

A Factorial Comparison of LLM-Based Multi-Agent Architectures for Automated Python Refactoring

Alexander Puma Pucho ✉ 

Institute of Computing, University of Campinas, SP, Brazil

Bruno B. P. Cafeo ✉ 

Institute of Computing, University of Campinas, SP, Brazil

Abstract

Background. LLM-based multi-agent systems (LLM-MAS) are increasingly applied to software engineering (SE) tasks. However, most systems are evaluated end-to-end, entangling architectural choices with model selection, prompting, and task-specific tooling. As a consequence, the effects of individual architectural factors remain empirically underexplored.

Aims. We investigate how Communication Structure (Hierarchical vs. Centralized) and Coordination Strategy (Dynamic vs. Static) influence three outcome dimensions in LLM-MAS for Python refactoring: output generation and validity; operational cost and execution efficiency; and structural transformation. We also examine whether the two factors interact.

Method. We conduct a controlled 2×2 factorial experiment under identical model, dataset, orchestration, prompting, and tool configurations, evaluated on 2,000 real-world Python refactoring instances drawn from open-source machine learning (ML) repositories. We measure output generation and validity through file generation, syntactic validity, and similarity-based acceptance; operational cost and execution efficiency through execution time, LLM calls, and token usage; and structural transformation through static code metrics.

Results. Coordination Strategy is the dominant architectural factor: Static configurations achieve substantially higher output validity and lower operational cost across all token-based and call-based measures. Communication Structure exhibits a heterogeneous effect, strongly influencing call-based behavior but having limited impact on token consumption. The two factors interact significantly on all cost measures, with the strongest interaction observed on LLM calls. Among the evaluated configurations, Hierarchical+Static consistently provides the best tradeoff between validity and cost, whereas Centralized+Dynamic fails to produce acceptable output for a substantial fraction of instances.

Conclusions. Coordination Strategy exerts more consistent effects than Communication Structure on both output validity and operational cost, while the interaction between the two factors is central to explaining architectural behavior. The study demonstrates how controlled factorial designs can isolate architectural effects in LLM-MAS research, and it provides empirical guidance for designing cost-efficient architectures with strong output validity for SE tasks.

2012 ACM Subject Classification Software and its engineering → Maintaining software

Keywords and phrases Architectural Comparison, Code Refactoring, LLM-based Multi-Agent Systems, Software Maintenance Automation, Multi-Agent Architectures

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.37

Category Technical Track Paper

Supplementary Material *Software (Source code, dataset, and analysis scripts):* <https://doi.org/10.5281/zenodo.20042886>

Funding This work was supported by FAEPEX under grant 3202/26, by the R&D projects with Shell Brasil Petróleo Ltda, registered as ANP number 21373-6 and 24282-6, both funded by Shell Brazil Technology, and by FAPESP grant number 2025/09227-5.



© Alexander Puma Pucho and Bruno B. P. Cafeo;
licensed under Creative Commons License CC-BY 4.0

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).

Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón;

Article No. 37; pp. 37:1–37:21



Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

LLM-MAS are increasingly adopted for SE tasks via role specialization, structured coordination, and tool grounding [9, 18]. Recent code-oriented systems vary substantially in internal organization: hierarchical arrangements of specialized agents [14, 19, 44], centralized coordination by a manager agent [2], and static or dynamic tool invocation [13, 28, 34, 43]. Yet the literature has advanced faster in proposing such systems than in understanding them: architectures are typically evaluated end-to-end, entangling architectural choices with model selection, prompting, and task-specific tooling [9, 18]. Such evaluations reveal whether a system succeeds or fails, but not which architectural decisions drive that outcome.

Isolating architectural effects requires a domain where (i) the task decomposes into subtasks suited to role specialization, (ii) outputs admit structured evaluation, and (iii) real-world instances are available at scale. Automated refactoring meets all three: it decomposes into analysis, localization, and transformation subtasks that map onto specialized agent roles [14, 44]; its outputs can be assessed via syntactic, structural, and similarity-based criteria; and labeled instances abound in open-source repositories [25]. Refactoring improves the internal structure of code without altering its observable behavior [5, 22], providing a repeatable workload for controlled comparison. LLM-MAS have begun to tackle this problem, yet existing systems cover only a narrow slice of the architectural design space [14, 34, 43, 44].

