) = \sum_i w_i \cdot c_i(s_t, a) \quad (1)$$

1726 with weights $\mathbf{w} = [w_{\text{rule}}, w_{\text{embed}}, w_{\text{pattern}}, w_{\text{task}}] = [0.25, 0.30, 0.25, 0.20]$ and individual confidence
 1727 components:

1728
 1729
 1730 $c_{\text{rule}}(s_t, a) = f_{\text{alignment}}(\omega, f_{\text{task}}, \text{ops}(a.\iota))$
 1731 $c_{\text{embed}}(s_t, a) = \xi \cdot \mathbf{E}_{\text{cache}}[a.\iota]$
 1732 $c_{\text{pattern}}(s_t, a) = \max_{p \in \mathcal{P}} p_{\text{score}} \cdot \mathbf{1}[a.\iota \text{ extends } p]$
 1733 $c_{\text{task}}(s_t, a) = \text{preference}(\tau, \text{task_type}, a.\iota)$
 1734

1735 **D.2.2 FALBACK ACTION GENERATION**

1736 When $|\mathcal{A}_{\text{valid}}(s_t)| \leq 1$ (only NO_OP available), the system employs progressive fallback strategies:

1738 **Algorithm 1** Fallback Action Generation

1740 1: **if** $\xi \cdot \mathcal{T}_{\text{candidates}} \neq \emptyset$ **then**
 1741 2: Add top-2 RAG candidates with $c = 0.35$
 1742 3: **else if** $\text{workflow_step} < 10$ **then**
 1743 4: Add dependency-free tools with $c = 0.30$
 1744 5: **else**
 1745 6: Force recovery actions: $\alpha \in \{\text{RECOVER_ERROR}, \text{RESTORE_CHECKPOINT}\}$
 1746 7: **end if**

1747 **D.2.3 ACTION SPACE CARDINALITY**

1749 The total action space size scales as:

$$|\mathcal{A}| = |\mathcal{A}_{\text{type}}| \times (|\mathcal{T}| + 1) + |\mathcal{A}_{\text{meta}}|$$

1752 where $|\mathcal{A}_{\text{meta}}| = 4$ represents tool-independent actions (NO_OP, CREATE_CHECKPOINT, etc.).
 1753 However, the effective action space at any state s_t is typically $|\mathcal{A}_{\text{valid}}(s_t)| \ll |\mathcal{A}|$ due to filtering
 1754 constraints.

1755 This structured action representation enables the MDP to maintain semantic coherence while pro-
 1756 viding sufficient flexibility for complex workflow orchestration scenarios.

1758 **D.3 MULTI-PHASE ADAPTIVE REWARD FUNCTION**

1760 The reward function implements a sophisticated two-phase training strategy that adapts based on the
 1761 agent’s learning progress. This design addresses the fundamental challenge of learning both *what*
 1762 tools to use and *when* to use them.

1764 **D.3.1 PHASE-ADAPTIVE STRATEGY**

1766 The core insight is that tool selection learning requires different optimization objectives at different
 1767 stages:

1768
 1769
$$R(s_t, a_t, s_{t+1}) = \begin{cases} R_{\text{coverage}}(s_t, a_t, s_{t+1}) & \text{if } \rho_{\text{success}} < \theta_{\text{adapt}} \\ R_{\text{sequence}}(s_t, a_t, s_{t+1}) & \text{otherwise} \end{cases}$$

 1770

1772 where ρ_{success} represents the current success rate and $\theta_{\text{adapt}} = 0.3$ is the adaptation threshold.

1773 **Phase I: Coverage-Focused Learning** ($\rho_{\text{success}} < 0.3$)

1774 During initial learning, the agent must discover which tools are relevant for different task types. The
 1775 reward function prioritizes exploration and tool discovery:

1778
$$R_{\text{coverage}} = R_{\text{exploration}} + R_{\text{discovery}} + R_{\text{completion}}$$

 1779
$$R_{\text{exploration}} = \alpha_{\text{attempt}} \cdot \mathbf{1}[\text{tool attempted}] + \alpha_{\text{novel}} \cdot \mathbf{1}[\text{first attempt}]$$

 1780
$$R_{\text{discovery}} = \beta_{\text{required}} \cdot \mathbf{1}[\iota \in \mathcal{T}_{\text{required}}] \cdot \mathbf{1}[\text{success}]$$

 1781
$$R_{\text{completion}} = \gamma_{\text{progress}} \cdot \Delta p + \delta_{\text{milestone}} \cdot |\mathcal{M}_{t+1} - \mathcal{M}_t|$$

1782 with reward weights: $\alpha_{\text{attempt}} = 3$, $\alpha_{\text{novel}} = 30$, $\beta_{\text{required}} = 50$, $\gamma_{\text{progress}} = 100$, $\delta_{\text{milestone}} = 40$.
 1783

1784 The key insight is that *any* successful execution of required tools receives substantial rewards, re-
 1785 gardless of execution order.

