AI-Workflows: An Evaluation Framework for LLM-Powered Workflow Generation Strategies

Designing Large Scale AI Systems Project

Matteo Costalonga
matteo.costalonga@studenti.unitn.it
University of Trento
Trento, Trentino Alto-Adige, Italy

Alex Reichert
alex.reichert@studenti.unitn.it
University of Trento
Trento, Trentino Alto-Adige, Italy

ABSTRACT

Large Language Models (LLMs) have demonstrated remarkable capabilities in understanding and generating structured outputs, opening new possibilities for automated workflow generation. This paper presents AI-Workflows, an evaluation framework designed to systematically compare different strategies for generating executable workflows from natural language requests under controlled configurations. AI-Workflows implements and evaluates three distinct generation strategies—*monolithic*, *incremental* and *bottom-up*—each representing a different approach to decomposing the workflow generation task. The framework also supports multiple workflow representations (linear and structured DAGs), configurable LLM backends (Google Gemini, Cerebras) and orthogonal enhancement features including interactive chat clarification and iterative refinement. We introduce an evaluation protocol and a metrics suite capturing:

(i) Structural workflow similarity for consistency analysis
(ii) Execution trace similarity for functional robustness
(iii) Constraint-based correctness scoring
(iv) Intent resolution and reasoning coherence measurements

Our experiments reveal important trade-offs between generation strategies in terms of structural stability, functional correctness and sensitivity to task decomposition, with no single strategy consistently dominating across all evaluation dimensions.

KEYWORDS

LLM Agents, Workflow Generation, Workflow Execution, Evaluation Framework, Prompt Engineering

1 INTRODUCTION

Recent advances in Large Language Models have expanded their role from passive language interfaces to active components capable of producing structured, executable artifacts. Beyond text generation, LLMs are increasingly used to orchestrate tools, invoke APIs and control multi-step processes, shifting evaluation from purely linguistic quality to execution-level correctness and reliability. As a result, task automation is entering a new paradigm in which executable content can be generated directly from natural language descriptions.

However, this transition exposes a fundamental abstraction gap: natural language expresses intent at a high level, while execution environments require precise, ordered and constraint-aware specifications. Workflow generation occupies this intermediate space, bridging human intent and machine execution by encoding control flow, data dependencies and decision logic. As such, workflows provide a natural and expressive representation for implementing LLM-driven automation.

1.1 Motivation

While the ability of Large Language Models to generate executable workflows has attracted significant attention, existing approaches vary widely in how the generation process is structured and the techniques they employ. These design choices entail implicit trade-offs between simplicity, controllability and computational cost, yet their impact on workflow quality and execution behaviour remains poorly understood.

Moreover, prior evaluations of agentic systems often emphasize task completion or final output correctness, providing limited visibility into structural stability, reasoning coherence and execution-level robustness. As a result, it is unclear which generation strategies are better suited for reliable deployment or how different failure modes manifest across approaches. This motivates the need for a systematic evaluation framework that enables controlled comparison of workflow generation strategies along multiple dimensions, beyond surface-level success metrics.

1.2 Challenges and Limitations

Although natural-language-driven automation is conceptually promising, contemporary LLMs encounter various challenges that undermine their accuracy and operational stability. Principal concerns primarily fall into three categories:

(1) **Reliability and Correctness Issues** These challenges directly affect the integrity and trustworthiness of the agent's output. LLM-based agents remain susceptible to **hallucinations** and **non-deterministic outputs**, which can result in the incorrect invocation of tools or the generation of erroneous results. A further concern is that workflows may complete execution without raising technical errors while nevertheless producing an incorrect outcome. This demonstrates that successful completion alone constitutes an insufficient criterion for correctness, necessitating more rigorous validation and verification mechanisms.

(2) **Robustness and Consistency Problems** These points highlight the agent systems' inherent **fragility**. The systems exhibit high sensitivity to input perturbations; even minor changes in the input, such as typographical errors or ambiguous phrasing, can precipitate substantial deviations

in the agent's behaviour and the quality of its output. Consequently, ensuring reproducibility and robustness across diverse inputs remains a significant hurdle.

(3) **Resource Management and Control** These issues relate to the **efficiency** and **governance** of the agent's actions. These systems frequently lack fine-grained control mechanisms over their operational processes. This deficiency can lead to the inefficient or inappropriate utilization of computationally expensive resources, such as unnecessarily repetitive calculations or the wasteful invocation of external APIs.

Collectively, while AI-driven agentic workflows offer substantial potential for advancing automation capabilities, their inherent unpredictability, susceptibility to hallucination and brittleness under edge cases underscore the imperative for systematic evaluation and rigorous empirical study.

1.3 Contributions

This project makes the following contributions:

(1) **Framework design:** a modular architecture for generating, executing, logging and evaluating AI-generated workflows.
(2) **Strategy implementations:** three contrasted workflow generation strategies (monolithic, incremental, bottom-up) plus a deprecated hierarchical prototype used to surface limitations.
(3) **Typed tooling interface:** a registry of 28 tools (23 atomic, 5 macro) used for generating and executing the workflows.
(4) **Evaluation suite:** automated metrics for similarity, correctness against task-specific constraints, intent resolution, reasoning coherence and efficiency (time, calls, tokens).

1.4 Report Outline

The remainder of this paper is organized as follows. Section 2 formalizes the problem and introduces the workflow representations. Section 3 describes the core architectural components for generating and executing workflows. Section 4 complements this view by detailing implementation choices, including core runtime components and application flow mechanisms. Section 5 describes the evaluation methodology, including the definition of computed metrics, the experimental setup and a comprehensive analysis of the empirical results. Section 6 discusses findings and limitations. Section 7 concludes with a summary of key lessons and possible future works.

2 PROBLEM STATEMENT

This section formalizes the workflow generation problem and introduces the workflow representations used throughout this work.

2.1 Problem Formulation

Given a natural-language request x and a set of available tools $\mathcal{T}$, the goal is to generate a workflow w that can be executed step-by-step to satisfy the user request. Formally, a workflow w is a directed acyclic graph (DAG) $G = (V, E)$ where:

- $V = \{v_1, v_2, \ldots, v_n\}$ is a set of nodes, each representing a *step* in the workflow.
- $E \subseteq V \times V$ is a set of directed edges encoding execution order and data dependencies.

Each step $v_i \in V$ is either:

- A **tool invocation** drawn from the tool registry $\mathcal{T}$, parametrized by inputs that may reference outputs of previous steps; or
- An **LLM-constructed step** for branching decisions, conditional logic or intermediate reasoning.

Figure 1 illustrates a representative workflow DAG containing both tool invocations and a branching decision node.

2.2 Workflow Representation

AI-Workflows supports two workflow representations, each offering different trade-offs between expressiveness and generation complexity:

Linear workflows are represented as a simple, sequential list of steps without explicit branching. Each step implicitly connects only to the next step in the list. This representation is simpler to generate, but cannot capture conditional decisions in during the execution.

Structured workflows augment the step list with explicit transition edges, thus enabling loops, conditional branches and error handling. This enables the agent to express complex task dependencies more explicitly and handle non-linear task flows more reliably.

