

Autonomous Data Processing using Meta-Agents

Udayan Khurana

ukhurana@us.ibm.com

IBM TJ Watson Research Center

Yorktown Heights, NY, USA

Abstract

Traditional data processing pipelines are typically static and hand-crafted for specific tasks, limiting their adaptability to evolving requirements. While general-purpose agents and coding assistants can generate code for well-understood data pipelines, they lack the ability to autonomously monitor, manage, and optimize an end-to-end pipeline once deployed. We present **Autonomous Data Processing using Meta-agents** (ADP-MA), a framework that dynamically constructs, executes, and iteratively refines data processing pipelines through hierarchical agent orchestration. At its core, *meta-agents* analyze input data and task specifications to design a multi-phase plan, instantiate specialized *ground-level agents*, and continuously evaluate pipeline performance. The architecture comprises three key components: a planning module for strategy generation, an orchestration layer for agent coordination and tool integration, and a monitoring loop for iterative evaluation and backtracking. Unlike conventional approaches, ADP-MA emphasizes context-aware optimization, adaptive workload partitioning, and progressive sampling for scalability. Additionally, the framework leverages a diverse set of external tools and can reuse previously designed agents, reducing redundancy and accelerating pipeline construction. We demonstrate ADP-MA through an interactive demo that showcases pipeline construction, execution monitoring, and adaptive refinement across representative data processing tasks.

CCS Concepts

- Information systems → Data management systems; • Computing methodologies → Multi-agent systems.

Keywords

autonomous data processing, multi-agent systems, large language models, pipeline orchestration, meta-agents

ACM Reference Format:

Udayan Khurana. 2026. Autonomous Data Processing using Meta-Agents. In . ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/nnnnnnnn>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Conference'17, Washington, DC, USA

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM. ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
<https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

1 Introduction

Data processing pipelines form the backbone of modern data-driven applications, transforming raw data into actionable insights. Traditional approaches to constructing these pipelines rely heavily on manual engineering, where domain experts design static workflows tailored to specific use cases. While effective for well-defined scenarios, this paradigm suffers from several critical limitations: inflexibility to changing data characteristics, inability to handle data drift, high development overhead for new tasks, and lack of autonomous error recovery.

Recent advances in large language models (LLMs) and agent-based systems have opened new possibilities for automating data science workflows [3, 9]. However, existing solutions primarily focus on single-point assistance or predefined pipeline templates, requiring substantial human intervention to orchestrate complex, multi-stage data processing tasks. General-purpose coding assistants can generate pipeline code but cannot autonomously monitor execution, adapt to runtime failures, or optimize performance based on observed data characteristics.

We introduce ADP-MA (Autonomous Data Processing using Meta-agents), a framework that addresses these limitations through hierarchical agent orchestration. ADP-MA distinguishes itself through three fundamental innovations. First, we employ meta-agents that reason about the overall data processing strategy, decomposing complex tasks into manageable phases while maintaining global coherence. Second, we dynamically instantiate and coordinate ground-level agents that execute specialized operations, with each agent equipped with domain-specific tools and capabilities. Third, we implement continuous monitoring and adaptive refinement mechanisms that enable the system to detect anomalies, handle data drift, and backtrack when necessary.

The meta-agent architecture operates at a higher level of abstraction than traditional agents. Rather than directly executing data operations, meta-agents perform strategic planning, agent creation and modification, and system-level monitoring. This separation of concerns enables more sophisticated reasoning about workflow design while delegating execution details to specialized ground-level agents. The framework maintains a library of reusable agent definitions and tools, accelerating development for similar tasks and promoting consistency across pipelines.

We incorporate four key design principles that differentiate ADP-MA from existing work. *Hierarchical planning* breaks large tasks into phases of reasonable size and logical consistency, optimizing context usage while avoiding issues like context rot [1]. *Iterative refinement* employs multiple rounds of planning, critiquing, and revision to reduce errors before code generation. *Progressive sampling* enables the system to test pipelines incrementally, revealing challenges at each stage and facilitating scalability. *Intelligent backtracking* provides mechanisms to reconsider higher-level decisions

when ground-level execution encounters intractable failures. ADP-MA is built with the Google Agent Development Kit (ADK) [5] and implements its own process-level sandboxing for safe code execution. However, the ideas and design, including agentic definitions, are independent of the choice of platform.

The contributions of this work include: (1) a hierarchical meta-agent architecture for autonomous data pipeline construction and management, (2) mechanisms for dynamic agent instantiation, tool integration, and adaptive workflow refinement, (3) strategies for context-aware optimization, progressive sampling, and intelligent backtracking, and (4) an interactive demonstration system that showcases these capabilities across representative data processing scenarios.

2 Related Work

The landscape of autonomous data processing has evolved rapidly with the advent of LLM-based agents. We organize related work into several key categories that contextualize our contributions.

2.1 Multi-Agent Systems for Data Science

AutoKaggle [9] represents a significant advancement in autonomous data science, employing a phase-based workflow with five specialized agents (Reader, Planner, Developer, Reviewer, and Summarizer) to complete end-to-end data science competitions. The framework divides tasks into six phases including background understanding, exploratory data analysis, data cleaning, feature engineering, and model building, achieving a validation submission rate of 0.85 and comprehensive score of 0.82 across eight Kaggle competitions. While AutoKaggle demonstrates the effectiveness of multi-agent collaboration for predictive modeling, it relies on predefined phase sequences and focuses primarily on machine learning tasks. ADP-MA extends this paradigm by dynamically determining workflow phases based on task analysis and supporting broader classes of data processing beyond predictive modeling.

Recent surveys on data agents [3] have proposed hierarchical taxonomies for classifying agent autonomy levels, ranging from L0 (manual operation) to L5 (fully autonomous systems). Most existing work clusters at L1-L3 levels, providing assistance for isolated tasks or executing predefined procedures. ADP-MA advances toward L4 autonomy by enabling self-orchestration of multi-stage pipelines with minimal human intervention, though full L5 autonomy with generative capabilities across arbitrary data tasks remains aspirational.

2.2 Automated Data Cleaning and Preparation

Data cleaning represents a particularly labor-intensive aspect of pipeline development. AutoDCWorkflow [8] introduces an LLM-based pipeline for automatically generating data cleaning workflows using OpenRefine operations. The system employs three iterative components—selecting target columns, inspecting column quality, and generating operations with arguments—to address quality issues including duplicates, missing values, and inconsistent formats. Experiments demonstrate that models like Llama 3.1 and Gemma 2 can generate high-quality cleaning workflows that effectively prepare datasets for downstream analysis.

AutoClean [14] takes a complementary approach by configuring LLMs as agent teams that mimic human data-cleaning processes, automatically generating cleaning rules from limited samples that can then be applied broadly to large corpora. This work demonstrates the feasibility of using LLMs to prepare training data at scale, including Common Crawl and specific target websites.

Research on the interaction between data cleaning and automated machine learning [13] reveals that cleaning decisions should be considered jointly with other pipeline hyperparameters. Systems like AutoClean extend AutoSklearn with sophisticated outlier detection and context-dependent imputation strategies, showing that cleaning preprocessors can be optimized alongside feature selection and model choice. ADP-MA builds on these insights by treating data preparation as an integral, adaptively configured phase within the broader pipeline rather than a standalone preprocessing step.

2.3 Benchmarks for Data Processing Automation

The evaluation of autonomous data processing systems requires comprehensive benchmarks that capture diverse task types, data modalities, and quality dimensions. We survey the principal benchmarks relevant to ADP-MA and describe how they inform our evaluation strategy.