1786 Phase II: Sequence-Optimized Learning ($\rho_{\text{success}} \geq 0.3$)

1787 Once basic tool usage is learned, the focus shifts to optimizing execution sequences and workflow
 1788 efficiency:
 1789

$$1790 \\ 1791 R_{\text{sequence}} = R_{\text{order}} + R_{\text{efficiency}} + R_{\text{completion}} \\ 1792 \\ 1793 R_{\text{order}} = \begin{cases} 15 \cdot |\mathcal{T}_{\text{correct.seq}}| & \text{if perfect sequence order} \\ 1794 5 \cdot |\mathcal{T}_{\text{near.seq}}| & \text{if near-correct order} \\ 1795 -5 \cdot \sum_{\iota} |i_{\text{actual}}(\iota) - i_{\text{expected}}(\iota)| & \text{otherwise} \end{cases} \\ 1796 \\ 1797 R_{\text{efficiency}} = \eta_{\text{step}} \cdot \frac{\max(0, k_{\text{target}} - k)}{k_{\text{target}}} + \eta_{\text{error}} \cdot \mathbf{1}[e_{\text{total}} = 0]$$

1798 where $i_{\text{actual}}(\iota)$ and $i_{\text{expected}}(\iota)$ represent actual and expected positions of tool ι in the execution
 1799 sequence.
 1800

1801 D.3.2 UNIVERSAL REWARD COMPONENTS

1803 Several reward components operate consistently across both phases:

1804 Progress Incentives:

$$1805 R_{\text{progress}} = \gamma_{\text{base}} \cdot \Delta p + \gamma_{\text{early}} \cdot \Delta p \cdot \mathbf{1}[k < 10] + \gamma_{\text{late}} \cdot \Delta p \cdot \mathbf{1}[p > 0.8]$$

1806 RAG-Enhanced Semantic Alignment:

$$1807 R_{\text{semantic}} = \lambda_{\text{rag}} \cdot s_{\text{rag}}(\iota) + \lambda_{\text{pattern}} \cdot s_{\text{pattern}}(\mathbf{e}_{\text{seq}}, \iota) + \lambda_{\text{task}} \cdot s_{\text{task}}(\tau, \iota)$$

1809 where $s_{\text{rag}}(\iota)$ is the RAG similarity score, s_{pattern} captures learned sequential patterns, and s_{task} rep-
 1810 resents task-tool alignment.
 1811

1812 Terminal Rewards:

1814 Upon episode completion, substantial rewards are distributed based on both success and training
 1815 phase:
 1816

$$1817 \\ 1818 R_{\text{terminal}} = \begin{cases} 150 + 50 \cdot r_{\text{coverage}} & \text{if coverage phase and success} \\ 1819 100 + 50 \cdot r_{\text{sequence}} & \text{if sequence phase and success} \\ 1820 \max(0, 50 \cdot p_{\text{final}}) & \text{if failure} \end{cases}$$

1821 where $r_{\text{coverage}} = \frac{|\mathcal{T}_{\text{executed}} \cap \mathcal{T}_{\text{required}}|}{|\mathcal{T}_{\text{required}}|}$ and r_{sequence} measures sequence correctness.
 1822

1823 D.3.3 PENALTY STRUCTURE

1825 The penalty system is designed to be *adaptive* and *minimal* during early learning:
 1826

$$1827 \\ 1828 R_{\text{penalty}} = R_{\text{error}} + R_{\text{repetition}} + R_{\text{stagnation}} \\ 1829 \\ 1830 R_{\text{error}} = \begin{cases} 0 & \text{if } \rho_{\text{success}} < 0.1 \\ 1831 -2 \cdot \Delta e_{\text{total}} & \text{if } 0.1 \leq \rho_{\text{success}} < 0.3 \\ 1832 -5 \cdot \Delta e_{\text{total}} & \text{otherwise} \end{cases} \\ 1833 \\ 1834 R_{\text{repetition}} = -\kappa \cdot (\text{count}(\iota, \mathbf{e}_{\text{seq}}) - 1) \cdot \mathbf{1}[\text{count}(\iota) \geq 2] \\ 1835 R_{\text{stagnation}} = -10 \cdot \mathbf{1}[a_t = a_{t-1} = \text{NO_OP}]$$

1836 where $\kappa = 1$ during coverage phase and $\kappa = 5$ during sequence phase.
 1837