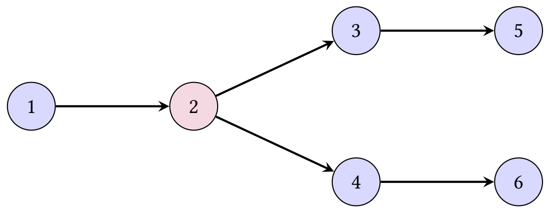

Figure 1: Representation of a structured workflow. Node 2 represents a branching decision, while all other nodes represent tool invocations.

In both workflow representations, each step corresponds to either a *call_tool* or a *call_llm* action:

call_tool Denotes a step in which a tool must be invoked by the runner application. In this case, the LLM is responsible for specifying the `tool_name` and the associated `parameters`.

call_llm Denotes a step in which the language model itself is called for reasoning or decision-making, requiring the LLM to construct an appropriate `prompt` to perform the intended action.

All steps, regardless of type, include a `unique identifier` for referencing outputs from previous steps and for managing transitions to subsequent steps, as well as a `thoughts` field that captures the agent's reasoning for coherence analysis. In structured workflows, steps need also to maintain a `transitions` list that defines the possible progression between steps, enabling conditional or sequential execution. Finally, the workflow concludes with a special `FinalStep`, which signals the termination of an execution path.

3 ARCHITECTURE

This section describes core components of AI-Workflows, including the orchestrator, generation strategies, enhancement features, tool infrastructure and agents.

3.1 Orchestrator

AI-Workflows is organized around a configurable orchestrator, responsible for coordinating all other components and managing the overall workflow lifecycle. The orchestrator primarily performs three tasks:

(1) **Workflow generation**, in which it follows a structured pipeline to produce a valid JSON artifact conforming to a specified schema. During this process, the orchestrator uses a user-defined set of features and chosen strategy, alongside with tools definitions and LLM agents, to produce an executable workflow that satisfies the user request. The implementation details are discussed in Section 4.2.

(2) **Workflow execution**, in which it takes a previously generated workflow and executes it step by step. At each step, the orchestrator produces a detailed, step-indexed state trace, capturing intermediate outputs, decisions and interactions with agents or tools, enabling evaluation of correctness and reproducibility of the subsequent workflows. The implementation details are discussed in Section 4.3.

(3) **Workflow save and visualisation**, which stores the generated workflow in JSON format for later use and also creates an interactive HTML visualization using PyVis, as shown in Figure 2.

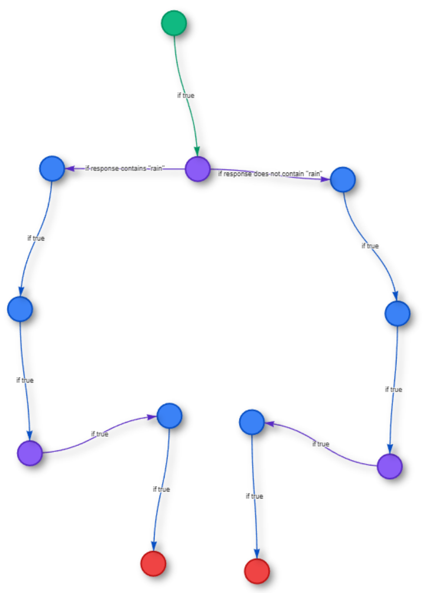

Figure 2: Example of workflow visualization generated using PyVis.

3.2 Generation Strategies

AI-Workflows implements three primary generation strategies, each representing a distinct approach for decomposing a user request for generating an executable workflow. During the construction stage, a single strategy is selected by the user and applied by the orchestrator to guide the generation process between the pre-processing and post-processing features.

3.2.1 ***Monolithic Strategy.***

This approach treats the user request as an indivisible specification and generates the entire workflow in a single pass.

This strategy favours simplicity and low orchestration overhead, as it avoids intermediate validation steps. However, it places a high cognitive and computational burden on the generator agent, which must understand the full context, resolve all dependencies and constraints simultaneously. As a result, it is best suited for relatively small, well-scoped requests, while it may struggle with complex tasks that require long-horizon planning or dynamic refinement.

3.2.2 ***Incremental Strategy.***

This approach still relies on a single generator agent that produces the workflow incrementally, emitting it as a sequence of batches of executable steps.

This strategy strikes a balance between monolithic generation and fully decomposed planning approaches. By allowing the generator to focus on a limited horizon at each step, it improves scalability and stability for longer workflows while preserving a coherent global structure. However, since planning and execution reasoning remain embedded within a single agent, opportunities for early validation or correction are limited compared to more interactive or multi-agent strategies.

3.2.3 ***Bottom-Up Strategy.***

This approach constructs a workflow through a staged planning process that progressively enriches the original user request with increasingly detailed execution constraints. Unlike strategies where planning and generation are tightly coupled, this approach explicitly separates planning concerns across multiple specialized planner agents.

This strategy emphasizes feasibility and correctness of execution by anchoring generation in concrete tool-level decisions. However, the sequential dependence between planning stages introduces additional latency and reduces flexibility in revising earlier decisions once subsequent stages have begun.

3.2.4 ***Hierarchical Strategy** (deprecated).*

This approach adopts a divide-and-conquer approach in which the user request is first decomposed into a set of grouped subtasks by one planner agent. A generator agent then produces partial workflows for each subtask independently. Finally, a merging agent combines the generated subtasks into a single executable workflow.

This strategy has been deprecated in AI-Workflows, as detailed in Discussion 6.1

3.3 Generation Features

On top of strategies, AI-Workflows also implements pre-processing and post-processing features that can be enabled for each strategy. Features are designed to be orthogonal: while strategies define the unique high-level pattern for generating the workflow, features only provide additional capabilities that enrich or constrain the strategy execution. They have been differentiated from strategies as they don't provide sufficient structural or procedural information to be executed independently.

Features can be applied at different stages of the generation process. *Pre-processing features* operate before the strategy execution and aim to enrich the initial user request, for example by adding missing contextual information. On the other hand, *post-processing features* operate after the strategy execution directly on the generated workflow.

AI-Workflows currently supports three main features:

- **Chat clarification**, which enables interactive, multi-turn communication with the user. This feature trivially helps each strategy, as it allows the system to resolve ambiguities, acquire missing information and iteratively align the generated workflow with the user's intent before strategy execution.
- **Refinement**, which focuses on improving an initial workflow by identifying and correcting potential issues, like invalid steps, broken transitions or incomplete logic.
- **Validation-Refinement**, which is a stronger version of the *refinement* feature as it alternates between validating intermediate results and refining them until a good quality workflow is produced or a fixed number of iteration rounds is met (default: 5).

These features are not mutually exclusive and can be combined together, allowing for flexible and expressive configurations of the overall reasoning process.

3.4 Tools

AI-Workflows provides a rich set of ready-to-use tools, including:

Macro Tools Each tool encapsulates a relatively complex operation, such as `"EXTRACT user info"`. This simplifies the agent's planning by reducing the number of decisions required. Macro tools correspond to intuitive human-level tasks, but they may limit flexibility: if the task requires a slightly different operation than the macro tool provides, the agent must improvise within the LLM reasoning process, potentially reducing reliability.