KRAMABENCH [11] is a benchmark for knowledge-intensive, real-world, multi-step data science pipelines. It comprises 104 manually curated tasks drawn from approximately 1,700 data files across 24 sources in six scientific and legal domains (archaeology, astronomy, biomedical, environment, legal, wildfire), totaling 1.7 GB of heterogeneous data. Each task requires an end-to-end pipeline that performs data discovery, integration, wrangling, and statistical reasoning before producing a final answer. Tasks are organized into 633 subtasks and split roughly 60/40 between hard and easy difficulty. Answer types include exact and approximate numeric values, exact and approximate lists, and exact strings. The benchmark reports baselines for several agentic systems: *smolagents* (iterative) achieves 50% end-to-end accuracy, DS-GURU with self-correction reaches 22%, and a GPT-o3 design-only baseline achieves 42%. KRAMABENCH is particularly relevant because it stresses multi-step pipeline orchestration—the core capability of ADP-MA’s meta-agent architecture—and provides established baselines for comparative evaluation.

DA-Code [7] is a benchmark of 500 complex data science code generation examples. Tasks span three categories: data wrangling (20%), machine learning (20%), and exploratory data analysis (60%), with a difficulty distribution of 22.8% easy, 57.3% medium, and 19.9% hard. Evaluation is execution-based: agent-generated code is run inside a Docker sandbox, and outputs are compared against ground-truth tables, charts, and ML metrics. Key metrics include completion rate, executable code percentage, table match score, and a normalized ML score. DA-Code’s emphasis on end-to-end code generation from natural-language specifications makes it a natural fit for evaluating ADP-MA’s ground-agent code generation capabilities.

DSEval [16] benchmarks LLM-powered data science agents across the entire data science lifecycle. It contains 825 problems organized into 294 problem sets drawn from four sources: textbook exercises

(187 problems), StackOverflow questions (202), LeetCode challenges (40), and Kaggle competitions (396). The Kaggle subset is particularly relevant for pipeline-oriented evaluation, as its tasks require multi-step data manipulation and analysis. DSEval measures problem solving rate, difficulty-weighted score, and agent efficiency (steps and tokens per successful solution), and provides a browser-based diagnostic tool for fine-grained performance analysis.

AgentBench [10], presented at ICLR 2024, evaluates LLMs as interactive agents across eight environments including database interaction and operating-system commands. Its database environment, which tests SQL query generation and data retrieval, is most directly relevant to data processing agents.

MultiAgentBench [2] evaluates multi-agent collaboration across coordination topologies (star, chain, tree, graph) that map onto ADP-MA’s architecture—e.g., star mirrors Orchestrator-to-agent delegation, tree reflects the meta-agent–ground-agent hierarchy. Its milestone-based KPIs are relevant for assessing orchestration efficiency.

MLGym [12] provides a Gym environment for 13 machine learning research tasks requiring hypothesis generation, data processing, and iterative experimentation. Evaluation of frontier models shows that current systems can improve on baselines through hyperparameter optimization but struggle to generate novel hypotheses.

Despite the breadth of these benchmarks, a notable gap persists in evaluations that assess *adaptive* capabilities such as handling data drift mid-pipeline, recovering from cascading failures, and optimizing resource usage across heterogeneous workloads. ADP-MA’s built-in monitoring, progressive sampling, and cross-run learning mechanisms target precisely these under-evaluated dimensions.

2.4 Self-Evolving and Adaptive Agent Systems

Recent work on self-evolving agents [4] explores systems that can improve their capabilities over time through experience and reflection. This emerging direction bridges static foundation models with the continuous adaptability required by lifelong agentic systems. The survey introduces a unified conceptual framework highlighting four key components: System Inputs, Agent System, Environment, and Optimizers, which together form the feedback loop underlying self-evolving designs.

Deep research agents [6] represent another relevant direction, tackling complex multi-turn informational tasks through dynamic reasoning, adaptive planning, multi-hop retrieval, and iterative tool use. These systems demonstrate that autonomous agents can successfully coordinate sophisticated workflows involving information acquisition, synthesis, and report generation.

The concept of meta-learning in agent systems relates closely to ADP-MA’s architecture. While meta-learning traditionally refers to learning how to learn, ADP-MA’s meta-agents reason about how to orchestrate other agents. This distinction is crucial: rather than optimizing a single agent’s performance through experience, ADP-MA optimizes the composition and coordination of multiple specialized agents through strategic planning at runtime.

2.5 Reinforcement Learning for Agent Reasoning

Advances in reinforcement learning for reasoning [15] have demonstrated that agents can learn to make better decisions through interaction with appropriate reward signals. RL has emerged as a foundational methodology for transforming LLMs into large reasoning models capable of complex logical tasks. However, applying RL to data processing presents unique challenges due to sparse rewards, long-horizon tasks, and diverse success criteria across different data types and objectives.

ADP-MA complements RL-based methods by providing explicit reasoning mechanisms through meta-agents while maintaining flexibility for future integration of learning-based optimization. The hierarchical structure enables targeted credit assignment—the Monitor can attribute success or failure to specific ground-level agents, potentially informing future RL-based improvements to agent selection and configuration.

In summary, existing systems exhibit several limitations that ADP-MA addresses: most lack true hierarchical reasoning, few can dynamically create agents based on runtime observations, cross-phase coordination remains limited, and mechanisms for managing context windows effectively are underdeveloped. ADP-MA addresses these gaps through explicit hierarchical planning with the Orchestrator, dynamic agent instantiation via the Architect, continuous monitoring through the Monitor, and context management strategies that partition work to avoid context degradation.

3 System Overview

ADP-MA employs a hierarchical architecture comprising three distinct layers: meta-agents for high-level orchestration, ground-level agents for task execution, and supporting infrastructure for tool management and monitoring. This section provides a comprehensive overview of the system design, architectural components, and operational workflow.

3.1 Architectural Design

The architecture, illustrated in Figure 1, is built around three primary meta-agents that coordinate to construct, execute, and refine data processing pipelines: the Orchestrator, the Architect, and the Monitor. Each meta-agent is a distinct autonomous agent with its own LLM context and prompt; they communicate through shared state and are invoked serially by the pipeline runner. These meta-agents operate at a strategic level, making decisions about workflow design and system-level adaptations rather than directly manipulating data.

The **Orchestrator** serves as the primary interface between users and the system. It analyzes task specifications, decomposes complex goals into structured phases, and maintains the overall execution context. The Orchestrator implements ADP-MA’s hierarchical planning approach, breaking down large tasks into manageable subproblems of appropriate size and logical coherence. This decomposition strategy serves dual purposes: optimizing context window utilization and ensuring focused problem-solving without context degradation.