- 1836 D.3.4 KEY DESIGN PRINCIPLES
 1837
 1838 This reward architecture embodies several key principles:
 1839
 1840 • **Progressive Complexity**: Early learning focuses on tool discovery; later learning optimizes execution patterns
 1841
 1842 • **Semantic Guidance**: RAG-enhanced rewards align tool selection with task semantics
 1843
 1844 • **Adaptive Penalties**: Error tolerance decreases as competence increases
 1845
 1846 • **Terminal Differentiation**: Final rewards depend on both success and current learning objectives

1847 The phase transition at $\rho_{\text{success}} = 0.3$ ensures that agents first master basic tool usage before attempting
 1848 to optimize execution sequences, leading to more stable and efficient learning convergence.
 1849

1850 D.4 MDP ENVIRONMENT DYNAMICS

1852 This section details the core environment mechanics that govern state transitions, completion criteria,
 1853 and adaptive behavior in our MDP framework. Unlike conventional MDP environments with
 1854 static rules, our system implements dynamic thresholds and curriculum-adaptive mechanisms that
 1855 evolve during training.

1856 D.4.1 ENVIRONMENT STEP EXECUTION

1858 The environment step function orchestrates the complete MDP transition process through three sequential phases:
 1859

$$1861 \quad 1862 \quad \text{MDPStep}(s_t, a_t) = \langle s_{t+1}, r_t, \text{done} \rangle$$

1863 where each component is computed through dedicated sub-procedures:
 1864

$$1866 \quad s_{t+1} = \text{StateTransition}(s_t, a_t) \\ 1867 \quad r_t = \text{AdaptiveReward}(s_t, a_t, s_{t+1}) \\ 1868 \quad \text{done} = \text{CompletionCheck}(s_{t+1})$$

1870 Phase I: State Transition ($s_{t+1} = \text{StateTransition}(s_t, a_t)$)

1872 The state transition simulates tool execution through an adaptive reliability model that adjusts success
 1873 probability based on training progress:

$$1876 \quad \rho_{\text{success}}(\iota, s_t) = \rho_{\text{base}} \cdot \prod_{d \in \text{deps}(\iota)} \gamma_{\text{dep}} \cdot \gamma_{\text{retry}}^{r_{\iota}} \cdot \gamma_{\text{semantic}} \\ 1877 \\ 1878 \quad \rho_{\text{base}} = \begin{cases} 0.95 & \text{if } \rho_{\text{episode}} < 0.1 \\ 0.90 & \text{if } 0.1 \leq \rho_{\text{episode}} < 0.3 \\ 0.85 & \text{otherwise} \end{cases}$$

1882 where ρ_{episode} represents the current episode success rate, $\gamma_{\text{dep}} = 0.8$ penalizes unmet dependencies,
 1883 $\gamma_{\text{retry}} = 0.95$ reduces reliability with retry attempts r_{ι} , and $\gamma_{\text{semantic}} \in [1.0, 1.3]$ rewards semantic
 1884 task-tool alignment. We note that this differs from the tool result simulator probability settings B.3,
 1885 which enables a testing environment different from the training environment.
 1886

1887 The transition updates multiple state components:

- 1888 • Tool execution states: $\psi_{\iota}^{t+1} \sim \text{Bernoulli}(\rho_{\text{success}}(\iota, s_t))$
- 1889 • Execution sequence: $\mathbf{e}_{\text{seq}}^{t+1} = \mathbf{e}_{\text{seq}}^t \cup \{\iota\}$ if successful

- 1890 • Error tracking: $e_{\text{consecutive}}^{t+1}$, e_{total}^{t+1} updated based on outcome
 1891 • Data flow state progression: ξ_{flow}^{t+1} advanced according to semantic operations
 1892

1893 **Phase II: Progress Computation (p_{t+1} within s_{t+1})**

1894 Progress update adapts to task structure through dual calculation modes:

1895

$$1896 p_{t+1} = \begin{cases} \frac{|\mathcal{T}_{\text{executed}} \cap \mathcal{T}_{\text{required}}|}{|\mathcal{T}_{\text{required}}|} + \beta_{\text{sequence}} & \text{if } \mathcal{T}_{\text{required}} \neq \emptyset \\ 0.3 \cdot p_{\text{milestone}} + 0.4 \cdot p_{\text{tool}} + 0.3 \cdot p_{\text{step}} & \text{otherwise} \end{cases}$$

1897

1898 where $\beta_{\text{sequence}} = 0.1$ provides sequence order rewards, $p_{\text{milestone}} = \frac{|\mathcal{M}_{\text{achieved}}|}{|\mathcal{M}_{\text{expected}}|}$, $p_{\text{tool}} = \min(1.0, \frac{|\text{successful_tools}|}{5})$, and $p_{\text{step}} = \min(1.0, \frac{k}{20})$.

