

Deterministic vs. LLM-Controlled Orchestration for Legacy Code Modernization

Naing Oo Lwin

Bucknell University, Lewisburg, PA, USA
Astrio, San Francisco, CA, USA
naing.oo.lwin@bucknell.edu

Rajesh Kumar

Bucknell University, Lewisburg, PA, USA
rajesh.kumar@bucknell.edu

Abstract

Modernizing legacy systems written in COBOL remains challenging due to scarce expertise, complex codebases, and strict correctness requirements. Recent large language model (LLM)-based approaches increasingly adopt agentic designs in which models control multi-step tool orchestration. However, it remains unclear whether LLM-controlled execution improves functional correctness, robustness, or efficiency in structured software engineering workflows. We present a controlled empirical study that isolates the role of execution control by comparing deterministic and LLM-controlled orchestration for COBOL-to-Python modernization. Using a unified framework, we hold models, prompts, tools, and inputs constant, varying only the execution control mechanism. We evaluate both approaches in terms of functional correctness, robustness under repeated execution, and computational efficiency. Our results show that deterministic orchestration achieves comparable functional correctness to LLM-controlled orchestration while improving worst-case robustness and reducing performance variability. It also reduces token consumption by up to 3.5 \times , resulting in substantially lower operational cost. These results indicate that, for structured code modernization tasks, explicit execution control can provide more stable and cost-efficient behavior than fully agentic orchestration without reducing correctness. The code is available at <https://github.com/astrio-ai/atlas>.

CCS Concepts

- **Software and its engineering** \rightarrow **Software maintenance tools**;
- **Computing methodologies** \rightarrow *Artificial intelligence*.

Keywords

LLM Code Modernization, Tool Orchestration, Execution Control, Deterministic Systems, Agentic Systems

ACM Reference Format:

Naing Oo Lwin and Rajesh Kumar. 2026. Deterministic vs. LLM-Controlled Orchestration for Legacy Code Modernization. In *Proceedings of the 3rd ACM International Conference on AI-powered Software (AIware '26)*. ACM, New York, NY, USA, 8 pages. <https://doi.org/XXXXXXX.XXXXXXX>

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.

AIware '26, Montreal, Canada

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/2026/06
<https://doi.org/XXXXXXX.XXXXXXX>

1 Introduction

Legacy software systems written in COBOL remain important in domains such as finance and government, where they support a substantial fraction of long-lived operational infrastructure. At the same time, maintaining and modernizing these systems has become increasingly difficult due to workforce attrition, limited publicly available learning resources, scarce documentation, and the complexity of long-lived codebases [2]. Modernization is therefore necessary not only to sustain existing systems but also to enable integration with contemporary software ecosystems.

Recent work has explored the use of large language models (LLMs) for code translation and modernization, including approaches that integrate tool use and multi-step reasoning [10, 15]. In particular, agentic designs enable LLMs to control multi-step tool orchestration by dynamically selecting actions, ordering operations, and determining when to retry or terminate. While such approaches have shown promise in open-ended settings, it remains unclear whether delegating execution control to LLMs improves functional correctness, robustness, or computational efficiency in structured software engineering workflows.

Structured modernization tasks are typically defined by explicit stages such as transformation, validation, and testing, with intermediate outputs that can be verified automatically. This contrast between structured execution and model-driven control raises a central question: *does LLM-controlled orchestration provide measurable benefits over deterministic execution in such settings?* Existing work has largely focused on overall generation quality or end-task performance, making it difficult to isolate how the orchestration strategy itself affects correctness, robustness under repeated execution, and computational cost when other factors are held constant.

In this paper, we present a controlled empirical study that isolates execution control as the sole experimental variable in LLM-based code modernization. Using ATLAS (Autonomous Transpilation for Legacy Application Systems), we compare deterministic and LLM-controlled orchestration for COBOL-to-Python modernization while holding models, prompts, tools, and inputs constant.

In deterministic orchestration, the sequence of tool invocations, validation steps, and retry policies is predefined and invariant across runs. In contrast, LLM-controlled orchestration delegates these decisions to the model, allowing execution paths to vary dynamically based on intermediate outputs. This distinction enables a direct comparison between fixed execution policies and model-driven control.

Our results show that both approaches achieve comparable functional correctness, but differ substantially in reliability and efficiency. Deterministic orchestration improves worst-case robustness and reduces variability across runs, while also reducing token

consumption by up to 3.5×, leading to significantly lower operational cost.

These findings are specific to structured code modernization tasks with executable validation, but within this scope, they indicate that explicit execution control provides a more stable and cost-efficient alternative to fully agentic orchestration without sacrificing correctness.

Briefly, our key contributions are summarized as follows:

- A controlled empirical framework for LLM-based code modernization that isolates execution control as the sole experimental variable, enabling causal comparison between deterministic and LLM-controlled orchestration and establishing execution control as a primary design factor in structured workflows.
- Empirical results demonstrating that deterministic orchestration achieves comparable functional correctness to LLM-controlled orchestration while providing improved worst-case robustness and reduced performance variability.
- Quantitative analysis showing that deterministic orchestration reduces token consumption by up to 3.5×, yielding substantially lower operational cost without sacrificing correctness.