Atomic Tools The agent is provided with many small, composable tools. For example, the task `"EXTRACT user info"` can be decomposed into `"GET user"`, `"parse HTML"` and `"READ biography"` atomic actions. This increases expressiveness and makes workflows more transparent, but planning becomes more challenging: the LLM must chain multiple small steps, increasing the risk of local errors or incorrect tool usage.

In addition to the built-in tools, users can easily define custom tools through a dedicated *tool decorator*. Each function annotated with this decorator is automatically registered to the tool registry and becomes visible system-wide.

The **tool registry** represents a core component of the proposed solution, as it is responsible for organizing and managing all the available tools. At startup, it searches for all the decorated tool functions, automatically loads them and retrieves useful metadata such as input parameters and output keys to generate unstructured tool representations that are later injected into the system prompt of the LLM agents. This mechanism ensures that agents can select and use tool definitions in a consistent and extensible manner.

3.5 Agents

AI-Workflows leverages LLM agents as the core components for both workflow generation and execution. These agents are capable of producing outputs in multiple formats, supporting both free-form chat responses and structured content that conforms to predefined schemas. The behaviour of each agent is determined by a system prompt, which instructs the model on the specific reasoning style, action format or domain constraints to follow. This allows AI-Workflows to guide agents consistently across different tasks while maintaining interpretability and reproducibility of results. Furthermore, agents are specialized according to their roles (e.g., generator, executor, planner), supporting a multi-agent architecture.

4 IMPLEMENTATION

This section describes the key implementation principles of the AI-Workflows framework. It presents the primary runtime objects used throughout workflow generation and execution, as well as the high-level definitions that guide these processes.

4.1 Environment and Core Runtime Objects

The project has been carried out using Python due to its extensive ecosystem for rapid prototyping, mature support for data modeling and validation and seamless integration with contemporary large language model (LLM) APIs.

Within this environment, the implementation is structured around a small set of core runtime objects that explicitly manage state, enforce structural constraints and mediate interactions between LLM agents and external tools. These objects form the operational backbone of the framework and are shared across both the generation and execution phases, ensuring consistency and traceability throughout the workflow lifecycle.

4.1.1 ***Generation Context.*** The *Context* is a data class that encapsulates all state required during the workflow generation phase. This object is threaded through pre-processing features, strategy execution, and post-processing features, enabling stateful transformations while preserving a modular and compositional design.

```
class Context(BaseModel):
    # Contextual input data
    agents: AgentSchema
    response_model: Type[BaseModel]
    prompt: str
    available_tools: List[Tool]

    # Contextual output data
    workflow: BaseModel
    workflow_path: str
```

During generation, the context may be modified in two ways: pre-processing features may update the input prompt, while generation strategies and post-processing features may initialize or incrementally update the workflow artifact.

4.1.2 ***Type-Safe Schemas.*** All externally structured entities, including workflows and LLM-generated outputs, are specified using typed schema models. In AI-Workflows, these schemas are implemented using Pydantic, and every artifact produced by an LLM is subject to schema validation prior to further processing.

```
class MessageResponse(BaseModel):
    result: str = Field(..., description="...")
```

Each schema field is annotated with a natural-language description that guides the LLM in populating its value. This approach significantly reduces system prompt verbosity while mitigating hallucinations and context drift during LLM inference.

4.1.3 ***Decorator-Based Tool Registry.*** As discussed in Section 3.4, tools are registered automatically through a `@tool` decorator, which extracts metadata from function signatures using the *inspect* module. This design enables straightforward creation of new tools:

```
@tool(name="calc", description="...", category="math")
def calculator(expression: str) -> CalculatorOutput:
    return CalculatorOutput(result=eval(expression))
```

The registry performs lazy initialization by scanning the *tools* directory tree and constructing an in-memory index of available tools together with their input and output schemas.

4.1.4 ***Agents.*** The system defines multiple agent roles, each implemented as a thin wrapper around an LLM provider client, such as the Google Generative AI SDK or the Cerebras Cloud SDK. Each implementation extends a shared abstract class, `AgentBase`, which establishes a common standard across providers and facilitates seamless model interchangeability.

4.1.5 ***Generation Strategies.*** All generation strategies inherit from a common abstract base class, `StrategyBase`, which exposes a uniform interface that allows the orchestrator to interchange strategies transparently.

```
class StrategyBase(ABC):
    @abstractmethod
    def generate(self, context, **kwargs) -> BaseModel:
        pass
```

Concrete strategies implement the *generate* method by composing agent calls and updating the shared `Context`. For instance, the bottom-up strategy first delegates analysis tasks to planner agents, then invokes a generator agent to assemble a workflow conditioned on the intermediate planning outputs.

4.2 Workflow Generation

The generation phase transforms a natural language request into an executable workflow through a configurable pipeline coordinated by the orchestrator. As illustrated in Figure 3, this process consists of three main stages:

(1) Pre-processing features
(2) Strategy execution
(3) Post-processing features

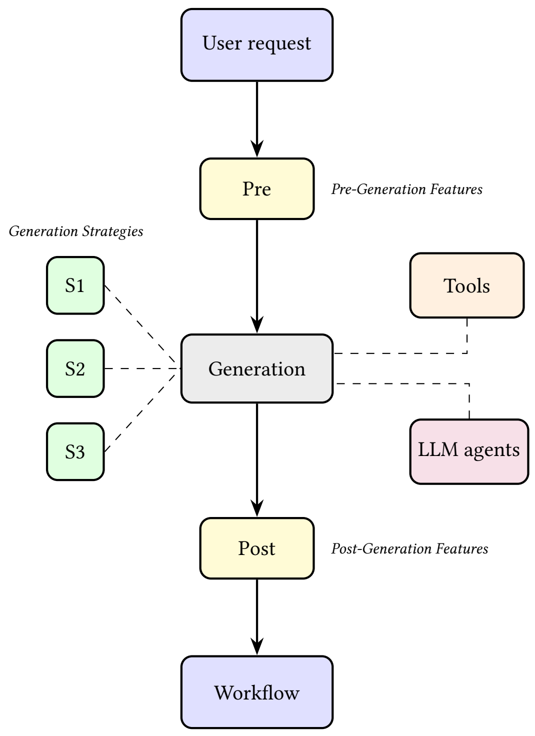

Figure 3: Generation flow defined by the orchestrator.

4.2.1 ***Pre-processing Features.*** Pre-processing features modify the user prompt before workflow generation begins. Currently, the only implemented pre-processing feature is the chat clarification, which initiates a multi-turn dialogue with the user:

(1) A structured chat session is initialized with a specialized system prompt
(2) The agent generates clarifying questions based on the initial request
(3) User responses are collected iteratively until sufficient information is obtained
(4) The conversation history is injected into the original prompt as additional context
(5) The enriched prompt replaces `context.prompt`

This approach preserves the original request while augmenting it with disambiguating information, enabling downstream components to operate on a more complete specification.

4.2.2 ***Strategy Execution.*** The selected strategy receives the context and produces a workflow conforming to the specified response model. Each strategy implements distinct decomposition logic:

Monolithic Strategy Constructs a complete system prompt by injecting tool definitions into the generation template, then issues a single structured generation call. The LLM must simultaneously reason about task decomposition, optimal tool selection, parameter binding and control flow.