1899 Progress is monotonically increasing: $p_{t+1} = \max(p_t, p_{\text{computed}})$ to prevent regression.

1900 **Phase III: Reward Calculation ($r_t = \text{AdaptiveReward}(s_t, a_t, s_{t+1})$)**

1901 The adaptive reward system operates through the two-phase strategy detailed in Section D.3, incorporating:

- 1902
- 1903 • Base exploration rewards for any non-NO_OP action
 - 1904 • Tool execution rewards scaled by training phase
 - 1905 • Progress increment rewards: $100 \cdot (p_{t+1} - p_t)$
 - 1906 • Required tool coverage bonuses (coverage phase) or sequence order bonuses (sequence phase)
 - 1907 • Semantic alignment rewards from RAG-enhanced action selection

1908 **Phase IV: Completion Assessment (done = CompletionCheck(s_{t+1}))**

1909 Episode termination employs hierarchical completion criteria:

1910

$$1911 \text{done} = (p_{t+1} \geq 1.0) \vee \\ 1912 \quad (\mathcal{T}_{\text{required}} \subseteq \mathcal{T}_{\text{executed}}) \vee \\ 1913 \quad (k > k_{\text{max}}) \vee \\ 1914 \quad (e_{\text{consecutive}} > e_{\text{max}}) \vee \\ 1915 \quad (|\mathcal{A}_{\text{valid}}(s_{t+1})| \leq 1)$$

1916

1917 where k_{max} and e_{max} are curriculum-adaptive thresholds. Success determination considers both completion method and achieved progress:

1918

$$1919 \text{success} = \begin{cases} \text{True} & \text{if } p_{t+1} \geq 0.95 \vee \mathcal{T}_{\text{required}} \subseteq \mathcal{T}_{\text{executed}} \\ 1920 \quad p_{t+1} \geq 0.5 & \text{if timeout or deadlock} \\ 1921 \quad \text{False} & \text{otherwise} \end{cases}$$

1922

1923 This four-phase execution framework ensures consistent state evolution while maintaining curriculum-appropriate difficulty and comprehensive performance assessment.

1924 **D.4.2 CURRICULUM-ADAPTIVE TRAINING**

1925 Additionally, the training process implements dynamic curriculum adjustment based on performance metrics:

1926 **Curriculum Stage Transitions:**

1927

$$1928 \text{stage}_{i+1} = \begin{cases} \text{stage}_i + 1 & \text{if } \rho_{\text{success}}^{(n)} > \theta_{\text{advance}} \wedge n \geq n_{\text{min}} \\ 1929 \quad \max(0, \text{stage}_i - 1) & \text{if } \rho_{\text{success}}^{(n)} < \theta_{\text{regress}} \\ 1930 \quad \text{stage}_i & \text{otherwise} \end{cases}$$

1931

1944 where i indexes curriculum update events, and $\rho_{\text{success}}^{(n)}$ represents the success rate over the last n
 1945 episodes. We have advancement threshold $\theta_{\text{advance}} = 0.7$, regression threshold $\theta_{\text{regress}} = 0.3$, and
 1946 minimum episodes $n_{\min} = 50$.

1947 Each curriculum stage modifies completion criteria given in table 12.

1949
 1950 Table 12: Curriculum-dependent completion thresholds.

Stage	Min Progress	Max Errors	Required Coverage
0	0.1	50	0.2
1	0.3	30	0.4
2	0.5	20	0.7
3	0.7	15	0.9
4+	0.7	10	1.0

1959 D.5 POLICY OPTIMIZATION FRAMEWORK AND MDP TRAINING

1961 The MDP training implements a sophisticated optimization framework that combines policy gradient
 1962 methods with curriculum learning and RAG-enhanced decision making. The optimization process
 1963 operates through carefully orchestrated phases that build upon the adaptive reward structure.
 1964 The policy update process operates directly on the multi-source confidence composition framework
 1965 established earlier, optimizing the neural network that learns to weight and combine different information
 1966 sources for action selection.

1967 Policy Architecture and Parameters:

1969 The trainable policy π_θ is implemented as a neural network that takes the encoded state representation
 1970 $\mathbf{s}_t \in \mathbb{R}^d$ and outputs action probabilities over the filtered valid action space $\mathcal{A}_{\text{valid}}(s_t)$.