The remainder of this paper is organized as follows. Section 2 reviews prior work on automated code translation and agentic LLM systems. Section 3 describes the deterministic and LLM-controlled orchestration approaches and the ATLAS framework. Section 4 details the experimental design, dataset, and evaluation metrics. Section 5 presents the empirical results and analysis. Finally, Section 6 concludes and discusses implications for LLM-based software engineering.

2 Background

2.1 Automated Code Translation

Recent work has explored the use of large language models (LLMs) for automated code translation and modernization, framing translation as a mapping from a source program to a semantically equivalent target representation [14]. Transformer-based models and large-scale pretraining have enabled direct, single-shot translation between programming languages, including legacy languages, often without explicit parallel corpora [4, 5].

Recent work also explores code translation and multilingual code generation at scale and studies factors that influence translation outcomes, such as prompt language and prompt design [1, 7, 8, 11]. While these approaches demonstrate promising translation quality, they primarily operate as one-pass generators and do not explicitly model execution control, iterative validation, or workflow structure. As a result, they provide limited insight into how different orchestration strategies affect correctness, robustness under repeated execution, or computational efficiency.

2.2 Agentic and Tool-Orchestrated LLM Systems

More recent systems introduce agentic behavior, enabling LLMs to invoke tools, perform multi-step reasoning, and iteratively refine outputs [3]. In most agentic frameworks, the LLM serves as a central orchestrator that dynamically selects tools, orders operations, and

determines retry or termination behavior at runtime [9, 15]. Several works propose generalized orchestration or workflow engines to standardize and evaluate such agentic behavior [6, 16].

These designs are well-suited to open-ended or exploratory tasks but implicitly treat execution control as a reasoning problem. Prior work has shown that this entanglement of planning and execution can introduce significant stochasticity, where small variations in model outputs lead to divergent execution paths [13]. This complicates reproducibility, stability, and cost predictability.

In contrast, structured software engineering workflows—such as legacy code modernization—are typically defined by fixed execution stages, explicit tool dependencies, and verifiable intermediate states [12]. However, existing agentic systems do not empirically isolate the impact of execution control strategy itself, making it unclear whether delegating orchestration to an LLM provides benefits over deterministic control when other factors are held constant.

Our work addresses this gap by directly comparing deterministic and LLM-controlled orchestration under controlled conditions, isolating execution control as the sole experimental variable.

3 Methodology

3.1 Deterministic Tool Orchestration

ATLAS is implemented as a single-agent system with deterministic tool orchestration. By deterministic orchestration, we mean that for fixed inputs, configuration, and system state, the sequence of tool invocations—including which tools are executed, their order, and the conditions under which they are triggered—is invariant across runs. Any non-determinism in ATLAS is strictly confined to the language model’s code generation and never affects execution control, tool selection, or workflow ordering.

ATLAS enforces a fixed, stage-based execution pipeline implemented in its core message-processing routine. Each run follows the same predefined sequence of stages, including code application, optional persistence, validation, and testing. The execution order is static and does not depend on language model outputs or intermediate results. Conditional branching is governed exclusively by deterministic predicates over observable system state and configuration flags, such as whether files were modified or whether specific validation stages are enabled.

Crucially, the language model does not participate in execution planning. It does not select tools, determine execution order, initiate retries, or control termination. All orchestration decisions are resolved through explicit, deterministic control logic evaluated at runtime. While user interaction may influence whether execution proceeds or terminates, it does not alter the structure or ordering of the pipeline itself.

Within individual stages, ATLAS further enforces determinism through fixed strategy ordering. When multiple mechanisms are available to apply code edits or perform validation, they are attempted sequentially in a predefined and static order until one succeeds. This ensures that fallback behavior is consistent and does not introduce stochastic variation.

The orchestration layer contains no random sampling, probabilistic branching, or stochastic scheduling. Execution depends solely on observable system state and configuration parameters.