Incremental Strategy Maintains a structured chat session and iteratively generates batches of 1-3 steps until completion. At each iteration:

1. The current workflow state is summarized using a sliding window (most recent 3 steps in full detail, earlier steps compressed)
2. The agent generates the next batch of steps as a structured response
3. Steps are validated and appended to the workflow
4. If the agent signals completion, metadata is extracted from the final step and the workflow is returned

Bottom-Up Strategy Executes four sequential phases, each delegated to specialised planner or generator agents:

1. **Tool Identification**: Analyses the request and returns a structured list of required tools with their purposes and data dependencies
2. **Tool Ordering**: Given the tool list, determines an execution order respecting data flow constraints
3. **Control Flow Analysis**: Identifies decision points requiring branching or conditional execution
4. **Workflow Assembly**: The generator agent synthesizes the final workflow by conditioning on all prior analysis results and the original request

4.2.3 ***Post-Processing Features.*** Post-processing features refine the generated workflow. The validation-refinement feature, for example, implements an iterative review loop:

1. A reviewer agent examines the workflow and produces a structured assessment containing either a list of issues or an approval signal
2. If issues are found, they are passed to a refiner agent that modifies the workflow to address them
3. The updated workflow is re-submitted for review
4. This process repeats for up to N rounds (default: 5) or until approval

Both reviewer and refiner operate with access to the original user prompt and tool definitions, enabling them to validate semantic correctness beyond schema compliance.

4.3 Workflow Execution

As illustrated in Figure 4, the execution phase interprets a generated workflow and produces a complete execution trace by invoking tools and LLM reasoning as specified. The executor operates as a state machine that processes steps according to their defined transitions.

4.3.1 ***State Management.*** Execution state is maintained as a dictionary mapping step IDs to their outputs:

```
state = {
    "1": {"forecasts": ["clear sky", "rainy", ...]},
    "2": "outdoor",
    "3": {"news": [{"title": "...", ...}, ...]},
    ...
}
```

The LLM executor agent receives this state alongside the workflow definition at each iteration and must determine the next step to execute based on transition conditions. This design decouples workflow structure (static) from execution state (dynamic).

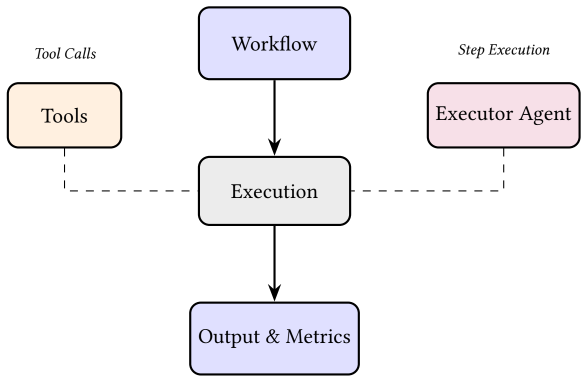

Figure 4: Execution flow defined by the orchestrator.

4.3.2 ***Step Execution Loop.*** The executor operates in a loop until a `FinalStep` is reached:

1. **Step Identification**: The executor agent analyses the workflow and current state to identify the next step according to transition logic
2. **Action Dispatch**:
   - For `call_tool` actions: The application extracts the tool name and parameters with all the placeholder references already resolved by the LLM (e.g., `{2.response}` → `"outdoor"`), and invokes the tool via the registry
   - For `call_llm` actions: The executor agent generates a response directly by resolving the prompt template against current state
3. **State Update**: The output is added to state, indexed by step ID

This design ensures deterministic execution while allowing the LLM to maintain reasoning about control flow without modifying the workflow structure at runtime.

5 EVALUATION

The proposed alternative designs will be evaluated through controlled experiments. The experimental plan includes:

- **Task Suite:** The systems will be assessed on a medium-complexity prompt, as to ascertain its capabilities while still keeping the simplicity of the final result limited. It is to say that high complexity prompts still give good results, but are less useful to give objective metrics.

- **Variable Toolset:** Different toolsets will be used to ascertain how the designs manage to decide which tools are necessary and how to reach their objectives.
- **Additional Features:** Establishing in what way the additional features described previously improve the results of the final workflows.

Each experiment will be run multiple times to account for the inherent non-determinism of LLMs. Once the experimental phase is complete, the collected metrics will be analysed to identify which architectural configurations perform best under specific conditions. All results are shared as described in Section 7.3.

5.1 Metrics

Evaluating AI-generated workflows requires a mix of correctness, robustness and performance measures. Key metrics include:

Robustness Measures how sensitive the system is to minor changes in input phrasing or content. This is crucial because real users often issue imperfect or ambiguous prompts; a reliable agent should remain stable under such variations rather than exhibiting dramatic shifts in behaviour.

Consistency and Reproducibility Evaluates whether repeated executions with similar inputs yield the same workflow or result. Reproducibility is essential for debugging, auditing and deploying workflows in production environments, where unpredictable variation can undermine trust and make validation difficult.

Task Completion vs. Correctness Assesses not only whether a workflow finishes but whether it accomplishes the intended task accurately. Since LLM agents can "complete" a workflow while still producing incorrect outcomes, this metric ensures that success is defined by correctness, not merely by the absence of errors.

Intent Resolution Measures how well the agent interprets the user's underlying goal rather than the literal surface form of the prompt. Effective intent resolution is vital for natural-language interfaces, where users expect the system to infer context, resolve ambiguities and act in alignment with their actual objectives.

Reasoning Coherence Inspects the internal chain of reasoning for logical structure and consistency. Coherent reasoning provides insight into whether the workflow was constructed based on sound intermediate steps, which is especially important when evaluating complex or multi-step tasks.

Efficiency Records time taken, number of LLM calls, or overall computational cost required to finish a task. This is an important practical metric, as highly inefficient workflows may be unusable in real-world applications even if they are correct and cost efficiency directly affects scalability.

5.2 Measurements

In order to give all metrics a way to be compared between each experiment, multiple workflows were generated utilising different wordings of the same prompt and some measurements were either taken during the execution of the process or computed after it on the finished workflows. In particular:

5.2.1 ***Measured data.*** Throughout the workflow generation and execution processes, the total time, number of LLM calls and tokens consumed are recorded, providing a quantitative measure of the **efficiency** of both generation and execution. This metric is also tracked for each optional feature the user has enabled.

5.2.2 ***Computed data.*** After the workflow generation and execution are completed, several metrics are computed from the collected data to evaluate performance and quality. These includes:

Workflow Robustness A pairwise similarity matrix between workflows is computed by aligning comparable steps, identifying matches and comparing corresponding transitions. The overall **workflow robustness** score is obtained as a weighted sum of the partial results, with the step alignment contributing most significantly to the final outcome.

Execution Consistency A pairwise similarity matrix between executions is computed by comparing the outcomes using recursive object analysis (i.e., pairwise step similarity) and greedy step matching. The final **execution robustness** score is a weighted combination of output alignment and coverage.