1971 Critically, the policy network learns to implicitly weight the confidence components defined earlier:

$$1974 \quad c_{\text{learned}}(s_t, a_t) = \pi_\theta(a_t | s_t) \approx f_\theta \left(\sum_j w_j \cdot c_j(s_t, \iota) \right)$$

1978 where $c_j \in \{c_{\text{rule}}, c_{\text{embed}}, c_{\text{pattern}}, c_{\text{task}}\}$ are the confidence components defined in Section D.2.1, and
 1979 f_θ represents the learned non-linear transformation.

1980 Objective Function Correspondence:

1982 The optimization objective directly corresponds to maximizing the expected cumulative reward defined in Section D.3.1:

$$1986 \quad J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^T R_{\text{adaptive}}(s_t, a_t, s_{t+1}) \right]$$

$$1989 \quad = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^T (R_{\text{coverage}}(s_t, a_t, s_{t+1}) \cdot \mathbf{1}[\rho_{\text{success}} < 0.3] + R_{\text{sequence}}(s_t, a_t, s_{t+1}) \cdot \mathbf{1}[\rho_{\text{success}} \geq 0.3]) \right]$$

1992 where R_{coverage} and R_{sequence} are the phase-specific reward functions detailed in equations (8)-(12)
 1993 of Section 4.3.

1995 D.5.1 EPISODE REWARD ADJUSTMENT

1997 Post-episode reward adjustment propagates final performance back through the trajectory:

1998

1999

2000

2001

2002

2003

2004

2005

2006

2007

2008

2009

2010

2011

2012

2013

2014

2015

2016

2017

2018

2019

$$r'_t = r_t \cdot \mu_{\text{performance}} + \lambda_{\text{position}} \cdot (1 - \frac{t}{T}) + \delta_{\text{rag}} + \delta_{\text{pattern}}$$

$$\mu_{\text{performance}} = \begin{cases} 1.5 & \text{if } \text{score}_{\text{final}} \geq 0.9 \\ 1.2 & \text{if } \text{score}_{\text{final}} \geq 0.7 \\ 1.0 & \text{if } \text{score}_{\text{final}} \geq 0.5 \\ 0.7 & \text{otherwise} \end{cases}$$

where δ_{rag} rewards RAG-guided selections and δ_{pattern} rewards pattern completion.

Gradient Computation with Experience Replay:

The policy gradient incorporates the episode reward adjustment mechanism:

$$\nabla_{\theta} J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta}} \left[\sum_{t=0}^T \nabla_{\theta} \log \pi_{\theta}(a_t | s_t) \cdot r'_t \right]$$

$$r'_t = r_t \cdot \mu_{\text{performance}} + \lambda_{\text{position}} \cdot (1 - \frac{t}{T}) + \delta_{\text{rag}} + \delta_{\text{pattern}}$$

where:

- $r_t = R_{\text{adaptive}}(s_t, a_t, s_{t+1})$ is the immediate reward from Section D.3.1
- $\mu_{\text{performance}} \in \{0.7, 1.0, 1.2, 1.5\}$ scales based on final episode score
- δ_{rag} provides additional reward when actions align with RAG-enhanced confidence $c_{\text{embed}}(s_t, t)$
- δ_{pattern} rewards actions that complete successful sequential patterns from $\mathcal{P}_{\text{successful}}$

Curriculum-Adaptive Learning Rate:

The learning rate adapts based on curriculum stage and phase transition status:

$$\alpha_{\text{policy}} = \alpha_{\text{base}} \cdot \gamma_{\text{curriculum}}^{\text{stage}} \cdot \beta_{\text{phase}}$$

$$\beta_{\text{phase}} = \begin{cases} 1.2 & \text{if recently transitioned to sequence phase} \\ 1.0 & \text{if stable in current phase} \\ 0.8 & \text{if performance declining} \end{cases}$$

where $\alpha_{\text{base}} = 3 \times 10^{-4}$ and $\gamma_{\text{curriculum}} = 0.95$.

Integration with State Encoding:

The policy update directly operates on the state encoding defined in Section 4.1:

$$\mathbf{s}_t = \text{Concat}(\text{encode}(\psi_t), \text{encode}(\phi_t), \omega \cdot f_{\text{task}}, \text{encode}(\xi_t))$$

$$\text{Loss}(\theta) = - \sum_{t=0}^T \log \pi_{\theta}(a_t | \mathbf{s}_t) \cdot r'_t + \beta_{\text{entropy}} \mathcal{H}(\pi_{\theta}(\cdot | \mathbf{s}_t))$$

where $\mathcal{H}(\cdot)$ is the entropy term encouraging exploration, with $\beta_{\text{entropy}} = 0.01$ during coverage phase and $\beta_{\text{entropy}} = 0.005$ during sequence phase.