A controlled architectural comparison requires factors that are both taxonomically grounded and experimentally manipulable. Recent surveys describe a broad LLM-MAS design space [7, 9, 17, 18, 39, 45], and within the refactoring literature the most clearly instantiated architectural distinctions concern how agents communicate and how execution is coordinated [14, 34, 43, 44]. We therefore focus on *Communication Structure* – how control and information flow are distributed among agents – and *Coordination Strategy* – whether task execution and tool invocation are prescribed in advance or determined adaptively at runtime. No prior refactoring study isolates their effects in a crossed design, leaving their individual and interaction effects empirically uncharacterized.

We address this gap with a controlled 2×2 factorial experiment manipulating *Communication Structure* [Hierarchical (H) vs. Centralized (C)] and *Coordination Strategy* [Dynamic (Dyn) vs. Static (Sta)], yielding four configurations: MAS-1 (H+Dyn), MAS-2 (H+Sta), MAS-3 (C+Dyn), and MAS-4 (C+Sta). All share identical model, dataset, orchestration, prompting, and tools, with each architecture processing every input instance, enabling estimation of main effects and their interaction while controlling for input-level variability.

The experiment uses $N = 2,000$ real-world file-level refactoring instances drawn from open-source Python ML repositories [25], each with aligned pre- and post-refactoring snapshots. Outcomes are analyzed across three planes: *output generation and validity* (file generation, syntactic validity, similarity-based acceptance), *operational cost and execution efficiency* (execution time, LLM calls, token consumption), and *structural transformation* (static code metrics). Given non-normal distributions, we use non-parametric procedures throughout: Pearson’s χ^2 and Fisher’s exact tests for generation and validity; ART-ANOVA for operational cost; and Friedman, Wilcoxon signed-rank, Kruskal–Wallis, and Dunn tests for structural transformation and efficiency; all with Holm–Bonferroni correction and effect sizes reported.

This experimental setting gives rise to three research questions.

- RQ1** *How do the four architectural configurations differ in file generation success, syntactic validity, and similarity-based output acceptance?*
- RQ2** *What are the main effects of Communication Structure and Coordination Strategy on operational cost, and do these two architectural factors interact?*
- RQ3** *How do the four architectural configurations differ in (a) the structural transformations they apply to the input code and (b) their execution efficiency?*

Taken together, this design yields four main contributions. **Experimentally**, it provides a controlled 2×2 factorial study isolating Communication Structure and Coordination Strategy as architectural factors in LLM-MAS for automated Python refactoring under common model, prompting, orchestration, tool, and dataset conditions. **Empirically**, it produces evidence on the main and interaction effects of these factors on output generation and validity, operational cost, and structural transformation. **Methodologically**, it demonstrates how controlled factorial designs can isolate architectural effects in LLM-MAS research, exposing interaction effects that end-to-end benchmarking conflates. **As an artifact**, it releases a replication package comprising 2,000 refactoring instances, four implemented LLM-MAS, execution logs, and analysis scripts.

2 System Architectures

2.1 Architectural Design Factors

We characterize the architectures along two manipulated factors governing execution control in LLM-MAS: *Communication Structure* and *Coordination Strategy*. Both are grounded in recent LLM-MAS surveys [7, 9, 17, 18, 39, 45] and were selected because they admit direct operationalization in automated refactoring under identical experimental conditions.

Communication Structure. Communication Structure concerns how control and information flow are organized among agents, commonly distinguished as decentralized, hierarchical, or centralized [7, 9, 17, 18, 39, 45]. We focus on the hierarchical and centralized structures, since both admit a clean crossing with Coordination Strategy under a fixed agent set, whereas decentralized communication would also require modifying agent count and autonomy, and is thus outside the scope of this experiment.