Correctness Each workflow is assessed against a uniquely defined constraint file corresponding to the specific user request, which specifies the expected structure and properties of the workflow. The evaluation considers the number of times each tool should be invoked, the total number of LLM calls, the occurrence of specific branch transitions guided by relevant keywords and the overall number of workflow steps. The overall **correctness** score is also here obtained by a weighted sum of the individual scores.

Reasoning Coherence The step-level thoughts produced by the LLM during the workflow generation process are analysed to assess whether they form a coherent and continuous chain of reasoning. Together with a structural coherence analysis, this provides a clear understanding of the overall **reasoning** score.

Intent Resolution The workflow's target objective is compared to the initial user request using embedding-based similarity, providing a quantitative measure of how effectively the user's **intent** is captured and fulfilled.

5.3 Experiments

Various experiments were conducted to provide concrete results on how the different designs perform according to the metrics described above. In addition, more focused analyses are presented to highlight specific advantages offered by particular configurations.

5.3.1 Experiment 1: Linear versus Structured Workflow Representations. This experiment demonstrates the limitations inherent in linear workflow architectures when confronted with prompts requiring divergent execution paths. A comparative analysis was conducted between the Monolithic strategy employing Linear and Structured representations, respectively, using a prompt requesting activity planning based on weather conditions.

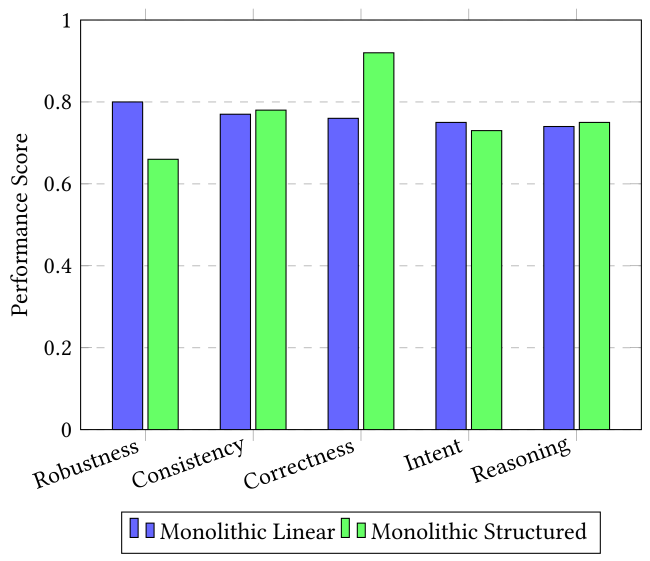

Figure 5: Performance comparison between Linear and Structured approaches in the Monolithic strategy

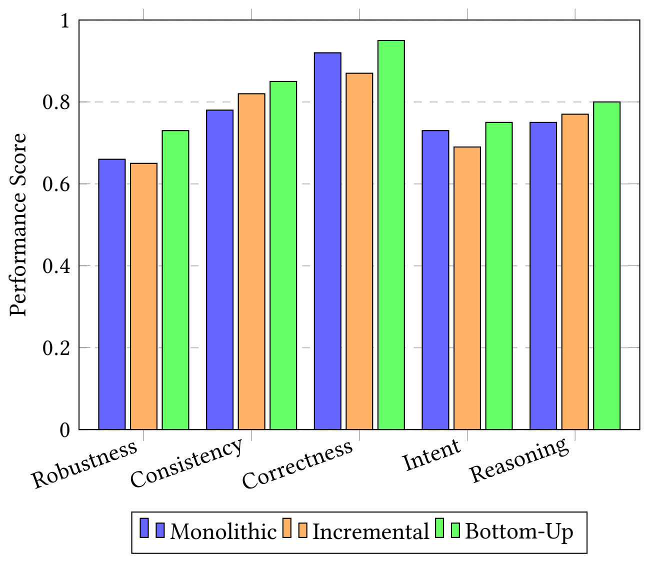

Figure 6: Comparative performance analysis of generation strategies

Table 1: Efficiency metrics for Linear and Structured Monolithic strategies

Strategy	Phase	Time (s)	API Calls	Tokens
Monolithic Linear	Generation	2.5	1	7,876
	Execution	7.6	9	34,548
Monolithic Structured	Generation	2.2	1	8,022
	Execution	5.77	7	25,213

The results indicate that while robustness and consistency metrics appear favorable for the linear design, this apparent performance is attributable to the inherent limitations in available execution paths, resulting in convergent outcomes. The most significant limitation of linear workflows manifests in correctness scores, as the inability to implement conditional branching necessitates the generation of superfluous workflow steps, thereby degrading overall correctness. From an efficiency perspective, the linear approach requires additional execution steps, resulting in increased API calls and a 45% average increase in token consumption.

These findings establish a clear disadvantage for linear workflows. Consequently, while linear representations remain supported, subsequent experiments exclusively employ structured workflows, which demonstrate superior performance for complex prompt scenarios.

5.3.2 *Experiment 2: Comparative Analysis of Generation Strategies.* This experiment provides an objective comparison of the three generation strategies, identifying the relative strengths and weaknesses of each approach.

Table 2: Efficiency metrics for different generation strategies

Strategy	Phase	Time (s)	API Calls	Tokens
Monolithic	Generation	2.2	1	8,022
	Execution	5.77	7	25,213
Incremental	Generation	7.53	4	32,421
	Execution	8.11	7	27,051
Bottom-Up	Generation	6.41	4	13,467.5
	Execution	18.05	7	25,980

All three generation strategies demonstrate comparable performance, with correctness exceeding 90%, ensuring satisfactory workflow outputs for end users. However, the Bottom-Up design exhibits superior performance across all evaluation metrics, while maintaining token consumption marginally higher than the Monolithic approach. The Incremental design demonstrates improvements over Monolithic in consistency and reasoning, though it exhibits reduced performance in correctness and intent metrics, suggesting that multiple sequential LLM calls may cause deviation from the original workflow objectives.

Overall, all three strategies achieve highly satisfactory results. Performance scores exceeding 60% indicate adequate functionality, while scores reaching 80% demonstrate excellent performance in the respective metric.

5.3.3 *Experiment 3: Impact of Generation Features.* This experiment evaluates the effectiveness of generation features using the Monolithic strategy as the baseline. Each feature is compared independently against the standard Monolithic configuration presented in Experiment 2.

First will be presented the Validation-Refinement feature.

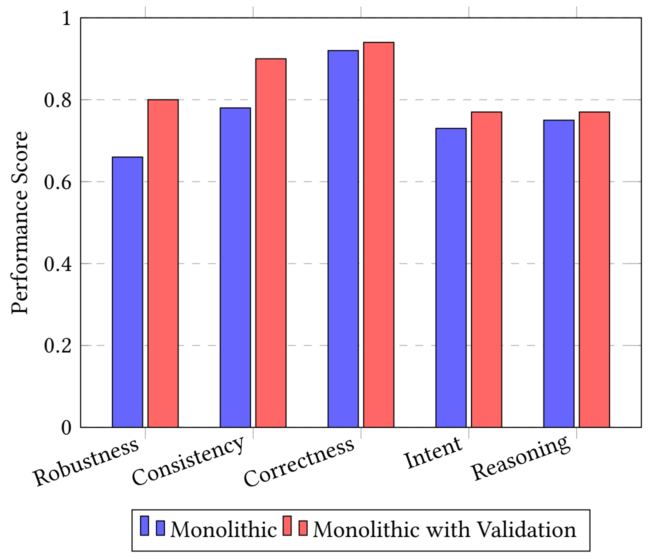

Figure 7: Performance impact of Validation-Refinement feature

The data clearly demonstrate that the Validation-Refinement feature significantly enhances workflow robustness and consistency. However, improvements in other performance dimensions remain modest.