Convergence and Performance Monitoring:

Policy convergence is monitored through multiple metrics aligned with the curriculum progression:

- Coverage Phase: Convergence measured by $\frac{|\mathcal{T}_{\text{executed}} \cap \mathcal{T}_{\text{required}}|}{|\mathcal{T}_{\text{required}}|} > 0.8$ consistently
- Sequence Phase: Convergence measured by sequence correctness score > 0.7 and step efficiency improvement

The parameter update incorporates gradient clipping ($\|\nabla_{\theta} J(\theta)\|_2 \leq 0.5$) and momentum-based optimization to ensure stable convergence across both training phases.

This policy update framework directly optimizes the multi-source confidence weighting to maximize the phase-adaptive reward signals, enabling the system to learn both tool discovery and sequence optimization through unified gradient-based learning.

D.5.2 EXPERIENCE COLLECTION AND REPLAY

The system maintains episode trajectories for pattern learning:

$$\begin{aligned}\mathcal{D}_{\text{episode}} &= \{(s_t, a_t, r_t, s_{t+1})_{t=0}^T\} \\ \mathcal{D}_{\text{patterns}} &= \text{ExtractPatterns}(\mathcal{D}_{\text{episode}}, \text{score}_{\text{final}}) \\ \mathcal{P}_{\text{successful}} &= \mathcal{P}_{\text{successful}} \cup \mathcal{D}_{\text{patterns}}\end{aligned}$$

Pattern Extraction: Sequential tool patterns are learned from successful episodes:

$$\begin{aligned}\text{pattern}_k &= \text{sequence}[i : i + k] \quad \forall i, k \in \{2, 3\} \\ \text{score}(\text{pattern}_k) &= \frac{\sum_{\text{episodes}} \text{score}_{\text{episode}} \cdot \mathbf{1}[\text{pattern}_k \in \text{episode}]}{\sum_{\text{episodes}} \mathbf{1}[\text{pattern}_k \in \text{episode}]}\end{aligned}$$

D.5.3 GENERATING THE MDP-OPTIMAL WORKFLOW PROMPT

Once an MDP-optimal workflow has been determined by the trained policy, it exists as a structured Python object containing a sequence of tools, dependency information, and rich metadata. To make this workflow actionable by a downstream execution agent, it must be translated into a comprehensive and unambiguous set of instructions. This is achieved through the `generate_mcp_prompt` function, which constructs a detailed prompt formatted according to a Multi-Agent Communication Protocol (MCP). The goal of this prompt is not merely to list the tools, but to provide a complete operational context, including the strategic reasoning, expected outcomes, and contingency plans. The generation process is composed of several automated steps that assemble distinct components of the final prompt.

Core Structure The entire prompt is encapsulated within a root `<mcp_task>` tag, creating a structured, machine-readable format. This structure is populated by several key sections, each generated by a dedicated helper function that extracts and formats information from the final workflow object.

Execution Plan Generation This is the central component of the prompt, outlining the sequence of actions the agent should take. The script employs an intelligent generation method, `_generate_smart_execution_plan`, to create a detailed, step-by-step guide. For each tool in the optimal sequence, this function populates the plan with a rich set of contextual details drawn from the `smart_actions` list in the workflow object:

- **Semantic Rationale:** If a tool was selected via semantic search (RAG), its relevance score is included (e.g., *Semantic match: 95%*). This immediately informs the agent of the confidence in that tool's applicability.
- **Generated Reasoning:** A concise, natural language explanation for the tool's selection is provided. This is generated by the `_generate_tool_reasoning` function, which considers factors like the tool's position in the sequence (e.g., *Initial data loading step*), its semantic capabilities (e.g., *Performs parse, transform operations*), and whether it was a mandatory requirement of the task.

- **Dependencies and Alternatives:** The prompt explicitly lists the direct prerequisites for each tool (e.g., *Requires: file_operations_reader*) and provides a list of alternative tools, ranked by semantic similarity, that can be used if the primary tool fails.
- **Expected Outcome and Confidence:** For each step, a high-level expected outcome is stated (e.g., *Expected: data_validated*), and a composite confidence score, calculated by *_calculate_comprehensive_confidence*, is displayed to manage the agent’s expectations about potential failures.

Intelligence and Semantic Insights To provide the agent with meta-level awareness of the plan’s quality, two sections are generated.

- The *<workflow_intelligence>* section provides a top-level summary, including the overall predicted success probability of the workflow, calculated by *_calculate_success_probability*.
- The *<semantic_insights>* section, generated by *_generate_semantic_insights*, offers a quantitative summary of how Retrieval-Augmented Generation (RAG) influenced the plan. It reports metrics such as the average semantic match score across all tools and the proportion of tools that were selected based on semantic relevance versus learned policy patterns.