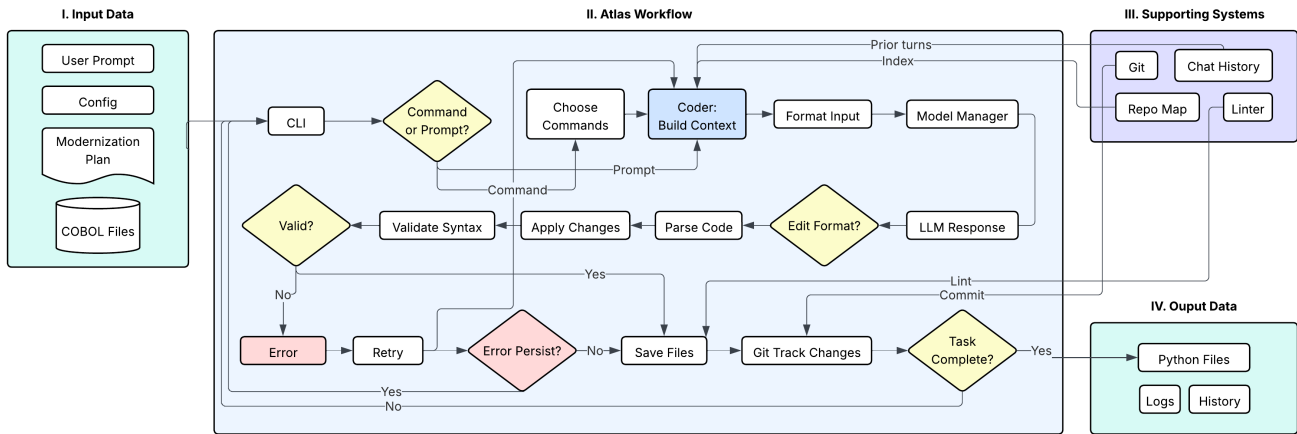


Figure 1: Overview of the ATLAS system architecture and end-to-end COBOL-to-Python modernization workflow.

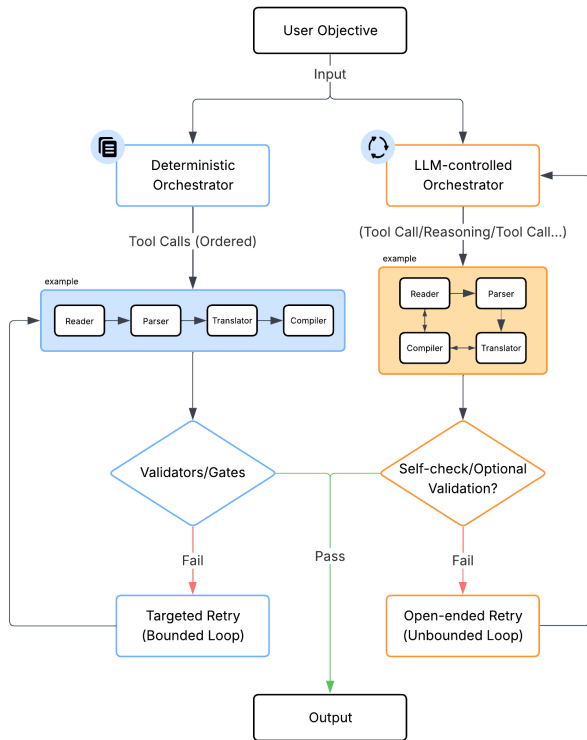


Figure 2: Comparison of deterministic orchestration and LLM-controlled orchestration for COBOL-to-Python modernization. The deterministic lane follows a fixed execution policy with explicit tool/validator stages and bounded retries, while the LLM-controlled lane selects actions adaptively with branching and variable-length tool-use trajectories.

Consequently, for identical inputs and configurations, ATLAS produces identical tool-call traces across runs. This contrasts sharply with agentic systems that delegate execution control to the language model, where tool selection and execution order may vary between runs, leading to increased execution variance and reduced reproducibility.

3.2 LLM-controlled Tool Orchestration

As a baseline, we evaluate an LLM-controlled orchestration paradigm in which the language model is responsible not only for code generation but also for execution control. In this setting, the model dynamically selects which tools to invoke, determines their execution order, chooses repair strategies in response to failures, and decides whether to continue or terminate execution.

Unlike deterministic orchestration, the execution flow in this paradigm is not predefined. Tool invocation and ordering emerge from the language model’s generation process and are influenced by intermediate outputs, prior tool results, and stochastic sampling. Consequently, identical inputs and configurations may lead to different execution traces across runs, even when the same underlying language model and prompts are used.

This design reflects a common assumption in agentic coding systems that execution control can be treated as a reasoning task delegated to the language model. While this flexibility enables adaptive behavior, it also introduces variability in tool usage, execution depth, and recovery behavior. Small variations in model outputs can lead to divergent sequences of operations, complicating reproducibility and stability.

In our experiments, this orchestration strategy serves as a contrastive baseline to deterministic execution. By holding the language model, prompts, tools, and inputs constant, we isolate the effect of LLM-controlled execution on functional correctness, execution stability, and failure behavior in structured legacy code modernization workflows.

4 Experiment & Evaluation Setup

4.1 Experimental Objective

The objective of our experiments is to isolate and quantify the impact of the execution control strategy on outcomes of legacy code modernization. Specifically, we compare deterministic tool orchestration with LLM-controlled orchestration for COBOL-to-Python modernization, holding all other factors constant.

To enable a controlled comparison, both orchestration strategies are implemented within the same ATLAS framework and use identical language models, prompts, tools, configurations, and source programs. The only experimental variable is the mechanism responsible for execution control, including tool selection, ordering, retry behavior, and termination decisions.

Our evaluation focuses on three dimensions: (i) functional correctness, measured by Computational Accuracy and Success Rate; (ii) robustness and stability under repeated execution, measured using tail-risk metrics and tool-call trace consistency; and (iii) computational efficiency, measured in tokens per successful translation. This setup allows us to attribute observed differences in correctness, robustness, and cost directly to the orchestration strategy rather than to model capability or prompt design.

4.2 Controlled Experimental Variables

To ensure a fair and controlled comparison between deterministic and LLM-controlled orchestration, we hold all experimental variables constant except for the execution control mechanism.