Now will be shown the Chat Clarification feature.

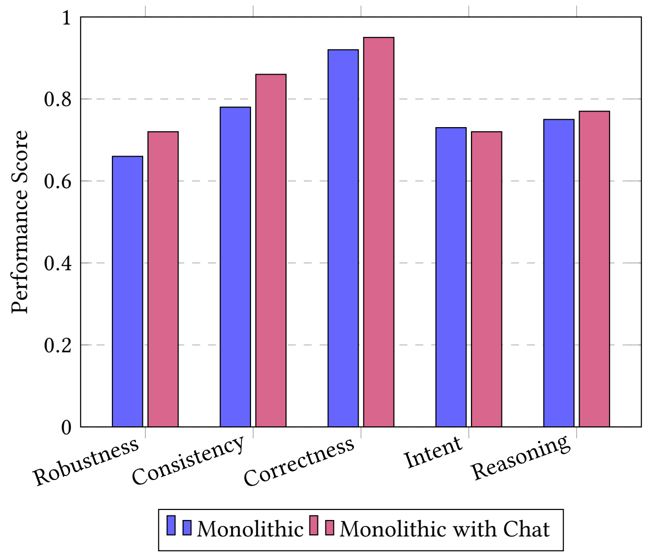

Figure 8: Performance impact of Chat Clarification feature

The Chat Clarification feature exhibits limited quantitative improvements. Experimental observations reveal that generated queries tend to focus on highly specific details (e.g., recipient email addresses). While this feature enriches workflows with previously omitted information, it has minimal impact on structural workflow quality, rendering its effects challenging to quantify through the established metrics.

At last will be displayed the Refinement feature.

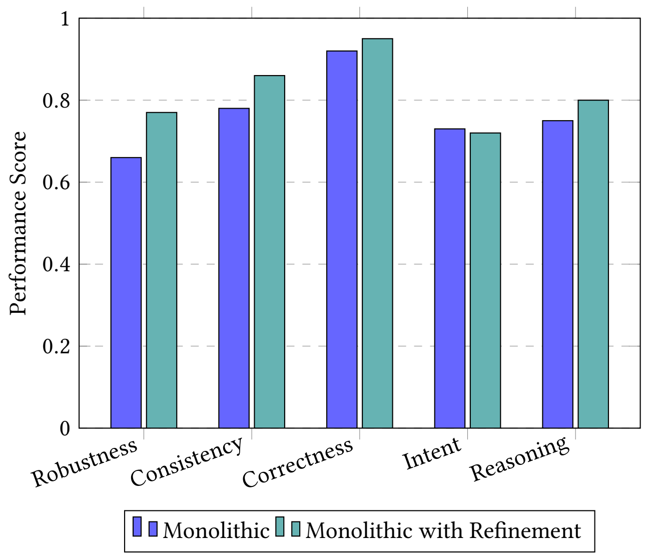

Figure 9: Performance impact of Refinement feature

Modest improvements in consistency and correctness are observed with the Refinement feature, consistent with its workflow optimization objectives. The slight degradation in intent metrics may be attributed to the iterative refinement process distancing the workflow from its initial specification, as optimization prioritizes output quality over strict adherence to the original plan.

Table 3: Efficiency metrics for Monolithic strategy with enhancement features

Strategy	Phase	Time (s)	API Calls	Tokens
Monolithic	Generation	2.2	1	8,022
	Execution	5.77	7	25,213
Monolithic with Validation	Generation	2.79	1	8,172
	Execution	9.09	7	25,478
	Review	8.29	3	32,141
	Refinement	12.81	2	20,857
Monolithic with Chat	Generation	2.36	1	8,498.5
	Execution	5.1	7	24,009
	Chat	4.52	5	18,467
Monolithic with Refinement	Generation	2.06	1	8,464
	Execution	7.85	8	30,971
	Refinement	3.67	8	10,006

Each feature provides distinct performance improvements. While the quantitative gains appear modest, this reflects the inherently strong performance of the baseline design. More complex prompts may yield more pronounced improvements. However, efficiency metrics reveal substantial cost increases for all features, with baseline costs at minimum doubling and Validation incurring a five-fold increase. Although the features demonstrate promise, the cost implications significantly outweigh the marginal performance gains observed with the already-effective baseline design.

5.3.4 Experiment 4: Toolset Granularity Analysis. This experiment quantifies the performance implications of toolset granularity constraints. The Bottom-Up strategy, identified as the optimal baseline approach in previous experiments, serves as the evaluation framework.

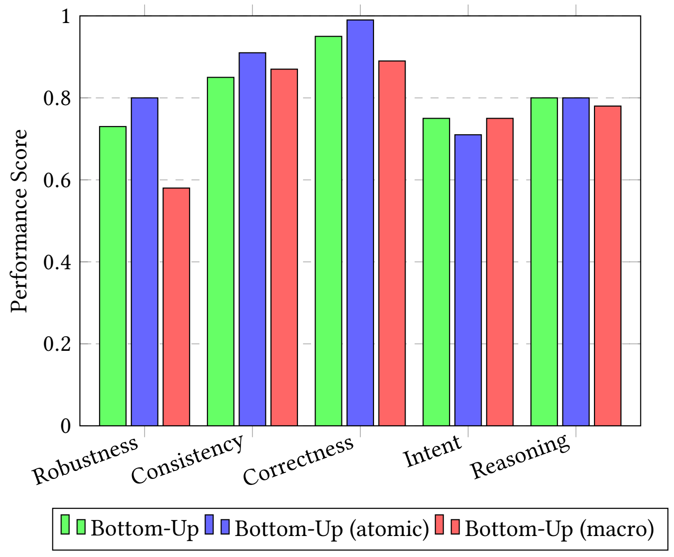

Figure 10: Performance comparison across toolset granularities

Table 4: Efficiency metrics for Bottom-Up strategy with varying toolset granularities

Strategy	Phase	Time (s)	API Calls	Tokens
Bottom-Up	Generation	6.41	4	13,467.5
	Execution	18.05	7	25,980
Bottom-Up (atomic only)	Generation	10.575	4	13,398
	Execution	7.99	7	25,013
Bottom-Up (macro only)	Generation	7.205	4	11,339
	Execution	9.28	6	24,132

The results clearly establish the superiority of the atomic toolset approach. The restriction to atomic tools proves non-constraining in practice, as workflows rarely require macro-level operations, with atomic tools providing comprehensive coverage of workflow requirements. Enhanced robustness and consistency emerge from the reduced option space, which promotes convergence toward similar design solutions. Conversely, the macro-only toolset exhibits degraded performance across most metrics, with particularly pronounced deterioration in robustness, as the system struggles to maintain structural stability while employing creative workarounds to achieve workflow functionality.