Contingency and Execution Guidance The final sections of the prompt provide static but critical instructions to ensure robust execution.

- The *<failure_handling>* block provides a clear, universal protocol for error handling. It instructs the agent to first attempt the primary tool, then cycle through the provided alternatives upon failure, and finally report the issue if all options are exhausted.
- The *<execution_tracking>* block sets the requirements for the agent’s response, instructing it to report its tool selection rationale, any deviations from the plan, and its confidence in each step’s outcome. This ensures that the execution results can be used for future learning and analysis.

Through this multi-faceted generation process, a simple sequence of tools derived from an MDP is transformed into a rich, self-contained operational directive that guides not just the “what” but also the “why”, “how”, and “what if” of task execution.

E DETAILED EXPERIMENTAL RESULTS

E.1 DETAILED MODEL PERFORMANCE ACROSS TASK TYPE

The following table presents model performance across different task types. Performance generally decreases from basic file processing tasks to advanced computation pipelines, though patterns vary across model architectures.

Table 13: Model performance across task types.

Task Type	GPT-4o-mini	GPT-5-mini	Claude-Sonnet-4	Gemini-2.5-Flash	DeepSeek-V3	DeepSeek-R1	Qwen2.5-32B	Llama-3.3-70B
Advanced processing	55.2%	54.1%	51.4%	50.0%	48.3%	50.0%	52.4%	55.8%
API data retrieval	59.0%	54.6%	50.0%	57.1%	49.5%	47.5%	57.1%	64.3%
Batch processing	82.0%	58.7%	60.5%	64.6%	58.6%	50.0%	64.0%	59.8%
Content analysis	77.9%	76.3%	66.0%	70.9%	70.0%	66.1%	82.1%	81.7%
Multi-step processing	67.5%	62.1%	51.7%	59.5%	59.5%	47.3%	72.9%	65.3%

E.2 COMPLETE MODEL PERFORMANCE ON REAL-WORLD TEST SET

In this section, we present the complete model performance table, including results under the baseline prompt and CoT prompt.

2160
2161 Table 14: Prompt type performance comparison: Success rates across different prompt types for
2162 each model.
2163

Model	Baseline			Chain-of-Thought			Optimal Workflow			Flawed Workflow		
	Full	Partial	Fail	Full	Partial	Fail	Full	Partial	Fail	Full	Partial	Fail
GPT-4o-mini	38.9	38.9	22.2	41.7	41.7	16.7	42.1	38.6	19.3	34.3	34.3	31.4
Gemini-2.5-Flash	33.3	33.3	33.3	39.5	39.5	21.1	55.3	41.2	3.5	34.1	33.5	32.4
GPT-5-mini	38.9	38.9	22.2	41.7	41.7	16.7	47.6	45.2	7.1	36.6	36.3	27.0
Llama-3.3-70B	16.7	16.7	66.7	16.7	16.7	66.7	39.6	27.1	33.3	25.4	25.4	49.3
Qwen2.5-32B	41.7	41.7	16.7	43.9	40.4	15.8	55.3	44.7	0.0	42.3	38.8	18.9
DeepSeek-V3	32.1	30.9	37.0	37.8	35.3	26.9	40.2	39.3	20.5	36.0	35.4	28.6
Avg	33.6	33.4	33.0	36.9	35.9	27.3	46.7	39.4	13.9	34.8	34.0	31.3

2172
2173

E.3 SCALING ANALYSIS ON THE REAL WORLD TEST SET

2174
2175 In this section, we present scaling analysis results for the Qwen2.5 series on the real-world test set
2176 under MDP-optimal workflow prompts. Table 15 reveals a dramatic emergence of workflow ex-
2177 ecution capabilities between the 3B and 7B parameter scales. The 3B model exhibits near-complete
2178 failure (93.3% failure rate with only 5.0% full success), indicating insufficient capacity for multi-step
2179 API coordination. A sharp capability transition occurs at 7B parameters, where the model achieves
2180 37.7% full success with failure rate dropping to 28.1%. The 14B model maintains similar perfor-
2181 mance (35.0% full success, 30.0% failure), while the 32B model demonstrates the strongest perfor-
2182 mance with 55.3% full success and zero failures. This non-smooth scaling pattern—characterized by
2183 sharp capability emergence between 3B and 7B, followed by steady improvement to 32B—provides
2184 evidence for emergent workflow execution abilities in real-world API interaction settings, corrobor-
2185 ating our simulation findings (Section 4.4).
21862187
2188 Table 15: Qwen2.5 series scaling analysis with Optimal prompt.