System Design - Architecture

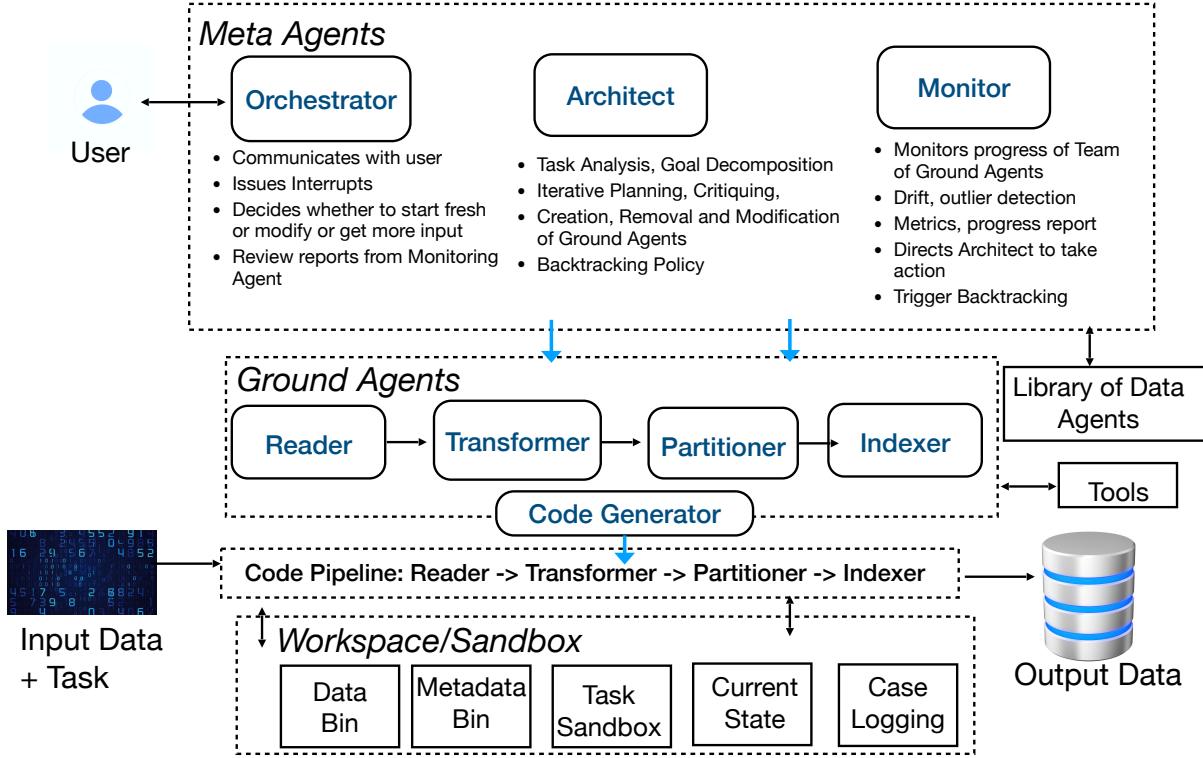


Figure 1: Architecture for ADP-MA showing the three meta-agents (Orchestrator, Architect, Monitor), ground-level agent ecosystem, and supporting infrastructure including the workspace/sandbox and tool library.

The **Architect** (Figure 2) translates high-level plans into concrete implementations by creating, modifying, and removing ground-level agents. The Architect maintains a library of previously designed agent definitions, enabling rapid composition of new pipelines from reusable components. When novel capabilities are required, the Architect synthesizes new agent definitions, specifies their tools and interfaces, and integrates them into the workflow. The Architect also implements backtracking policies, determining when ground-level failures necessitate higher-level replanning versus local remediation.

The **Monitor** provides continuous oversight of pipeline execution. It tracks progress across all active ground-level agents, detects anomalies such as data drift or outliers, generates metrics and progress reports, and issues interrupts when intervention is required. The Monitor employs rule-based heuristic checks to identify issues that might compromise pipeline correctness or efficiency. Based on monitoring insights, it directs the Architect to take corrective actions or communicates with users when human input is necessary.

3.2 Ground-Level Agent Ecosystem

Ground-level agents execute specialized data operations within sandboxed environments. ADP-MA's current implementation includes several agent types: Reader agents for data ingestion across diverse formats, Indexer agents for creating searchable representations, Partitioner agents for workload distribution, Graph agents for relational analysis, Transformer agents for data manipulation, and Compressor agents for storage optimization. This set of agent types is illustrative rather than exhaustive; the architecture supports extension to additional specialized agents as task requirements evolve.

Each ground-level agent operates within a process-level sandbox that isolates code execution through namespace confinement and I/O redirection. The sandbox prepares a restricted execution namespace containing only pre-imported libraries (pandas, datetime) and input data, executes LLM-generated code via Python's exec within this namespace, captures stdout, and tracks peak memory usage via tracemalloc. Input DataFrames are copied before injection to prevent mutation of upstream state. The sandbox auto-detects and invokes functions matching the stage_* naming convention, returning structured results that include success status, output

DataFrame, execution time, memory usage, and any error tracebacks. Ground-level agents receive instructions from the Architect, execute their assigned operations within these sandboxes, and report results to the Monitor.

3.3 Knowledge and Tool Management

The system maintains several persistent repositories that enable efficient pipeline construction and execution. The Library of Data Agents stores validated agent definitions that have been successfully used in prior tasks. When the Architect encounters similar requirements, it retrieves and adapts these definitions rather than creating agents from scratch, significantly accelerating development.

The Tools library contains verified implementations of common data operations, ranging from basic transformations to specialized analytical functions. Ground-level agents can invoke these tools, reducing the burden of code generation and minimizing errors. The current implementation intentionally limits tool availability to maintain safety and predictability, though the architecture supports dynamic tool expansion.

The system also maintains comprehensive metadata about data characteristics, pipeline state, and execution history. This metadata enables the Monitor to perform drift detection, facilitates context-aware planning by the Orchestrator, and supports intelligent backtracking decisions by the Architect.

3.4 Operational Workflow

Pipeline construction and execution follow an iterative process, summarized in Algorithm 1. Upon receiving a data processing task, the Orchestrator performs initial analysis to understand data characteristics and user objectives. It then generates a high-level plan structured as a sequence of phases, each addressing a distinct aspect of the overall task.

For each phase, the Orchestrator invokes the Architect to design ground-level agents capable of executing required operations. The Architect consults the agent library for reusable components and synthesizes new agents when necessary. Ground-level agents are instantiated within sandboxes and provided with appropriate tools and data access.

As execution proceeds, the Monitor continuously evaluates progress and performance. When issues arise, it classifies them as either local (addressable by the current agent) or global (requiring higher-level replanning). Local issues trigger focused remediation by the Architect, such as modifying agent parameters or providing additional tools. Global issues may result in phase restructuring or complete workflow revision by the Orchestrator.

Context management is critical for maintaining agent effectiveness over long-running pipelines. Large contexts in monolithic agentic applications lead to performance degradation, making it essential to divide workload and agentic scopes into manageable chunks. In ADP-MA, one remediation action involves splitting an agent's work into two serial agents with more focused responsibilities. Ongoing work aims to predict optimal context boundaries without wasting computational resources.

Algorithm 1 ADP-MA Operational Workflow

Require: Task specification T , Input data D
Ensure: Processed output O

- 1: // Stage 1: Data Understanding
- 2: $S \leftarrow$ Orchestrator profiles D (schema, statistics, summary)
- 3: // Stage 2: High-Level Planning
- 4: $P = \{p_1, \dots, p_n\} \leftarrow$ Architect plans phases from T and S
- 5: // Stage 3: Critique and Refinement
- 6: **repeat**
- 7: $C \leftarrow$ Architect critiques P (Level-1)
- 8: **if** $C.\text{severity} \geq \text{MAJOR}$ **then**
- 9: $P \leftarrow$ Architect revises plan using C
- 10: **end if**
- 11: **until** $C.\text{severity} \leq \text{MINOR}$ **or** max iterations
- 12: // Stage 4: Phase Expansion
- 13: **for** each phase $p_i \in P$ **do**
- 14: $E_i \leftarrow$ Architect expands p_i into substeps with agent types
- 15: Critique E_i (Level-2); revise if needed
- 16: **end for**
- 17: // Stage 5: Ground-Agent Execution
- 18: **for** each substep $e \in E_1 \cup \dots \cup E_n$ (per strategy) **do**
- 19: **for** $\ell \in \{\text{XS}, \text{S}, \text{M}, \text{FULL}\}$ **do**
- 20: Generate code via coding LLM; execute on ℓ -sample of D
- 21: **if** execution fails **then**
- 22: Refine code with error context; retry (up to budget)
- 23: **end if**
- 24: **end for**
- 25: Monitor evaluates output; issue verdict
- 26: **end for**
- 27: // Stage 6: Finalization
- 28: Assemble pipeline; write case documentation and event log
- 29: **return** Final processed output O

4 System Implementation

This section describes the concrete implementation of ADP-MA, covering the six-stage pipeline, the two-level critique mechanism, execution strategies, progressive sampling, runtime monitoring, schema contracts, and case documentation.

4.1 Meta-agent Pipeline Construction

The end-to-end pipeline is realized following sequential stages, each orchestrated by the ADKLLMPipelineRunner:

- (1) **Data Understanding.** The system begins by ingesting the input data source(s) and applying a suite of *data tools*: file-format detection, schema inference, and descriptive statistics computation. An LLM summarizes the findings in natural language, producing a *data summary* that becomes part of the shared context passed to all subsequent stages. When multiple data sources are provided (e.g., for join tasks), each is profiled independently and described in the summary.
- (2) **High-Level Planning.** The planning LLM receives the data summary together with a *task context* object that encapsulates the user goal, a detailed task description, the expected

output schema (including exact column names when specified), evaluation criteria, and arbitrary task-specific parameters. The LLM produces a minimal plan consisting of one to three ordered *phases*, each with a stated objective and rationale. An anti-scope-creep prompt directive instructs the LLM to include only phases directly required by the goal; extraneous phases are flagged as MAJOR or CRITICAL during critique.