Language Models. All experiments are conducted using the same set of underlying language models. For a given experiment, the model architecture, version, decoding parameters (e.g., temperature), and random seed initialization protocol are identical across orchestration strategies. This ensures that any observed differences are not attributable to model capability or sampling configuration.

Prompts and Instructions. Both orchestration strategies use identical system prompts, task instructions, and formatting constraints. The language model receives the same contextual information and code inputs in both settings, with no orchestration-specific prompt engineering.

Tools. Both orchestration strategies have access to the same tool set with identical interfaces and permissions. The available tools include:

- `read_file`: read file contents,
- `write_file`: write or modify file contents.
- `list_files`: enumerate repository files,
- `web_scrape`: retrieve external web content,
- `run_command`: execute shell commands,
- `git`: execute version control operations,

No tools are added, removed, or modified between orchestration strategies.

Inputs and Configuration. The same source programs, test inputs, execution environment, timeout limits, and configuration flags are used across all runs. Validation and testing stages are enabled or disabled identically for both orchestration modes.

Isolation of Execution Control. The sole experimental variable is the mechanism responsible for execution control, including tool selection, invocation order, retry behavior, and termination decisions. By controlling all other factors, we ensure that differences in correctness, robustness, execution stability, and computational cost can be attributed directly to the orchestration strategy.

4.3 Dataset and Test Harness

We evaluate ATLAS using the NIST COBOL85 Test Suite, a standardized and widely used benchmark designed to validate conformance to the COBOL 85 language specification. The suite consists of several hundred self-contained COBOL programs, each paired with a structured set of test cases that exercise specific language features and execution behaviors. The programs in our evaluation set have an average length of 1,200 Lines of Code (LOC), representing non-trivial logic segments common in legacy enterprise environments.

These programs collectively cover a broad range of core COBOL constructs, including arithmetic and control flow (NC), copy and call semantics (SM, IC), intrinsic functions (IF), sorting (ST), communication (CM), debugging and report (DB, RW), and multiple file I/O modalities such as sequential, indexed, and relative access (SQ, IX, RL).

A key advantage of the NIST COBOL85 suite is that it provides executable test harnesses with well-defined expected outcomes, enabling automated and reproducible correctness evaluation. For each COBOL program, the suite specifies a collection of test cases whose outcomes can be validated deterministically, making it well suited for measuring functional equivalence after translation.

In our experiments, each COBOL source file is translated to Python and evaluated by executing the translated program against the same test inputs used for the original COBOL implementation. Computational Accuracy is computed by comparing the observable outputs of the translated program to those produced by the reference COBOL execution. Success Rate additionally requires that the translated program executes without runtime errors and passes all associated test cases.

By leveraging an established, executable test suite with known semantics, our evaluation avoids reliance on heuristic similarity metrics and enables direct measurement of functional correctness, robustness under repeated execution, and execution stability across orchestration strategies.

4.4 Evaluation Metrics

We evaluate deterministic and LLM-controlled orchestration using complementary metrics that capture functional correctness, reliability, robustness to stochasticity, and execution stability. All metrics are computed under controlled conditions, holding the language model, prompts, tools, and inputs fixed.

Computational Accuracy (CA). Computational Accuracy measures functional equivalence between the generated program and a reference implementation over a set of test inputs:

$$CA(p) = \frac{1}{|T_p|} \sum_{t \in T_p} \mathbf{1}[\text{exec}(y_p, t) = \text{exec}(y_p^*, t)], \quad (1)$$

Table 1: Comparison of deterministic and LLM-controlled tool orchestration across models on COBOL-to-Python modernization. Metrics are averaged across programs and repeated runs. Higher is better for all metrics.

Model	Orchestration	CA \uparrow	SR \uparrow	P5-CA \uparrow	CVaR _{0.1} \uparrow
Claude-Sonnet-4-5	Deterministic	0.966	0.902	0.959	0.953
	LLM-controlled	0.964	0.918	0.956	0.949
GPT-5.1-Codex-Max	Deterministic	0.969	0.910	0.962	0.956
	LLM-controlled	0.964	0.937	0.958	0.951
Grok-Code-Fast-1	Deterministic	0.961	0.872	0.951	0.943
	LLM-controlled	0.958	0.906	0.941	0.934

where p indexes a program, T_p denotes its test inputs, y_p is the generated output, and y_p^* is the reference program. CA is averaged across programs to obtain mean correctness.

Success Rate (SR). Success Rate measures the run-to-run reliability of an orchestration strategy under stochastic execution. For each source program p , the translation pipeline is executed N independent times using different random seeds. A run is considered successful if the generated program (i) compiles or executes without error and (ii) passes all evaluation test cases.

The success rate for program p is defined as:

$$SR(p) = \frac{1}{N} \sum_{i=1}^N \mathbf{1}[\text{run}_i(p) \text{ is successful}], \quad (2)$$

where $\mathbf{1}[\cdot]$ denotes the indicator function.

Aggregating across all programs P , we report the mean success rate:

$$SR = \frac{1}{|P|} \sum_{p \in P} SR(p). \quad (3)$$

Worst-Case Robustness. To characterize robustness under stochastic variation, we repeat each program N times with independent random seeds and compute tail-risk metrics over the resulting CA values $\{CA_p^{(i)}\}_{i=1}^N$.