- **Hierarchical:** agents are organized into sequential functional tiers, and control emerges from tier-to-tier sequencing rather than a dedicated coordinator. Outputs from one tier are passed directly to the next. We operationalize this through three tiers: *analysis* (file reading and static code inspection), *localization* (refactoring suggestion and evaluation), and *resolution* (code transformation and output generation).
- **Centralized:** a dedicated coordinator orchestrates the remaining agents through explicit control artifact specifying execution order. We operationalize this through a *Planning Agent* that decomposes the request into an ordered step list and dispatches work.

Coordination Strategy. Coordination Strategy concerns whether execution and tool use are fixed in advance or determined adaptively at runtime [39]. We operationalize it as follows:

- **Dynamic:** tool invocation is delegated to *ReAct*-style agents [46], which interleave reasoning and acting and decide at run time whether and how to invoke their tools.
- **Static:** tool invocation is prescribed by the workflow or planning artifact. Tools are called when and where the architecture specifies, independently of intermediate observations.

2.2 Controlled Invariants and Shared Setup

To isolate architectural effects, all four configurations share the same model, orchestration substrate, tool set, agent responsibilities, and retry bounds. Only Communication Structure and Coordination Strategy vary across architectures.

Model and Orchestration. All agents use the *open-mixtral-8x22b*¹ model through the same public API, with default inference parameters and no fine-tuning beyond role-specific instructions. All workflows are implemented in *LangGraph*², which represents execution as explicit stateful graphs. Holding both the model and orchestration framework constant prevents architectural comparisons from being confounded by model or framework variation.

External Tools. Each architecture has access to the same three tools over source-code and file artifacts: (i) ANALYZE CODE, implemented with *Radon*³ and *Lizard*⁴; (ii) READ FILE; and (iii) WRITE FILE. The difference across architectures is therefore not *which* tools are available, but *where* tool invocation is controlled: by ReAct agents in Dynamic configurations, and by the workflow/plan in Static configurations.

Execution Types and Agent Roles. Two execution types are used throughout the study. *Prompt-based agents* produce outputs through a single LLM inference and do not invoke tools directly. *ReAct agents* [46] interleave reasoning and tool use, selecting actions from intermediate observations. Architectures are constructed from a fixed set of roles with stable responsibilities across configurations. The *Suggestion Agent*, *Evaluation Agent*, *Refactoring Agent*, and *Planning Agent* are prompt-based: the Suggestion Agent generates candidate refactoring suggestions from the input code and analysis results; the Evaluation Agent assesses those candidates and provides feedback for iterative refinement; the Refactoring Agent applies validated decisions to produce the modified source file; and the Planning Agent, present only in centralized configurations (MAS-3, MAS-4), decomposes the refactoring request into an ordered step list specifying execution order and responsibility assignment. The *File System Agent* and *Code Analysis Agent*, exclusive to Dynamic configurations (MAS-1, MAS-3), are ReAct agents encapsulating file I/O operations (READ FILE, WRITE FILE) and static source code analysis (ANALYZE CODE), respectively.

Shared Iterative Mechanisms. All architectures include two bounded loops applied identically across configurations. In the *Suggestion-Evaluation loop*, the Suggestion Agent is re-invoked with corrective feedback when a proposed refactoring is rejected. In the *syntactic-validity loop*, the Refactoring Agent is re-invoked with AST parser errors when the generated file fails to parse. Both loops are capped at three iterations; once the budget is exhausted, the last generated result is forwarded.