- (3) **Critique and Refinement.** The plan enters a *Level-1 critique loop* (see Figure 2) in which a separate LLM call evaluates the plan against a rubric covering phase ordering, dependency correctness, goal coverage, and scope appropriateness. Each critique iteration returns a structured result with an overall severity drawn from the set {NONE, MINOR, MAJOR, CRITICAL} a list of issues with per-issue severity and affected component identifiers, and concrete suggestions. The loop exits when (a) severity drops to MINOR or NONE, (b) the maximum iteration count (default 10) is reached, or (c) an optional diff-based convergence detector determines that successive refinements have fallen below a configurable delta threshold. A MAJOR or CRITICAL verdict triggers automatic re-planning before the next critique iteration. An optional *dual-judge* mode submits the plan to two independent LLM calls and merges the resulting critiques for improved reliability.
- (4) **Phase Expansion.** Each approved phase is expanded by the Architect LLM into one to three *substeps*. Every substep specifies a ground-agent type (e.g., Reader, Profiler, Transformer, Validator, Joiner), an objective, implementation hints, an input/output *schema contract*, and optional dependency pointers to other substeps. A *Level-2 critique loop* similar to Level-1 reviews each expansion for completeness, agent-type validity, and input–output compatibility.
- (5) **Ground-Agent Execution.** The expanded substeps are dispatched to ground-level agents according to one of three *generation strategies* (described in Section 4.3). For each substep the ground-agent runner performs code generation via the coding LLM, sandbox execution with progressive sampling, and iterative refinement on failure. Section 4.4 details the progressive sampling protocol.
- (6) **Finalization.** Upon completion of all ground agents, the system assembles a standalone pipeline.py file containing the generated functions composed in execution order, writes a monitor summary, and emits a structured event stream for downstream replay or visualization.

The framework supports separate LLM configurations for planning (meta-agents) and coding (ground agents), enabling cost-quality trade-offs—for instance, a larger model for planning and a smaller, faster model for code generation. Supported LLM providers include Anthropic, Google, OpenAI, DeepSeek, and Ollama, making the system agnostic to the underlying foundation model.

4.2 Two-Level Critique Mechanism

The critique mechanism is the primary quality gate before any code is generated. It operates at two granularities:

Level 1 – Plan Critique. Evaluates the ordered sequence of phases against the task context. Key checks include: (i) phase ordering

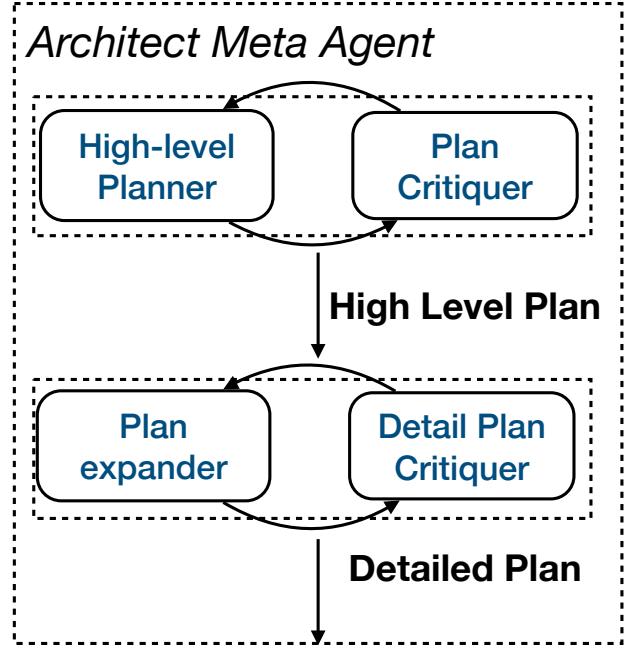


Figure 2: The Architect meta-agent workflow. The Architect receives task context and data summary from the Orchestrator, generates a high-level plan, refines it through a two-level critique loop (Level 1 for plan structure, Level 2 for phase expansion), and produces schema-contracted substep definitions that are dispatched to ground-level agents for execution.

respects data dependencies; (ii) every aspect of the user goal is addressed by at least one phase; (iii) no phase introduces work outside the stated scope; and (iv) the number of phases is minimal. An excessive phase count is flagged as MAJOR; out-of-scope phases as CRITICAL.

Level 2 – Expansion Critique. Evaluates each phase’s substep decomposition. Checks verify that the selected agent types are appropriate for the substep objective, that input columns required by each substep are produced by a preceding substep or present in the original data, and that the substep count remains within the one-to-three budget per phase.

Both levels share the same exit logic, parameterized by CritiqueLoopConfig: maximum iterations, exit severity threshold, optional convergence exit, and optional requirement that all phases be expanded before exit (Level 1 only). The structured output format—with per-issue issue_id, severity, affected_component, and suggestion—enables deterministic routing of critique feedback back into the planner prompt for targeted revision.

4.3 Generation Strategies

The dispatch of ground agents to execution follows one of three strategies, selectable at runtime:

- **Centralized.** Substeps execute serially in plan order. Each agent receives the *full accumulated context*: the current DataFrame

(reflecting all prior agents’ outputs) and an updated input schema. This mode maximizes inter-agent information flow at the cost of sequential latency. It is best suited for small pipelines with tight data dependencies.

- **Autonomous.** Substeps are grouped into *parallel batches* by dependency depth. Agents within the same batch execute concurrently via `asyncio.gather`, governed by a semaphore with a configurable maximum (default 4). Each agent receives only the original input schema (contract-only context), ensuring independence. After a batch completes, outputs are merged before the next batch begins. This mode is suited for large pipelines with independent stages.
- **Hybrid.** Phases execute sequentially, preserving inter-phase data flow, while agents *within* a phase execute in parallel when they have no intra-phase dependencies. Interface validation is performed between phases to check that the output columns of one phase satisfy the input requirements of the next. This mode balances the full-context benefits of centralized execution with the throughput of autonomous parallelism.

4.4 Progressive Sampling

To reduce cost and surface errors early, each ground agent is tested through an escalating sequence of sample sizes before full-scale execution:

Table 1: Progressive sampling levels.

Level	Rows	Purpose
XS	10	Syntax and basic logic validation
S	100	Functional correctness
M	1000	Performance and edge-case coverage
FULL	All	Production execution

Execution begins at the configured starting level (default XS). On success at a given level, the agent promotes to the next level; on failure, the coding LLM receives the error traceback and the failing code, generates a revised implementation, and retries at the same level. A maximum refinement budget (default 3 revisions) caps retry cost. Each revision is recorded with its trigger (`initial`, `error_fix`, or `scale_up`), execution time, peak memory, and output row count. This progressive protocol catches the majority of errors on cheap, small samples while ensuring that scaling behavior is validated before the full dataset is processed.

4.5 Backtracking

When a ground agent exhausts its refinement budget without producing a correct result, the failure holds up the remainder of the pipeline. Rather than aborting, ADP-MA employs a backtracking mechanism that unwinds execution to a prior state and attempts recovery at a higher level of abstraction. Two levels of backtracking are supported:

- **Phase-level backtracking.** The system discards the outputs of the failing substep and reverts to the DataFrame produced by the previous successfully completed stage. The Architect is then re-invoked to re-expand the current phase, potentially

choosing different agent types, revised schema contracts, or alternative implementation hints. Execution resumes from the re-expanded substeps.

- **Plan-level backtracking.** If phase-level backtracking also fails—or if the Monitor determines that the failure reflects a fundamental flaw in the plan rather than an implementation error—the system backtracks all the way to the planning stage. The Architect receives the accumulated error evidence (failing agent types, error classes, and the substep objectives that could not be fulfilled) and generates a revised high-level plan, which then proceeds through the full critique, expansion, and execution cycle.