We report the 5th percentile CA:

$$P5\text{-CA}(p) = \text{Percentile}_5(\{CA_p^{(i)}\}), \quad (4)$$

which captures typical worst-case behavior while remaining robust to isolated outliers.

Conditional Value at Risk (CVaR) measures failure severity:

$$CVaR_\alpha(p) = \mathbb{E}\left[CA_p^{(i)} \mid CA_p^{(i)} \leq P_\alpha\right], \quad (5)$$

where α is set to 0.1 in our experiments. CVaR quantifies the average correctness among the worst-performing fraction of runs.

5 Results and Discussion

5.1 Main Results and Key Takeaways

Across all evaluated models and orchestration strategies, the results reveal a consistent tradeoff between correctness robustness and execution success frequency. Deterministic orchestration achieves higher computational accuracy and stronger worst-case robustness, while LLM-controlled orchestration attains higher success

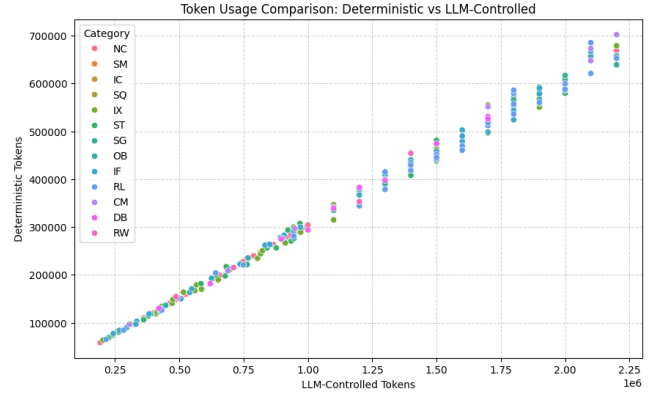


Figure 3: Token usage per successful translation for deterministic vs. LLM-controlled orchestration.

rates across repeated runs. Specifically, deterministic pipelines consistently produce higher mean Computational Accuracy (CA), improved 5th-percentile accuracy (P5-CA), and higher Conditional Value at Risk (CVaR), indicating reduced tail-risk failures and more stable functional behavior.

In contrast, LLM-controlled orchestration demonstrates higher Success Rate (SR), suggesting that adaptive agentic decision-making increases the likelihood that a runnable program is produced in any given execution. Importantly, both orchestration strategies achieve comparable overall capability when evaluated using a single finalized translation per program, indicating that orchestration primarily influences reliability characteristics rather than fundamental translation ability.

5.2 Evaluation Metrics and Performance

Table 1 reports aggregated performance across models using Computational Accuracy (CA), Success Rate (SR), and tail-risk robustness metrics (P5-CA and CVaR_{0.1}). Across all evaluated models, deterministic orchestration achieves the highest CA values, indicating stronger functional equivalence to reference implementations. The improvement is accompanied by consistently higher P5-CA and CVaR scores, demonstrating that deterministic workflows reduce degradation in worst-case executions and mitigate severe failure modes.

Conversely, LLM-controlled orchestration yields higher SR across all models. Because SR measures reliability over repeated stochastic executions, this result indicates that agentic orchestration more frequently produces compilable and test-passing programs across independent runs. However, lower P5-CA and CVaR values indicate that when failures occur, they tend to be more severe or less predictable than under deterministic control.

Taken together, these results show that orchestration strategy primarily shifts the distribution of outcomes: deterministic orchestration compresses variance and improves worst-case guarantees, whereas LLM-controlled orchestration increases execution success likelihood while introducing heavier performance tails.

5.3 Efficiency and Cost Analysis

The most significant divergence between the two orchestration strategies is observed in computational efficiency and operational expenditure. While both approaches achieve comparable functional outcomes, the resource requirements for agentic, LLM-driven control are substantially higher than those for fixed execution pipelines.

Token Consumption and Throughput. Deterministic orchestration achieves a drastic reduction in token consumption across all evaluated COBOL categories compared to the LLM-controlled baseline. As illustrated in Figure 3, LLM-controlled orchestration often requires between 1.75 million and 2.25 million tokens for complex modules such as Numeric (NC) and Sequential I/O (SQ). In contrast, the deterministic pipeline completes these same tasks within a significantly lower range of 400,000 to 700,000 tokens. This represents a reduction in token consumption by up to an order of magnitude for highly structured modernization tasks. The clustering of data points along the lower y-axis in Figure 3 demonstrates that deterministic control prevents “agentic drift” or unbounded reasoning loops common in LLM-controlled systems, where small variations in model outputs can lead to divergent and inefficient execution paths.

Operational Cost Impact. The financial implications of these efficiency gains are summarized in Figure 4, which estimates the total cost per successful translation based on token usage. The Sequential I/O (SQ) category exhibits the highest cost disparity, where LLM-controlled orchestration costs exceed \$140, while deterministic orchestration achieves the same result for approximately \$40. Similar trends are visible in the NC and IC categories, where agentic control significantly inflates costs due to repeated reasoning steps and open-ended retry mechanisms. Across all modules, the “planning tax” associated with delegating workflow logic to the LLM results in a consistent cost overhead without a proportional increase in functional correctness. These results suggest that for enterprise-scale modernization, embedding LLMs within deterministic frameworks is essential for maintaining economic viability and cost predictability.