2.3 MAS-1: Hierarchical + Dynamic

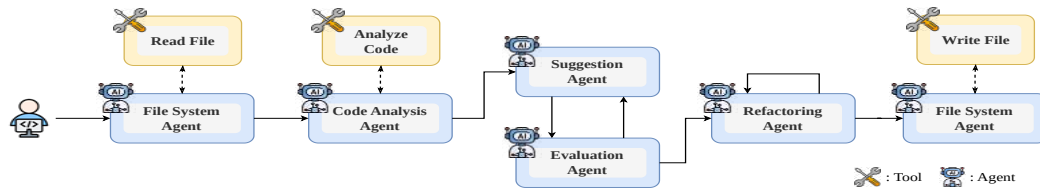
MAS-1 instantiates the *Hierarchical* communication structure with *Dynamic* coordination. As shown in Figure 1, execution is organized into three functional tiers. The *analysis* tier consists of the File System Agent, which reads the input file through READ FILE, and the Code Analysis Agent, which invokes ANALYZE CODE. The *localization* tier contains the Suggestion and Evaluation agents, which iteratively refine candidate refactorings through the shared Suggestion-Evaluation loop. The *resolution* tier contains the Refactoring Agent, which applies the accepted transformation under the shared syntactic-validity loop, after which the File System Agent writes the final output through WRITE FILE. MAS-1 therefore separates functionality by tier while leaving tool use to agent-level decisions at run time.

¹ <https://mistral.ai/news/mixtral-8x22b>

² <https://docs.langchain.com/oss/python/langgraph/overview>

³ <https://radon.readthedocs.io/>

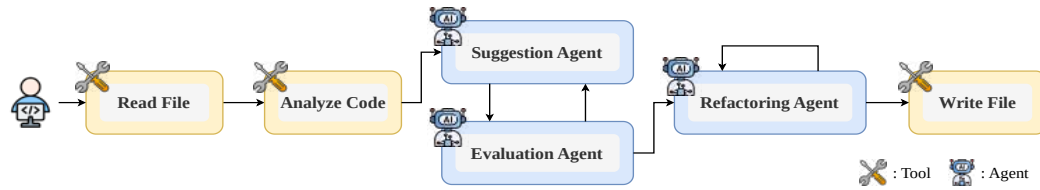
⁴ <https://github.com/terryyin/lizard>



■ **Figure 1** MAS-1 (H+Dyn). A tier-based workflow is combined with ReAct tool invocation. The File System and Code Analysis agents decide at run time when and how to invoke their tools.

2.4 MAS-2: Hierarchical + Static

MAS-2 preserves the same *Hierarchical* tier structure as MAS-1 but switches from *Dynamic* to *Static* coordination. As shown in Figure 2, the analysis tier is still followed by localization and resolution, but READ FILE, ANALYZE CODE, and WRITE FILE are invoked directly by the workflow rather than by ReAct agents. The Suggestion–Evaluation loop and the syntactic-validity loop remain unchanged. MAS-2 is thus architecturally matched to MAS-1 on communication structure and functional decomposition, differing only in the locus of tool orchestration, making the MAS-1/MAS-2 pair the cleanest comparison for the effect of Coordination Strategy under hierarchical control.



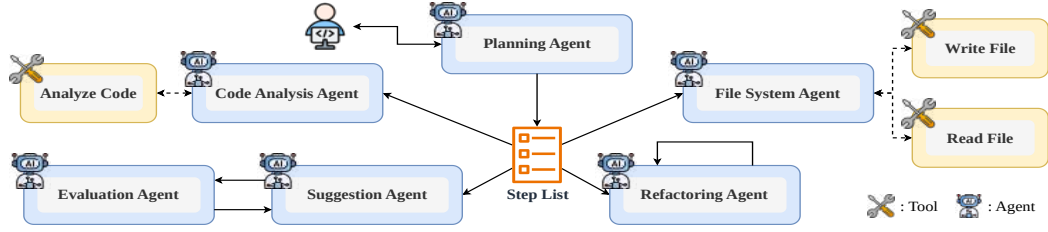
■ **Figure 2** MAS-2 (H+Sta). The tier-based organization is unchanged, but tool calls are embedded directly in the workflow rather than delegated to ReAct agents.