The current implementation uses heuristic triggers to choose between these two levels: a single substep failure initiates phase-level backtracking, while repeated phase-level failures (default threshold: 2 per phase) or a Monitor PAUSE verdict escalate to plan-level backtracking. Per-phase and global retry caps (default 2 per phase, 3 total) prevent unbounded backtracking loops.

While effective in practice, this heuristic approach has limitations. The decision of *when* to backtrack and *how far* to unwind is based on fixed thresholds rather than on a principled cost model that weighs the expense of further local refinement against the cost of replanning. Developing a more principled backtracking policy—potentially informed by learned estimates of repair probability, accumulated revision cost, and plan-level error attribution—is an active direction of future work.

4.6 Code Execution Sandbox

Each ground agent’s generated code is executed within a process-level sandbox that provides namespace isolation, I/O capture, and resource tracking. The sandbox operates as follows:

- (1) **Namespace preparation.** A restricted Python namespace is constructed containing only pre-imported libraries (`pandas` as `pd`, `datetime`) and the input DataFrame. When additional data sources are specified, they are injected as `df_{name}` variables. All input DataFrames are copied before injection to prevent mutation of upstream state.
- (2) **Execution.** The generated code is executed via Python’s `exec` within the prepared namespace. The sandbox redirects `stdout` to capture print output and uses `tracemalloc` to measure peak memory consumption during execution.
- (3) **Function invocation.** After executing the code to define functions, the sandbox scans the namespace for callables matching the `stage_*` naming convention and invokes the first match, passing a fresh copy of the input DataFrame.
- (4) **Result capture.** The sandbox returns a structured result containing: success/failure status, the output DataFrame (if any), execution time in milliseconds, peak memory in megabytes, captured `stdout`, output row count, and—on failure—the error message and full traceback.

This design provides functional isolation sufficient for the system’s threat model (LLM-generated data-processing code operating on user-supplied datasets) without the overhead of container-based sandboxing. The namespace is confined to pre-imported libraries and does not restrict Python’s `__builtins__`, so generated code

Table 2: Default monitoring thresholds.

Check	Warning	Critical
Revision count	≥ 2	≥ 4
Row drop (%)	$\geq 30\%$	$\geq 90\%$
Row growth (%)	$\geq 500\%$	—
Null-rate increase	$\geq 20 \text{ pp}$	—
Agent wall-clock time	$\geq 60 \text{ s}$	$\geq 300 \text{ s}$
Cost vs. budget	$\geq 80\%$	$\geq 100\%$
Peak memory	$\geq 500 \text{ MB}$	—

retains access to standard library imports. The primary safety mechanisms are the copy-on-read input policy, stdout redirection, memory tracking, and the timeout enforced by the progressive sampling loop.

4.7 Pipeline Monitoring

The `PipelineMonitor` evaluates the system state after every ground-agent execution using *rule-based heuristics*—critically, without incurring additional LLM calls. The monitor maintains configurable thresholds (Table 2) and issues one of five verdicts: `CONTINUE`, `WARN`, `PAUSE`, `ABORT`, or `RETRY`.

Four additional monitoring features operate across and within runs:

Data-drift detection. After each agent, the monitor compares the output DataFrame to the input along three axes: row count change, column set change, and per-column null-rate increase. Thresholds are configurable; a row drop exceeding 90% triggers an automatic `PAUSE`.

Adaptive sampling. When the monitor observes a pattern of agents passing at the XS level but failing at S, it raises an alert indicating that the smallest sample is unrepresentative and suggests starting future agents at the S level.

Cross-run learning. On initialization, the monitor optionally loads failure patterns from prior case folders (up to a configurable maximum of 10 cases). Failure patterns—agent type, error class, input characteristics—are matched against the current run, and preemptive warnings are issued when a known-problematic configuration is detected.

Cost tracking. The monitor accumulates token usage and estimated USD cost across all LLM calls and warns when spending approaches a configurable budget ceiling (default \$1.00).

4.8 Schema Contracts

Each ground agent is governed by an `EnhancedSchemaContract` that specifies: (i) required input columns with type constraints (`dtype`, `nullability`, pandas API checks); (ii) columns to add, preserve, or remove; (iii) value constraints (range, enum, regex pattern, uniqueness, non-null); (iv) a row-count constraint relative to the input (`SAME`, `LESS`, `GREATER`, or `ANY`); and (v) free-text postconditions and invariants. Contracts flow through the Architect’s expansion prompts and into the ground-agent code-generation prompt, ensuring that the generated code respects both upstream and downstream schema expectations. At execution time, a `ContractVerificationResult` records violations with severity, affected column,

expected vs. actual value, and (where applicable) offending row indices.

When the user supplies an `expected_output` specification with exact column names, the final ground agent receives strict instructions to produce exactly those columns, and the output is validated against the requirement schema before finalization.

4.9 Case Documentation and Audit Trail

Every pipeline run produces a self-contained `case folder` that serves as a complete audit trail:

- `case_document.json` — a structured record of all meta-agent decisions: data understanding summary, plan versions across critique iterations, phase expansions, and monitoring verdicts.
- `monitor_summary.json` — the full list of alerts raised, with severity, evidence, and recommended actions.
- `events.jsonl` — a time-stamped event stream (stage starts and completions, agent progress, critique results, errors) suitable for UI replay.
- `ground_agents/` — the generated Python source for each ground agent, including all revision history.
- `pipeline.py` — the assembled, standalone executable pipeline composed from the successful agent functions.

This comprehensive logging ensures reproducibility: any pipeline run can be inspected post-hoc to understand why specific planning or execution decisions were made, facilitating debugging, compliance auditing, and iterative improvement of the framework itself.

4.10 Human-in-the-Loop Checkpoint

An optional *human checkpoint* can be enabled at the boundary between planning and execution. When active, the system pauses after the critique loop converges and presents the finalized plan to the user for approval. The user may approve, modify, or reject the plan. A rejection triggers re-planning (up to two rejections), after which the system proceeds with the best available plan. This mechanism preserves the option for human oversight in high-stakes scenarios while remaining fully automatic when the checkpoint is disabled.

4.11 System Tuning and Agent Prompt Evolution

The design of ADP-MA’s meta-agent prompts, critique rubrics, and monitoring thresholds evolved through a combination of analytical design and empirical feedback gathered from running the system on a variety of common data processing tasks. The initial prompt templates and pipeline structure were designed from first principles—e.g., the two-level critique hierarchy, progressive sampling levels, and scope-guard directives emerged from analysis of failure modes in early prototypes. However, many design decisions required iterative refinement informed by observing system behavior on concrete workloads.

To assemble a representative development workload, we used commercial LLM queries (e.g., “list a variety of data processing tasks covering cleaning, aggregation, feature engineering, multi-source integration, and graph analytics”) to generate a diverse set

of task specifications. Synthetic input datasets for these tasks were generated similarly, using LLM-assisted scripts to produce tabular data with controlled characteristics such as missing values, mixed types, duplicates, and temporal structure. This approach ensured broad coverage of common data processing patterns without relying on proprietary or domain-specific datasets.

The resulting task suite (listed in Appendix A.7) spans five categories: tabular cleaning, aggregation and analytics, feature engineering, multi-source integration, and temporal graph processing. Running these tasks end-to-end exposed recurring failure patterns—such as scope creep in planning, fragile column-name assumptions in generated code, and insufficient error context in refinement prompts—that drove targeted revisions to the prompt templates, critique rubrics, and monitoring rules. For example, the anti-scope-creep directive in the planning prompt and the phase-count budget in Level-1 critique were both added after observing that unconstrained planners routinely generated unnecessary validation or profiling phases. Similarly, the requirement to pass full error tracebacks (rather than just error messages) to the refinement LLM was motivated by observing that terse error summaries led to ineffective code fixes.

This iterative tuning process was manual rather than automated: each round involved inspecting case documentation, identifying systematic weaknesses, revising the relevant prompt or threshold, and re-running the affected tasks to confirm improvement. While this approach does not guarantee global optimality, it produced a stable configuration that achieves high pass rates across the development workload (Appendix A.7).