5.3.5 Experiment 5: Strategic Feature Integration. To demonstrate the potential for substantial performance improvements through features, this experiment evaluates strategically selected feature combinations. The combinations shown will be Incremental with Revision-Refinement and Bottom-Up with Refinement.

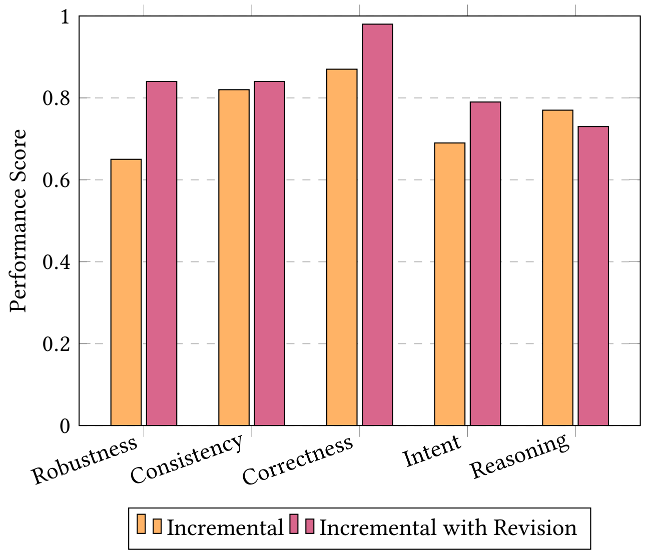

Figure 11: Performance enhancement of Incremental strategy with Validation-Refinement

Table 5: Efficiency metrics for Incremental strategy with Validation-Refinement

Strategy	Phase	Time (s)	API Calls	Tokens
Incremental	Generation	7.53	4	32,421
	Execution	8.11	7	27,051
Incremental with Refinement	Generation	9.09	4	28,122
	Execution	12.69	8	31,250
	Review	15.865	3	33,490
	Refinement	8.7877	2	19,943

The Validation-Refinement feature yields substantial improvements to the Incremental design, significantly enhancing robustness and correctness metrics. Intent comprehension demonstrates notable advancement, indicating improved alignment with prompt objectives. A marginal decline in reasoning scores may reflect the trade-off inherent in the revision process: as structural convergence toward satisfactory outcomes increases, some deviation from initial conceptual focus occurs.

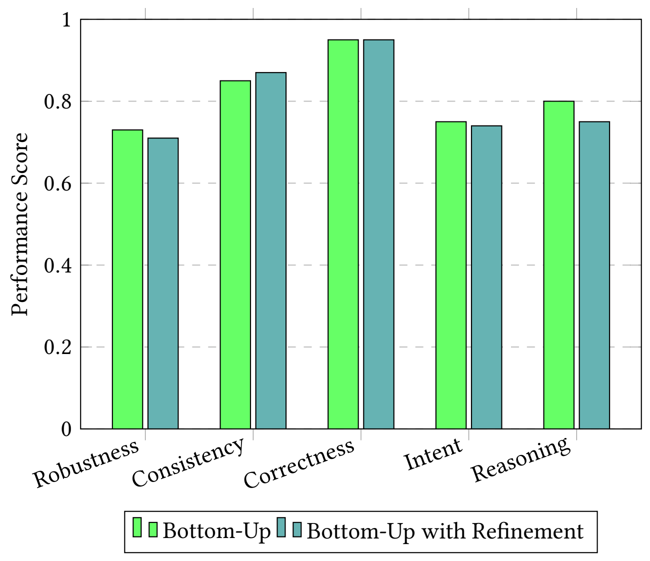

Figure 12: Performance impact of Refinement on Bottom-Up strategy

Table 6: Efficiency metrics for Bottom-Up strategy with Refinement

Strategy	Phase	Time (s)	API Calls	Tokens
Bottom-Up	Generation	6.41	4	13,467.5
	Execution	18.05	7	25,980
Bottom-Up with Refinement	Generation	6.52	4	13,212
	Execution	11.23	7	24,378
	Refinement	2.41	1	8,968

The Refinement feature yields minimal improvements to the Bottom-Up design, as this strategy inherently generates workflows with limited optimization potential, producing highly satisfactory results independently.

These findings establish that features function as powerful optimization tools in specific contexts where their integration addresses identifiable system weaknesses. However, their effectiveness is contingent upon baseline performance limitations. Consequently, these features are best conceptualized as supplementary optimization mechanisms rather than universal performance enhancers for already-effective systems.

5.3.6 Experiment 6: Large Language Model Generalizability. This final experiment evaluates whether workflow representation quality varies across different LLM architectures employed in the generation phase. As an example will be presented Gemini 2.5 Flash.

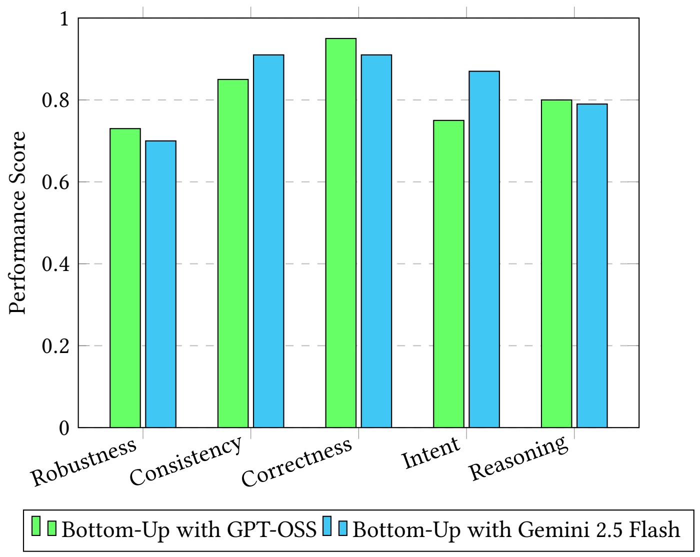

Figure 13: Performance comparison across different LLM architectures

Table 7: Efficiency metrics for Bottom-Up strategy with different LLMs

Strategy	Phase	Time (s)	API Calls	Tokens
Bottom-Up with GPT-OSS	Generation	6.41	4	13,467.5
	Execution	18.05	7	25,980
Bottom-Up with Gemini	Generation	48.28	1	16,678
	Execution	48.395	4	16,953

These results demonstrate sufficient system robustness for generalization across different LLM architectures. However, effectiveness remains partially constrained by model capacity, as lower-capability models exhibit increased susceptibility to task misinterpretation and malformed output generation, as elaborated in Discussion 6.3.

6 DISCUSSION

This section discusses failure modes and limitations of the AI-Workflows framework emerged after the experimental evaluation.

6.1 Hierarchical Decomposition and Fragment Merging

The hierarchical prototype demonstrated that compositional workflow generation via fragment merging is fundamentally brittle under strict schema constraints. While hierarchical decomposition has long been studied in automated planning as a way to manage complexity through task hierarchies, prior work already highlights persistent challenges in maintaining semantic coherence across decomposed components. Classical hierarchical planning formalisms often require extensive domain authoring and suffer from limited scalability, largely because local task decomposition can obscure global dependencies and constraints [1]. These limitations become particularly pronounced in data-driven generation settings, where explicit global coordination is absent.