5.4 Test-Suite Summary (COBOL Baseline vs. Python Translation by Deterministic and LLM-Controlled Orchestration)

Table 2 summarizes outcomes from the NIST COBOL85 test harness, comparing the original COBOL programs with Python translations generated under deterministic and LLM-controlled orchestration.

At the aggregate level, both orchestration strategies produce nearly identical execution statistics. The number of programs executed, total errors, and inspected cases remain unchanged across settings, indicating that both approaches reach comparable levels of overall functional coverage. This suggests that orchestration choice does not materially affect whether translations can ultimately be produced for a given program.

Differences emerge primarily in pass and failure counts. Deterministic orchestration yields slightly higher pass totals and fewer failures compared to LLM-controlled orchestration, consistent with the higher CA and robustness metrics observed in Table 1. However,

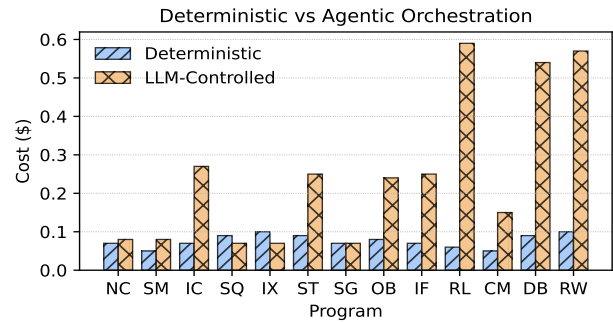


Figure 4: Cost comparison between deterministic and LLM-controlled orchestration across COBOL benchmark programs. Each pair of bars shows the cost per successful translation for a given program under identical models, prompts, tools, and inputs, with only the execution control mechanism varied. Deterministic orchestration consistently achieves lower or comparable costs, whereas LLM-controlled orchestration incurs substantially higher costs in most cases due to increased token usage and adaptive execution paths.

these differences are modest, reinforcing the conclusion that both strategies achieve similar baseline translation capability.

Overall, the test-suite results confirm that orchestration affects consistency and robustness rather than raw feasibility: both approaches successfully translate the same set of programs, while deterministic orchestration provides marginally stronger correctness guarantees in final outputs.

5.5 Limitations

Although our experiments provide a controlled comparison between deterministic and LLM-controlled orchestration, several limitations should be considered when interpreting the results. First, our evaluation focuses exclusively on COBOL-to-Python modernization. While this task represents an important and realistic legacy migration scenario, the findings may not directly generalize to other programming languages, modernization targets, or heterogeneous enterprise systems involving mixed-language dependencies. Structured legacy workflows such as COBOL translation exhibit well-defined execution stages and validation procedures, which may amplify the benefits of deterministic orchestration compared to more exploratory software engineering tasks.

Second, the evaluation relies on the NIST COBOL85 test suite, which, although widely used and executable, does not fully capture the complexity of production enterprise environments. Real-world COBOL systems frequently interact with external infrastructure such as JCL pipelines, CICS transactions, VSAM data stores, database systems, and vendor-specific dialect extensions. These operational dependencies introduce integration challenges and environmental constraints that are not represented in standardized benchmark programs. As a result, measured correctness and robustness may overestimate performance relative to large-scale industrial modernization settings.

Third, correctness measurements depend on the completeness and quality of available test oracles. Computational Accuracy and

Table 2: NIST COBOL85 test-harness summary for the original COBOL baseline and for Python translations produced under deterministic vs. LLM-controlled orchestration. Counts are aggregated by module.

Module	COBOL Baseline								Python (Deterministic)								Python (LLM-controlled)							
	Programs	Executed	Error	Pass	Fail	Deleted	Inspect	Total	Programs	Executed	Error	Pass	Fail	Deleted	Inspect	Total	Programs	Executed	Error	Pass	Fail	Deleted	Inspect	Total
NC	90	90	0	4352	0	6	11	4369	90	90	0	4390	0	6	11	4407	90	90	0	4375	0	6	11	4392
SM	15	15	0	290	0	3	1	294	15	15	0	295	0	3	1	299	15	15	0	292	0	3	1	296
IC	13	13	0	97	0	0	0	97	13	13	0	99	0	0	0	99	13	13	0	98	0	0	0	98
SQ	81	81	0	512	0	6	81	599	81	81	0	525	0	6	81	612	81	81	0	520	0	6	81	607
IX	39	39	0	507	0	1	0	508	39	39	0	516	0	1	0	517	39	39	0	513	0	1	0	514
ST	39	39	0	278	0	0	0	278	39	39	0	283	0	0	0	283	39	39	0	281	0	0	0	281
SG	5	5	0	193	0	0	0	193	5	5	0	196	0	0	0	196	5	5	0	195	0	0	0	195
OB	5	5	0	16	0	0	0	16	5	5	0	17	0	0	0	17	5	5	0	16	0	0	0	16
IF	42	42	0	732	0	0	0	732	42	42	0	744	0	0	0	744	42	42	0	738	0	0	0	738
RL	32	32	0	1827	0	5	0	1832	32	32	0	1860	0	5	0	1865	32	32	0	1850	0	5	0	1855
CM	7	0	7	0	0	0	0	0	7	0	7	0	0	0	0	0	7	0	7	0	0	0	0	0
DB	10	2	1	42	274	72	0	388	10	2	1	115	201	72	0	388	10	2	1	108	208	72	0	388
RW	4	4	0	40	0	0	0	40	4	4	0	44	0	0	0	44	4	4	0	43	0	0	0	43
Total	382	367	8	8886	274	93	93	9346	382	367	8	9006	154	93	93	9346	382	367	8	8986	174	93	93	9346