4.12 Evaluation Plan

Beyond the internal benchmark suite used for development tuning (Appendix A.7), we plan to evaluate ADP-MA against established external benchmarks to measure its effectiveness relative to existing systems. We are currently in the process of evaluating ADP-MA on KRAMABENCH [11], a benchmark of 104 knowledge-intensive, multi-step data science tasks spanning six scientific and legal domains. KRAMABENCH is particularly well suited for assessing ADP-MA because its tasks require end-to-end pipeline orchestration—data discovery, integration, wrangling, and statistical reasoning—which exercises the full meta-agent hierarchy rather than isolated code generation. Published baselines for KRAMABENCH include smolagents (50% end-to-end accuracy), DS-GURU with self-correction (22%), and GPT-o3 design-only (42%), providing established reference points for comparison.

The evaluation adapter translates each KRAMABENCH task into an ADP-MA pipeline run by mapping the task specification to a user goal and task context, loading the associated data files as primary and additional data sources, and comparing the pipeline output against the ground-truth answer. We intend to report end-to-end accuracy, per-domain breakdowns, and diagnostic metrics including planning iterations, agent revision counts, and monitor interventions. Additional external benchmarks such as DA-Code [7] and DSEval [16] are planned as follow-on evaluations.

5 Conclusion

We have presented ADP-MA, a meta-agent framework for autonomous data processing that advances the state of practice in several key dimensions. Through hierarchical orchestration, ADP-MA achieves greater adaptability and autonomy than existing approaches that rely on static pipelines or predefined agent templates. The separation between strategic meta-agents and execution-oriented ground-level agents enables sophisticated reasoning about workflow design while maintaining efficiency in task execution.

ADP-MA addresses critical limitations in current data processing automation, including inflexibility to data drift, limited cross-phase optimization, and inadequate error recovery mechanisms. By incorporating continuous monitoring, iterative refinement, and intelligent backtracking, the system can handle the uncertainties and complexities inherent in real-world data processing tasks.

Future work will focus on several promising directions. Expanding the framework’s capabilities to handle diverse data modalities beyond tabular data represents an important next step. Incorporating reinforcement learning to optimize meta-agent decision-making based on accumulated experience could further enhance performance. Developing further benchmarks to cover a representative variety of real-world data science, mining, and processing tasks will enable more rigorous assessment of autonomous data processing systems. Finally, exploring mechanisms for meta-agents to synthesize new tools and agent types autonomously would move the framework closer to fully general-purpose data processing automation.

The meta-agent paradigm offers a pathway toward data processing systems that can operate with increasing autonomy while remaining interpretable and controllable. As LLM capabilities continue to advance, frameworks like ADP-MA will play a crucial role in translating these capabilities into practical tools that democratize sophisticated data analysis and engineering.

References

- [1] Anthropic. 2025. Building Effective Agents: Context Engineering. <https://www.anthropic.com/engineering/effective-context-engineering-for-ai-agents>. Accessed: 2025.
- [2] Various Authors. 2025. MultiAgentBench: Evaluating the Collaboration and Competition of LLM Agents. *arXiv preprint arXiv:2503.01935* (2025).
- [3] Zui Chen et al. 2025. A Survey on Data Science with Large Language Models. *arXiv preprint arXiv:2505.03418* (2025).
- [4] Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao Li, Zhaochun Ren, Nikos Aletras, Xi Wang, Han Zhou, and Zaiqiao Meng. 2025. A Comprehensive Survey of Self-Evolving AI Agents: A New Paradigm Bridging Foundation Models and Lifelong Agentic Systems. *arXiv preprint arXiv:2508.07407* (2025).
- [5] Google. 2025. Google Agent Development Kit (ADK). <https://google.github.io/adk-docs/>. Accessed: 2026.
- [6] Yuxuan Huang, Yihang Chen, Haozheng Zhang, Kang Li, Huichi Zhou, Meng Fang, Linyi Yang, Xiaoguang Li, Lifeng Shang, Songcen Xu, Jianye Hao, Kun Shao, and Jun Wang. 2025. Deep Research Agents: A Systematic Examination and Roadmap. *arXiv preprint arXiv:2506.18096* (2025).
- [7] Yue Huang, Qihui Luo, Jiawen Li, et al. 2024. DA-Code: Agent Data Science Code Generation Benchmark. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*.
- [8] Lan Li, Liri Fang, Bertram Ludäscher, and Vete I. Torvik. 2024. AutoDCWorkflow: LLM-based Data Cleaning Workflow Auto-Generation and Benchmark. *arXiv preprint arXiv:2412.06724* (2024).
- [9] Ziming Li, Qianbo Zang, David Ma, Jiawei Guo, Tuney Zheng, Minghao Liu, Xinyao Niu, Yue Wang, Jian Yang, Jiaheng Liu, Wanjun Zhong, Wangchunshu Zhou, Wenhao Huang, and Ge Zhang. 2024. AutoKaggle: A Multi-Agent Framework for Autonomous Data Science Competitions. *arXiv preprint arXiv:2410.20424* (2024).

- [10] Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Zhang, and Jie Tang. 2024. AgentBench: Evaluating LLMs as Agents. In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*.
- [11] Bertram Ludäscher, Lan Li, Liri Fang, Yilun Long, Udayan Khurana, and Juliana Freire. 2025. KRAMABENCH: A Benchmark for Knowledge-Intensive, Real-World, Multi-Step Data Science Pipelines. *arXiv preprint arXiv:2506.06541* (2025).
- [12] Deepak Nathani, Lovish Madaan, Nicholas Roberts, Nikolay Bashlykov, Ajay Menon, Vincen Moens, Amar Budhiraja, Despoina Magka, Vladislav Vorotilov, Gaurav Chaurasia, Dieuwke Hupkes, Ricardo Silveira Cabral, Tatiana Shavrina, Jakob Foerster, Yoram Bachrach, William Yang Wang, and Roberta Raileanu. 2025. MLGym: A New Framework and Benchmark for Advancing AI Research Agents. *arXiv preprint arXiv:2502.14499* (2025).
- [13] Felix Neutatz, Binger Chen, Ziawasch Abedjan, and Eugene Wu. 2022. Data Cleaning and AutoML: Would an Optimizer Choose to Clean? *Datenbank-Spektrum* 22, 2 (2022), 121–130.
- [14] Xingyu Shen, Shengding Hu, Xinrong Zhang, Xu Han, Xiaojun Meng, Jiansheng Wei, Zhiyuan Liu, and Maosong Sun. 2025. AutoClean: LLMs Can Prepare Their Training Corpus. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (System Demonstrations)*. Association for Computational Linguistics, 85–95.
- [15] Kaiyan Zhang, Yuxin Zuo, Bingxiang He, Youbang Sun, Runze Liu, Che Jiang, Yuchen Fan, Kai Tian, Guoli Jia, Pengfei Li, Yu Fu, Xingtai Lv, Yuchen Zhang, Sihang Zeng, Shang Qiu, Haozhan Li, Shijie Wang, Yuru Wang, Xinwei Long, Fangfu Liu, et al. 2025. A Survey of Reinforcement Learning for Large Reasoning Models. *arXiv preprint arXiv:2509.08827* (2025).
- [16] Yuge Zhang, Qiyang Jiang, Xingyu Han, Nan Chen, Yuqing Yang, and Kan Ren. 2024. Benchmarking Data Science Agents. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 5636–5652.

A Appendix

A.1 Ground-Agent Types

Table 3 lists the ground-agent types currently supported by the Architect during phase expansion. New types can be added by registering an agent definition in the schema catalog; no changes to the orchestration logic are required.

Table 3: A sample of supported ground-agent types.