Although the divide-and-conquer strategy improves modularity at generation time, it implicitly assumes that independently generated fragments can be composed without shared global context. In practice, this assumption fails: step-ID collisions arose from generators operating with local views of the workflow, while independently produced fragments encoded incompatible structural or semantic assumptions, such as inconsistent data conventions or execution ordering. Cross-fragment dependencies were frequently omitted, and strict JSON schema enforcement further amplified these inconsistencies into hard execution failures. Since schema validation guarantees syntactic well-formedness but cannot resolve semantic conflicts, the merging agent was limited to mechanical reconciliation, unable to repair or reinterpret conflicting fragments without violating schema constraints. These observations underscore fundamental limitations of unguided hierarchical decomposition for reliable workflow synthesis in AI-Workflows.

6.2 Context Degradation in Incremental Strategy

When workflow nodes are generated through multiple independent steps (i.e., incremental strategy), a partial loss of contextual information may occur, even when summarization mechanisms are employed. This phenomenon, referred to as **context degradation**, can cause the loss or hallucination of data references and step dependencies, leading to incomplete or inconsistent workflows. As a result, downstream steps may be generated without full awareness of the outputs produced by earlier stages, potentially affecting the overall correctness and interpretability of the workflow.

During evaluation, it was observed that some workflows may appear more linear than intended, despite the underlying workflow model being structured, as the LLM attempts to reconstruct missing structural information retrospectively. Additionally, context drift may lead to the generation of misleading transitions or partially disconnected branches. Although such issues were significantly less frequent than those observed in the initial **hierarchical architecture** (Discussion 6.1), they nonetheless represent a limitation of the incremental generation strategy.

6.3 Task Incomprehension and Malformed Output

Task incomprehension and malformed outputs remained a recurring failure mode, particularly for lower-capacity language models. These models often failed to fully capture the user's intent, misinterpreting high-level objectives or overlooking implicit constraints. Such semantic ambiguity propagated into the generated workflows, resulting in malformed control-flow structures, missing or incorrect transitions and dead-end branches. In conditional workflows, this frequently forced artificial branch merging across semantically distinct tasks, producing logically invalid execution paths, even when the output appeared superficially well-structured [4].

In other cases, malformed outputs arose despite correct task understanding, reflecting brittleness in structured generation rather than semantic failure. Although the inclusion of extensive descriptive guidance within schema fields, together with simplified system prompts, improved output coverage and reduced outright refusals, lower-capacity models sometimes still struggled to reliably adhere to strict schema constraints, thus producing non-executable workflows. Together, these observations indicate that neither detailed schemas nor post-hoc validation can fully compensate for limited model capacity, exposing a fundamental tension between robustness and accessibility in workflow generation.

6.4 Quality of Metrics

The adopted metrics do not aim to provide an exact or exhaustive evaluation of the generated workflows. Instead, they represent a compromise between evaluation accuracy and feasibility. A fully manual assessment of all workflows would have been prohibitively expensive in terms of time and effort; therefore, automated metrics were preferred to enable scalable and repeatable analysis.

While these metrics can certainly be improved and refined, they still offer a meaningful and consistent indication of workflow quality. In particular, they allow the identification of general trends and behaviours across different experimental settings, even if they are not sufficiently precise to capture all qualitative aspects of individual workflows. As a result, the metrics should be interpreted as indicative measures rather than definitive judgments, providing a reasonable baseline for comparative analysis and high-level evaluation.

7 CONCLUSION

This section provides a summary of the work conducted, highlights the key insights derived from the development and evaluation phases and outlines potential directions for future enhancements to improve the reliability, adaptability and generalizability of LLM-powered workflow systems.

7.1 Summary

This work introduced AI-Workflows, an evaluation framework for systematically analysing LLM-powered workflow generation strategies under controlled and reproducible conditions. By framing workflow generation as the synthesis of executable DAGs from natural language requests, the framework moves beyond surface-level task completion metrics and enables fine-grained investigation of structural stability, functional correctness, intent resolution, reasoning coherence, and efficiency. Through the implementation and comparison of multiple generation strategies, workflow representations, and enhancement features, AI-Workflows establishes a principled foundation for studying the behaviour of agentic workflow systems at both generation and execution stages.

7.2 Key Lessons

The systematic evaluation of workflow generation within the AI-Workflows framework yields several critical insights for the design of LLM-based agentic systems. First, the choice of workflow representation is fundamental; while linear workflows offer lower orchestration overhead, structured directed acyclic graphs (DAGs) are indispensable for tasks requiring conditional logic and branching, as they significantly reduce the generation of superfluous steps and improve overall correctness. Second, the experimental data reveals a significant trade-off between planning complexity and execution reliability. The bottom-up strategy, characterized by staged planning phases, consistently demonstrates superior performance across all evaluation metrics, albeit at the cost of higher latency and sequential dependencies. Third, while orthogonal features such as iterative refinement and chat clarification enhance workflow consistency and robustness, they incur substantial increases in token consumption and computational time, suggesting that their deployment should be calibrated based on task-specific requirements. Finally, the granularity of the toolset profoundly impacts planning success; while macro tools simplify the reasoning process by reducing the decision horizon, they can restrict the system's flexibility in addressing non-standard requests. These findings underscore that there is no "one-size-fits-all" strategy for workflow generation; instead, optimal performance requires a balanced configuration of representation, planning decomposition, and resource efficiency.

7.3 Future Work

Future research directions include improving the evaluation framework by developing more reliable and robust metric systems, as well as incorporating human-in-the-loop validation to enhance assessment accuracy. Efforts can also focus on improving workflow generation quality and extending support to additional LLM backends, thereby increasing the generalizability of the findings. These enhancements would further strengthen AI-Workflows as a foundation for principled and reliable study and deployment of LLM-powered workflow systems.

CODE

All experimental results will be made publicly available in a Google Sheets document [3], while the code will be released through the GitHub repository [2], together with detailed instructions for execution to guarantee reproducibility.

REFERENCES

[1] Pascal Bercher, Ron Alford, and Daniel Höller. 2019. A Survey on Hierarchical Planning-One Abstract Idea, Many Concrete Realizations.. In *IJCAI*. 6267–6275.
[2] Reichert Alex Costalonga Matteo. 2026. AI-Workflows. https://github.com/wamuumu/ai-workflows.
[3] Reichert Alex Costalonga Matteo. 2026. AI Workflows Metrics. https://docs.google.com/spreadsheets/d/1NI1UVD8nQ_wCcqrUvtZBqOlGUgmrKu4HsEMqdm3orDs/edit?usp=sharing.
[4] Guanghui Wang, Jinze Yu, Xing Zhang, Dayuan Jiang, Yin Song, Tomal Deb, Xuefeng Liu, and Peiyang He. 2025. STED and Consistency Scoring: A Framework for Evaluating LLM Structured Output Reliability. *arXiv preprint arXiv:2512.23712* (2025).