2.5 MAS-3: Centralized + Dynamic

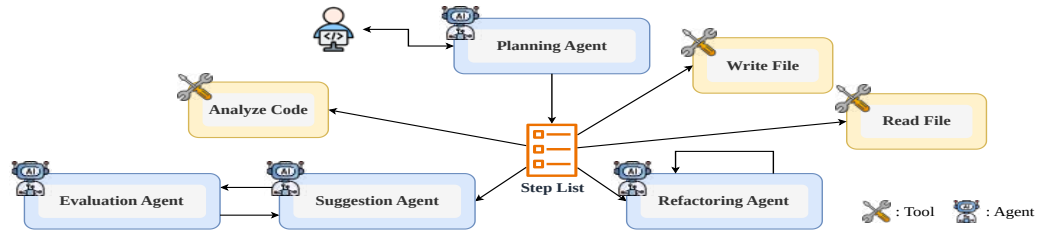
MAS-3 replaces the tier-based control structure with *Centralized* coordination while retaining *Dynamic* tool use. As shown in Figure 3, execution begins with a Planning Agent that decomposes the refactoring request into an explicit ordered step list identifying which agent should perform each step. File System and Code Analysis agents remain ReAct-based and retain autonomy over when and how to invoke their tools. Suggestion and Evaluation agents perform iterative refinement, and Refactoring Agent applies the validated transformation under the shared retry bounds. MAS-3 therefore centralizes global control without eliminating adaptive local tool use: relative to MAS-1, the key change is the explicit planner.

2.6 MAS-4: Centralized + Static

MAS-4 combines *Centralized* communication structure with *Static* coordination. As shown in Figure 4, the Planning Agent generates a step list that specifies both agent dispatch and tool calls. Execution follows that plan directly: file access and static analysis are performed through workflow-level tool invocation, after which the Suggestion, Evaluation, and Refactoring agents proceed under the same bounded refinement and syntactic retry mechanisms used in the other architectures. MAS-4 thus removes agent-level autonomy over tool use while preserving explicit planning as the locus of coordination.



■ **Figure 3** MAS-3 (C+Dyn). A Planning Agent specifies the global step sequence, while ReAct agents retain local autonomy over tool invocation.



■ **Figure 4** MAS-4 (C+Sta). Planning Agent controls both agent sequencing and tool invocation.

3 Experimental Design

3.1 Research Questions and Rationale

The experiment addresses three research questions organized around the three outcome dimensions, spanning the full evaluation scope of the study: from the capacity of each architecture to produce accepted outputs, through the cost incurred in doing so, to the structural character of those outputs and the efficiency with which they are generated.

RQ1-Rationale: LLM-MAS are non-deterministic and can fail to produce output, parseable code, or structurally plausible refactorings. Characterizing where each architecture breaks down is essential for assessing its practical usability and for interpreting any downstream comparison on cost or structural outcomes.

RQ2-Rationale: The cost of LLM-MAS directly constrains their deployment at scale, yet prior work bundles cost with full-system design. Isolating each factor's effect, and detecting whether they interact, is therefore essential for principled architectural guidance under realistic budgets.

RQ3-Rationale: Accepted output does not imply equivalent transformation behavior: architectures may differ in how aggressively they restructure the input code. Characterizing these structural differences, together with architecture-level efficiency, is essential for selecting a configuration aligned with a desired refactoring style and runtime profile.

3.2 Study Design and Experimental Units

The experimental object of the study is an LLM-MAS architecture for automated Python refactoring. The experimental unit is a single file-level refactoring instance. Treatments correspond to the four architectures obtained by crossing *Communication Structure* (Hierarchical vs. Centralized) with *Coordination Strategy* (Dynamic vs. Static): MAS-1 through MAS-4. The design is paired by construction: every instance is exposed to all four treatments under identical model, prompting, orchestration, and tool conditions. This repeated-measures setup

controls for input-level heterogeneity and allows observed differences to be attributed more directly to the manipulated architectural factors. All non-architectural elements are held constant across configurations, as detailed in Section 2.2.