Agent Type	Responsibility
Reader	Data ingestion across diverse file formats
Profiler	Schema inference, statistics, data quality assessment
Transformer	Data manipulation, cleaning, reshaping
Validator	Output quality checks and constraint verification
Joiner	Multi-source merges and schema alignment
Indexer	Searchable representation construction
Partitioner	Workload distribution and data splitting
Graph	Relational and graph-structured analysis
Compressor	Storage optimization and encoding
Aggregator	GroupBy, rolling statistics, pivots
FeatureEngineer	Encoding, scaling, datetime feature extraction

A.2 Critique Loop Configuration

The critique loop is parameterized by `CritiqueLoopConfig`, whose fields are listed in Table 4. All parameters have sensible defaults and can be overridden per-run.

Table 4: Critique loop configuration parameters.

Parameter	Default	Description
<code>max_iterations</code>	10	Max critique-refine cycles
<code>exit_on_severity</code>	MINOR	Exit when severity \leq threshold
<code>require_all_phases_expanded</code>	false	L1: wait for all expansions
<code>enable_convergence_exit</code>	false	Diff-based early stopping
<code>convergence_threshold</code>	0.1	Delta below which converged
<code>convergence_min_iterations</code>	2	Min iters before convergence

A.3 Event Stream Schema

The event logger emits structured events of the following types, enabling both real-time streaming (via callbacks) and post-hoc replay:

- `stage_start` / `stage_complete` – marks entry and exit of each of the six pipeline stages.
- `agent_progress` – reports ground-agent status (generating, executing, refining, success, failed) with sample level, revision number, and timing.
- `critique_result` – records each critique iteration’s severity, issue count, and summary.
- `monitor_verdict` – captures the monitor’s verdict and associated alerts after each agent.
- `plan_version` – snapshots the plan after each refinement iteration.
- `error` – logs exceptions with traceback and context.

A.4 Task Context Passthrough

A rich `task_context` object flows through every meta-agent prompt, ensuring that downstream agents have access to the user’s full intent without relying on the accumulation of multi-turn conversational context. The task context comprises:

- `task_description`: A detailed natural-language description of the processing goal.
- `expected_output`: Schema requirements including exact column names and row-level descriptions.
- `evaluation_criteria`: Success criteria against which the output will be judged.
- `task_parameters`: Arbitrary key-value pairs for domain-specific configuration.

This design decouples planning quality from context-window management: the LLM receives a self-contained specification at each stage rather than a growing conversational history.

A.5 Multi-Source Data Support

When a task requires joining or merging data from multiple files, additional data sources are passed as named `DataFrames`. Each additional source is profiled during the Data Understanding stage and described in the data summary. During ground-agent execution, additional sources are injected into the sandbox namespace as `df_{name}` variables (see Section 4.6), enabling agents to perform real joins and merges without generating mock data. This mechanism

supports an arbitrary number of auxiliary data sources alongside the primary input.

A.6 TDD and Code Review Support

The framework optionally integrates test-driven development (TDD) into ground-agent execution. When enabled, the coding LLM first generates unit tests for each agent function before generating the implementation. Tests are executed in the sandbox; the implementation must pass all tests before proceeding to the next sample level. Three modes are supported: `disabled` (default), `optional` (tests generated but failures do not block), and `strict` (test failures block promotion).

Independently, an LLM-powered *code review* step can inspect generated code for correctness, error handling, performance, pandas best practices, and contract compliance. Review findings are fed back into the refinement prompt when issues are detected.

A.7 Development and Tuning Tasks

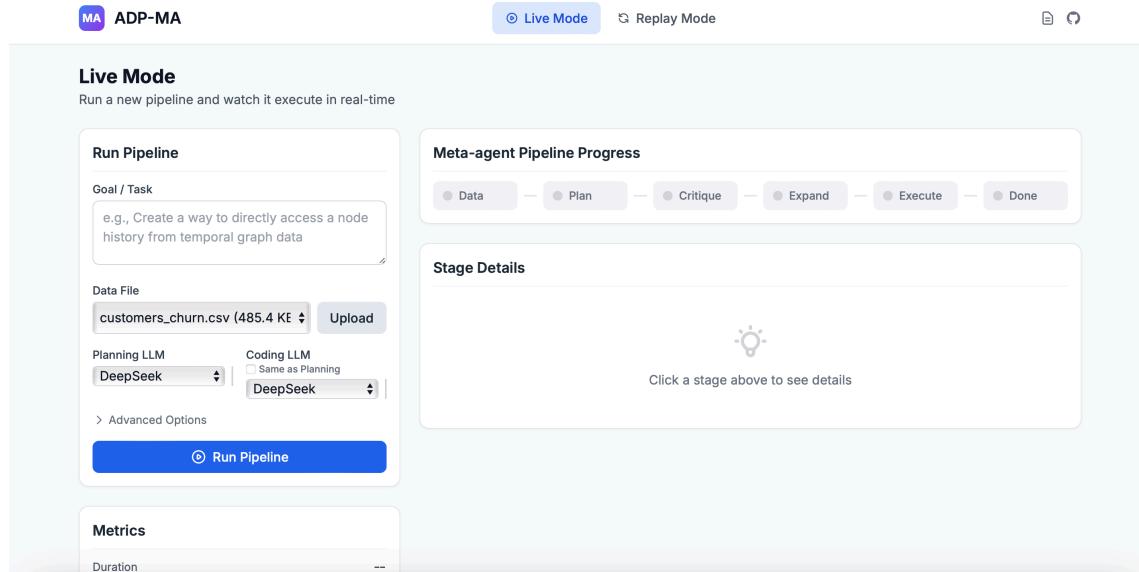
Table 5 lists the 25 data processing tasks used to iteratively tune prompts, critique rubrics, and monitoring thresholds as described in Section 4.11. Task specifications and synthetic datasets were generated using commercial LLM queries.

A.8 Demonstration Interface

We provide a web-based demonstration tool with two operating modes: *Live Mode* and *Replay Mode*. In Live Mode, users submit a natural-language goal, select the planning and coding LLMs from a set of available providers (e.g., OpenAI, Anthropic, Google, or local Ollama models), and observe the pipeline execute in real time via a streaming WebSocket connection. In Replay Mode, users browse previously completed cases and replay the recorded event stream at configurable playback speeds, enabling review and presentation of past runs. Both modes share a common visualization layout: a meta-agent pipeline progress bar, per-stage detail panels, a ground-agent pipeline view, generated code inspection, and execution metrics.

Table 5: Data processing tasks used for system tuning and prompt evolution.

ID	Task Name	Category	Difficulty
TC-001	Missing Value Imputation	Tabular Cleaning	Easy
TC-002	Duplicate Removal	Tabular Cleaning	Easy
TC-003	Type Standardization	Tabular Cleaning	Medium
TC-004	Outlier Detection (IQR)	Tabular Cleaning	Medium
TC-005	Multi-Format Date Parsing	Tabular Cleaning	Medium
TC-006	Text Normalization	Tabular Cleaning	Medium
AA-001	Simple GroupBy Aggregation	Aggregation & Analytics	Easy
AA-002	Multi-Level GroupBy	Aggregation & Analytics	Medium
AA-003	Rolling Statistics	Aggregation & Analytics	Medium
AA-004	Pivot Table Construction	Aggregation & Analytics	Medium
AA-005	Percentile Analysis	Aggregation & Analytics	Medium
FE-001	One-Hot Encoding	Feature Engineering	Easy
FE-002	Label Encoding	Feature Engineering	Easy
FE-003	Numeric Scaling (Min-Max, Z-Score)	Feature Engineering	Medium
FE-004	Datetime Feature Extraction	Feature Engineering	Medium
FE-005	Binning (Equal-Width, Quantile)	Feature Engineering	Medium
MI-001	Inner Join	Multi-Source Integration	Easy
MI-002	Left Join with Null Handling	Multi-Source Integration	Medium
MI-003	Multi-Table Join	Multi-Source Integration	Hard
MI-004	Schema Alignment & Merge	Multi-Source Integration	Medium
TG-001	Node History Index	Temporal Graph	Medium
TG-002	Edge Timeline Construction	Temporal Graph	Medium
TG-003	Graph Metrics Computation	Temporal Graph	Hard
TG-004	Temporal Subgraph Extraction	Temporal Graph	Medium
TG-005	Event Sequence Pattern Analysis	Temporal Graph	Hard

**Figure 3: Live Mode initial state.** The left panel provides a task input area and LLM selection dropdowns for the planning and coding models. The right panel shows the meta-agent pipeline progress bar and placeholders for stage details and generated code.