Success Rate evaluate behavioral equivalence only with respect to the provided test cases. Programs with incomplete coverage may appear correct despite latent semantic errors that are not exercised by the test suite. While the use of an executable benchmark improves objectivity compared to similarity-based metrics, it does not eliminate the broader challenge of validating semantic equivalence in complex software systems.

Another limitation arises from assumptions about tooling and workflow structure. Deterministic orchestration benefits from clearly defined stages, explicit validators, and stable tool interfaces. In environments where validation mechanisms are weak, poorly specified, or unavailable, deterministic pipelines may offer fewer advantages relative to adaptive agentic approaches. Consequently, the observed robustness improvements should be interpreted in the context of workflows that support structured verification.

In addition, although we control prompts, models, and decoding parameters to isolate execution control as the sole experimental variable, the results remain dependent on the selected language models and prompting configuration. Different prompting strategies, model architectures, or sampling regimes could alter the relative performance characteristics of orchestration strategies. Our conclusions therefore characterize behavior under controlled conditions rather than universal properties of all LLM-based systems.

Finally, while we provide a quantitative analysis of computational efficiency and operational expenditure in Figures 3 and 4, these results are primarily based on token consumption and API-based pricing. While token usage provides a consistent, model-agnostic measure of resource requirements, it does not fully capture wall-clock latency, local infrastructure overhead, or the total engineering cost of system-level deployment in production environments. Furthermore, our cost estimates are based on current API pricing models, which may vary across providers and over time. Future evaluations that incorporate comprehensive runtime performance and longitudinal operational resource usage would provide a more holistic understanding of the practical economic trade-offs between deterministic and agentic systems.

5.6 Discussion

The results indicate that execution control primarily affects the distribution of outcomes rather than the overall capability of LLM-based code modernization systems. Deterministic orchestration

consistently improves worst-case correctness and reduces performance variability, while LLM-controlled orchestration increases the likelihood of producing a runnable solution in any given run. This pattern suggests that the choice of orchestration strategy governs how uncertainty introduced by stochastic language models propagates through multi-step workflows. In structured modernization tasks with fixed stages and verifiable intermediate states, constraining execution appears to bound error accumulation and produce more predictable behavior without materially limiting achievable correctness.

These findings also help clarify when agentic orchestration may or may not be advantageous. Tasks characterized by clearly defined transformation pipelines, strong validation signals, and deterministic dependencies benefit from explicit execution policies that separate planning from generation. In contrast, open-ended software engineering activities—such as exploratory refactoring, requirements discovery, or incomplete specifications—may still benefit from adaptive LLM-driven control, where flexibility outweighs strict reproducibility. Rather than viewing deterministic and agentic approaches as competing paradigms, the results suggest they occupy different regions of the software engineering design space.

More broadly, the study supports an architectural perspective in which language models function most effectively as generative or interpretive components embedded within structured execution frameworks. Treating execution control itself as a stochastic reasoning problem introduces variability that may not improve correctness in workflows already governed by explicit rules and validators. This observation motivates future hybrid designs that combine deterministic default execution with carefully bounded LLM intervention, enabling adaptive behavior while preserving reproducibility, robustness, and predictable operational overhead.

6 Conclusion

Legacy code modernization remains a critical challenge as organizations seek to migrate long-lived systems while preserving functional correctness and operational reliability. While recent LLM-based approaches increasingly adopt agentic designs that delegate execution control to language models, the architectural and economic impacts of these orchestration strategies have remained largely unexamined. In this work, we conducted a controlled

empirical study comparing deterministic and LLM-controlled tool orchestration for COBOL-to-Python modernization.

Our results indicate that orchestration strategy influences both reliability and efficiency. Deterministic orchestration improves worst-case correctness and reduces outcome variability, resulting in stronger robustness guarantees. In contrast, LLM-controlled orchestration attains higher success rates across stochastic runs but does so at substantially higher computational cost. We observe a clear “planning tax” in agentic workflows: LLM-driven control consumes several times more tokens and incurs markedly higher operational cost than structured pipelines, while yielding only marginal gains in final translation accuracy.

These findings suggest that for structured software engineering workflows with fixed stages and strong validation signals, language models are most effective when embedded within explicit execution policies rather than tasked with managing workflow logic. The stark disparity in resource consumption indicates that deterministic structure is not only a matter of reliability but a prerequisite for the economic viability of large-scale enterprise migrations.