Model Size	Full Success Rate	Partial Success Rate	Failure Rate
Qwen2.5-3B	5.0%	1.7%	93.3%
Qwen2.5-7B	37.7%	34.2%	28.1%
Qwen2.5-14B	35.0%	35.0%	30.0%
Qwen2.5-32B	55.3%	44.7%	0.0%

2193
2194

E.4 REAL-WORLD API FAILURE DISTRIBUTION

2195
2196 We present a record of 974 total API calls across 23 different live APIs, achieving an overall success
2197 rate of 51.3% (500 successful calls, 474 failures). This 48.7% natural failure rate demonstrates the
2198 inherent uncertainty in real-world API interactions that our benchmark aims to capture. Unlike the
2199 simulated tools in our main experiments, these APIs execute actual HTTP requests to live endpoints,
2200 exposing agents to real-world system behaviors. The error modes and system behaviors directly
2201 come from real-world API interactions. Table 16 shows the distribution of error types observed
2202 during our experiments.
22032204
2205 Table 16: Error Type Distribution in API Calls

Error Type	Counts	Percentage of Failures
TIMEOUT	337	71.10%
OPERATION_FAILED	41	8.65%
INVALID_INPUT	27	5.70%
INVALID_RESPONSE	21	4.43%
NETWORK_ERROR	21	4.43%
RATE_LIMIT_ERROR	1	0.21%

2212
2213 We now present an explanation of all the error types presented in the table. The predominant error
2214 type, `TIMEOUT`, reflects our implementation of a 30-second timeout threshold for API responses.

2214 This design choice aligns with common industry practices for production systems, where timeout
 2215 mechanisms are essential safeguards against indefinite waiting and resource exhaustion. Import-
 2216 antly, the occurrence of timeout errors depends entirely on real-world network conditions, server
 2217 load, and backend processing times, making these failures authentic reflections of the unpredictable
 2218 nature of distributed systems rather than artificial constraints. `OPERATION_FAILED` errors result
 2219 from Python exceptions during response processing, such as JSON parsing failures or encoding er-
 2220 rors. `INVALID_INPUT` are mapped from HTTP 400 status codes returned by APIs when request
 2221 parameters are missing or malformed. `INVALID_RESPONSE` errors occur when APIs return un-
 2222 parseable content, such as empty response bodies or data that doesn't match expected schemas.
 2223 `NETWORK_ERROR` are triggered by issues including DNS resolution failures and connection re-
 2224 fusals. `RATE_LIMIT_ERROR` directly map to HTTP 429 responses when API rate limits are ex-
 2225 ceeded.

F LIMITATIONS

2226 Our simulation uses a simplified probabilistic failure model (base success rate 0.8 with penalty
 2227 adjustments) that does not capture real-world complexities such as input-dependent errors, time-
 2228 varying reliability, or realistic recovery mechanisms like exponential backoff. These simplifications
 2229 balance reproducibility, scalability, and our focus on instruction quality variation. While the real-
 2230 world task set (Section 3.4) shows simulation rankings correlate with real-world performance, future
 2231 work could incorporate input-conditioned failure probabilities, explicit rate limiting, and correlated
 2232 failure patterns.

G THE USE OF LARGE LANGUAGE MODELS

2233 In adherence to the ICLR 2026 policy on the use of Large Language Models (LLMs), we disclose
 2234 that an LLM was utilized as a specific component within our benchmark construction methodology.

2235 **Role in Code Development.** The implementation of our experimental framework, including the
 2236 automated task generation pipeline and the simulation environment, was expedited with the assis-
 2237 tance of LLM-based coding tools. Specifically, we utilized Anthropic's Claude for generating code
 2238 snippets, debugging complex logic, and refactoring. These tools acted as programming assistants,
 2239 and all final code was reviewed, validated, and integrated by the human authors.

2240 **Role in Manuscript Preparation.** Our writing process for the appendices involved a collaborative
 2241 human-LLM workflow. The initial drafts of all appendix sections were written by the human authors
 2242 to ensure the factual and technical accuracy of the content. These drafts were then processed by an
 2243 LLM for the purpose of polishing the language, improving formal structure, and enhancing clarity.
 2244 Following the LLM's revisions, the authors conducted a final, thorough review to make critical edits,
 2245 verify all statements, and ensure the text precisely reflected our methodology and findings.

2246 The core research ideation, experimental design, analysis of results, and the primary drafting of the
 2247 main manuscript were conducted by the human authors. We take full and final responsibility for all
 2248 content presented in this paper, including any code and text produced with the assistance of LLMs.

2249
 2250
 2251
 2252
 2253
 2254
 2255
 2256
 2257
 2258
 2259
 2260
 2261
 2262
 2263
 2264
 2265
 2266
 2267