The screenshot shows the ADP-MA interface in Replay Mode. At the top, there are tabs for "Live Mode" and "Replay Mode" (which is selected). Below the tabs, the title "Replay Mode" is displayed with the subtitle "Replay previous pipeline executions to review or demonstrate".

Select Case to Replay: A list of 100 cases. One case is highlighted: "case_20260130_102653_bde4bd3a Has Events". The description says "Build chronological edge list for e..." and it was run on 1/30/2026 at 10:30 AM with 1 agents.

Meta-agent Pipeline Progress: A horizontal timeline showing the pipeline stages: Data, Plan, Critique, Expand, Execute, Done. The "Plan" stage is currently active.

Stage: Data Understanding:

- Data Analysis:**
 - Rows:** 5,000
 - Columns:** 4
 - Column types: node1 (object), node2 (object), event_type (object), timestamp (object)
- Summary:**

This dataset represents event logs for interactions between nodes, detailing updates or creations of connections with timestamps. The data is complete without null values and structured consistently across 5000 records, indicating high quality and reliability. Recommended processing includes converting the 'timestamp' column to a datetime format for temporal analysis, followed by aggregating events over time periods or analyzing connectivity patterns between nodes.

Figure 4: Replay Mode showing the Data Understanding stage. The stage panel displays detected row count, column count, column types, and a natural-language summary of the dataset characteristics.

The screenshot shows the ADP-MA interface in Replay Mode. The "Replay Mode" tab is selected. Below the tabs, the title "Replay previous pipeline executions to review or demonstrate" is displayed.

Select Case to Replay: A list of 100 cases. One case is highlighted: "case_20260130_193832_6b90287b Has Events". The description says "create a way to directly access th..." and it was run on 1/30/2026 at 07:38 PM with 7 agents.

Meta-agent Pipeline Progress: A horizontal timeline showing the pipeline stages: Data, Plan, Critique, Expand, Execute, Done. The "Plan" stage is currently active.

Stage: Planning:

Plan:

- Strategy:** Build a dictionary mapping each node to its chronological event history by processing the event log data.
- Phases:**
 - Data Preparation:** Convert timestamp to datetime, sort data chronologically, and validate required columns exist.
 - Build History Access:** Create a dictionary where each key is a node and the value is a sorted list of its events (event_type, timestamp).

Figure 5: Replay Mode showing the Plan stage. The detail panel displays the selected strategy (e.g., centralized) and the decomposed phases with their objectives, providing visibility into the Architect meta-agent's output.

The screenshot shows the ADP-MA interface in Replay Mode. At the top, there are tabs for 'Live Mode' and 'Replay Mode' (which is selected). Below the tabs, the title 'Replay Mode' is displayed, followed by the subtitle 'Replay previous pipeline executions to review or demonstrate'. On the left, a sidebar titled 'Select Case to Replay' lists four cases: 'case_20260130_193832_6b90287b' (Has Events), 'case_20260130_180602_c001f03e' (Has Events), 'case_20260130_180431_5b25697d' (Has Events), and 'case_20260130_180301_191c11e4' (Has Events). Each case entry includes a timestamp and the number of agents involved. To the right, the 'Meta-agent Pipeline Progress' is shown as a horizontal sequence of colored boxes: Data (green), Plan (green), Critique (blue, currently active), Expand (green), Execute (green), and Done (green). A 'Done' button is located at the end of the progress bar. Below the progress bar, the 'Stage: Critique' section is expanded, showing the 'Plan Critique' section with a 'Severity: MINOR' message. Under 'Issues', three items are listed with exclamation marks: 'The plan lacks explicit handling for both node1 and node2 columns when building node history, potentially missing events where a node appears in node2 but not node1.', 'No mention of handling duplicate timestamps or tie-breaking logic for events occurring at the same time.', and 'The plan does not specify how to handle events where a node interacts with itself (if node1 == node2).'. Under 'Recommendations', three items are listed with plus signs: 'Clarify that both node1 and node2 columns should be processed to capture all events for each node.', 'Add a step to resolve timestamp ties (e.g., by event_type order or adding a secondary sort key).', and 'Consider outputting the history in a structured format (e.g., list of dicts) for easier downstream use.'

Figure 6: Replay Mode showing the Critique stage. The panel displays the critique severity level (e.g., MINOR), a list of identified issues, and recommendations for plan improvement.

The screenshot shows the ADP-MA interface in Replay Mode. The layout is identical to Figure 6, with 'Live Mode' and 'Replay Mode' tabs at the top, and 'Replay Mode' selected. The 'Select Case to Replay' sidebar shows five cases: 'case_20260130_102653_bde4bd3a' (Has Events), 'case_20260130_102412_a1aee839' (Has Events), 'case_20260130_102145_f2e994f9' (Has Events), and 'case_20260130_101854_f697abd5' (Has Events). The 'Meta-agent Pipeline Progress' bar shows the stages: Data, Plan, Critique, Expand (blue, active), Execute, and Done. The 'Stage: Expansion' section is expanded, showing two phases: 'Timestamp Conversion and Sorting' and 'Edge List Construction'. The 'Timestamp Conversion and Sorting' phase contains two substeps: 'TypeConverter' (Convert the 'timestamp' column to datetime format) and 'NodeHistoryBuilder' (Sort events chronologically and ensure bidirectional edges are handled correctly). The 'Edge List Construction' phase contains one substep: 'ChronologicalEdgeListBuilder' (Construct a chronological edge list for each node pair, ensuring bidirectional edges are handled correctly).

Figure 7: Replay Mode showing the Expand stage. Each phase is expanded into concrete substeps with assigned agent types, representing the transition from abstract plan to executable agent assignments.

The screenshot shows the ADP-MA interface in 'Replay Mode'. At the top, there's a navigation bar with 'ADP-MA' and tabs for 'Live Mode' and 'Replay Mode'. Below the navigation is a section titled 'Replay Mode' with the sub-section 'Replay previous pipeline executions to review or demonstrate'. On the left, a panel titled 'Select Case to Replay' lists several cases with their details and status (e.g., 'Has Events'). On the right, a large panel titled 'Meta-agent Pipeline Progress' shows a sequence of stages: Data, Plan, Critique, Expand, Execute, and Done. The 'Execute' stage is highlighted. Below this, the 'Stage: Execution' section shows an 'Execution Summary' with three steps: 1.1: TypeConverter (success), 1.2: NodeHistoryBuilder (success), and 2.1: ChronologicalEdgeListBuilder (success). To the right, a 'Generated Code' section displays Python code for the TypeConverter step, including imports and logic for handling empty dataframes and required columns.

Figure 8: Replay Mode showing the Execute stage. During execution, the layout switches to a side-by-side view: the left panel summarizes agent status (e.g., 3 successful agents), while the right panel displays the generated Python code with revision history and syntax highlighting.

This screenshot shows the ADP-MA interface in 'Replay Mode' after a pipeline has completed. The interface is divided into several panels: 'Playback Controls' (Completed, Case duration: 3m 44s, Progress: 35 / 33 events, Replay button, playback speed slider from 1x to 1.9x), 'Generated Code' (a large dark panel showing the final Python code for the TypeConverter step), 'Metrics' (Duration: 224.0s, Agents: 3/3, LLM Calls: 5, LLM Time: 94.7s), and 'Ground-agent Pipeline' (a list of stages with sample levels: TypeConverter (XS → S → M → FULL), NodeHistoryBuilder (XS → S → M → FULL), and ChronologicalEdgeListBuilder (XS → S → M → FULL), all marked as completed with green checkmarks).

Figure 9: Replay Mode after pipeline completion. The interface shows playback controls with auto-suggested speeds, summary metrics (total duration, agent count, LLM calls), generated code, and the ground-agent pipeline depicting each agent's progression through sample levels (XS → S → M → FULL).