Rather than replacing agentic approaches, our results motivate a clearer separation between generation and execution. We point toward hybrid architectures that leverage the generative power of LLMs within deterministic frameworks to balance adaptive behavior with reproducibility, robustness, and predictable overhead. Future work will extend this analysis to additional modernization domains and larger industrial systems to further refine the boundary between structured execution and autonomous reasoning.

Acknowledgments

We would like to thank Alan Marchiori, Todd Schmid, and Hung Pham for their valuable feedback.

References

- [1] Aamer Aljaghami, Mohammed Banabila, Musab Alshehri, Mohammed Kabini, and Mohammad D. Alahmadi. 2025. Evaluating Large Language Models for Code Translation: Effects of Prompt Language and Prompt Design. arXiv:2509.12973 [cs.SE] <https://arxiv.org/abs/2509.12973>
- [2] Agnieszka Ciborowska, Aleksandar Chakarov, and Rahul Pandita. 2021. Contemporary COBOL: Developers’ Perspectives on Defects and Defect Location. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 227–238. doi:10.1109/icsme52107.2021.00027
- [3] Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V. Chawla, Olaf Wiest, and Xiangliang Zhang. 2024. Large Language Model based Multi-Agents: A Survey of Progress and Challenges. arXiv:2402.01680 [cs.CL] <https://arxiv.org/abs/2402.01680>
- [4] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? arXiv:2310.06770 [cs.CL] <https://arxiv.org/abs/2310.06770>
- [5] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised Translation of Programming Languages. arXiv:2006.03511 [cs.CL] <https://arxiv.org/abs/2006.03511>
- [6] Maria Emilia Mazzolenis and Ruirui Zhang. 2025. Agent WARPP: Workflow Adherence via Runtime Parallel Personalization. arXiv:2507.19543 [cs.AI] <https://arxiv.org/abs/2507.19543>
- [7] Gabriel Orlanski, Kefan Xiao, Xavier Garcia, Jeffrey Hui, Joshua Howland, Jonathan Malmaud, Jacob Austin, Rishabh Singh, and Michele Catasta. 2023. Measuring The Impact Of Programming Language Distribution. arXiv:2302.01973 [cs.SE] <https://arxiv.org/abs/2302.01973>
- [8] Jialing Pan, Adrien Sadé, Jin Kim, Eric Soriano, Guillem Sole, and Sylvain Flamant. 2023. SteloCoder: a Decoder-Only LLM for Multi-Language to Python Code Translation. arXiv:2310.15539 [cs.CL] <https://arxiv.org/abs/2310.15539>
- [9] Libin Qiu, Yuhang Ye, Zhirong Gao, Xide Zou, Junfu Chen, Ziming Gui, Weizhi Huang, Xiaobo Xue, Wenkai Qiu, and Kun Zhao. 2025. Blueprint First, Model Second: A Framework for Deterministic LLM Workflow. arXiv:2508.02721 [cs.SE] <https://arxiv.org/abs/2508.02721>
- [10] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. arXiv:2302.04761 [cs.CL] <https://arxiv.org/abs/2302.04761>
- [11] Qingxiao Tao, Tingrui Yu, Xiaodong Gu, and Beijun Shen. 2024. Unraveling the Potential of Large Language Models in Code Translation: How Far Are We? arXiv:2410.09812 [cs.SE] <https://arxiv.org/abs/2410.09812>
- [12] Michele Tufano, Anisha Agarwal, Jinu Jang, Roshanak Zilouchian Moghaddam, and Neel Sundaresan. 2024. AutoDev: Automated AI-Driven Development. arXiv:2403.08299 [cs.SE] <https://arxiv.org/abs/2403.08299>
- [13] Karthik Valmeekam, Sarath Sreedharan, Matthew Marquez, Alberto Olmo, and Subbarao Kambhampati. 2023. On the Planning Abilities of Large Language Models (A Critical Investigation with a Proposed Benchmark). arXiv:2302.06706 [cs.AI] <https://arxiv.org/abs/2302.06706>
- [14] Zhen Yang, Fang Liu, Zhongxing Yu, Jacky Wai Keung, Jia Li, Shuo Liu, Yifan Hong, Xiaoxue Ma, Zhi Jin, and Ge Li. 2024. Exploring and Unleashing the Power of Large Language Models in Automated Code Translation. arXiv:2404.14646 [cs.SE] <https://arxiv.org/abs/2404.14646>
- [15] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. arXiv:2210.03629 [cs.CL] <https://arxiv.org/abs/2210.03629>
- [16] Qianqian Zhang, Jijia Liao, Heting Ying, Yibo Ma, Haozhan Shen, Jingcheng Li, Peng Liu, Lu Zhang, Chunxin Fang, Kyusong Lee, Ruochen Xu, and Tiancheng Zhao. 2025. Unifying Language Agent Algorithms with Graph-based Orchestration Engine for Reproducible Agent Research. arXiv:2505.24354 [cs.CL] <https://arxiv.org/abs/2505.24354>

Received 15 February 2026