3.3 Dataset Description

Isolating architectural effects requires labeled, real-world refactoring instances at scale. We therefore use a dataset [29] derived from *MLRefScanner* [25], a collection of refactoring-labeled commits mined from open-source Python ML repositories – a coherent ecosystem, since Python is the dominant language of the ML domain and its mature static analysis tooling directly supports our validation and measurement pipeline. Following the sampling procedure adopted in prior work [28], we use a sample of $N = 2,000$ file-level refactoring instances drawn from 167 repositories. The dataset spans eight refactoring types: Add Parameter, Rename Method, Remove Parameter, Change/Rename Parameter, Extract Method, Rename Class, Change Return Type, and Inline Method. Each instance contains two aligned file versions: the pre-refactoring snapshot (`original.py`), which serves as input to all four architectures, and the developer-produced post-refactoring snapshot (`human_ref.py`), used exclusively as a reference for calibrating the plausibility filter in downstream structural analyses and not as an oracle that generated outputs must reproduce.

3.4 Experimental Pipeline and Data Collection

For each input instance i , `original.py` is processed independently by each architecture $a \in \{1, 2, 3, 4\}$, producing two outputs: a candidate refactored file (`mas_refa,i.py`) when one is produced, and a system-level log (`sys_statisticsa,i.json`) recording execution status, execution time, LLM calls, and input, output, and total token counts regardless of outcome. Generated files undergo a three-stage validation pipeline: (i) *file generation* – whether any output file was produced; (ii) *syntactic validity* – whether the file parses via Python’s abstract syntax tree (AST) parser; (iii) *similarity-based acceptance* – whether the file exceeds the plausibility threshold defined in Section 3.5. Structural metrics are computed only for accepted outputs, and per-instance metric deltas $\Delta M_{a,i} = M(\text{mas_ref}_{a,i}.py) - M(\text{original}_i.py)$ quantify how strongly each architecture changes the structural properties of the input file, forming the basis of the structural comparisons reported for RQ3(a). Each `sys_statisticsa,i.json` is retained regardless of execution outcome: it operationalizes output generation success at the file generation stage (RQ1) and provides the cost and efficiency measures analyzed for RQ2 and RQ3(b).

3.5 Output Validation and Similarity-Based Filtering

Syntactic validity is necessary but not sufficient to treat a generated file as a plausible refactoring of the input. An architecture may produce Python code that parses successfully while being structurally unrelated to the original file. To reduce contamination in downstream structural analyses, we therefore apply a similarity-based plausibility filter that retains only outputs with strong correspondence to the input.

Composite Similarity Score. For each pair (`original.py`, candidate file), we compute a normalized similarity score $S \in [0, 1]$ as a weighted sum of six token-based similarity measures, capturing token overlap (Jaccard [24], $w = 0.111$; Dice [38], $w = 0.114$), global edit sensitivity and ordering (Levenshtein distance [23], $w = 0.241$; Levenshtein ratio [33], $w = 0.227$), and tolerance to limited reordering (Jaro [6], $w = 0.134$; Jaro–Winkler [40], $w = 0.172$). The weights are adopted from prior work on token-based code similarity [4].

Threshold Selection. We analyze the empirical distribution of S separately for each source: MAS-1 through MAS-4 and the human refactoring files. Shapiro-Wilk tests reject normality ($p < 0.05$) and D’Agostino’s test indicates left-skewed distributions consistent with the presence of low-similarity outliers. We fit a two-component Gaussian Mixture Model (GMM) to each source-specific distribution and define the source-specific threshold as the score at which the posterior probability of belonging to the high-similarity component equals 0.5, i.e., the Bayes decision boundary under symmetric misclassification costs. The resulting thresholds are 0.7698 for MAS-1, 0.7797 for MAS-2, 0.6226 for MAS-3, 0.7231 for MAS-4, and 0.8521 for the human refactoring distribution.

Reference Threshold and Filtering Policy. To enforce a single acceptance criterion across architectures, we adopt the threshold derived from the human refactoring distribution, *0.8521*, as the common filtering threshold for all automated outputs. This choice anchors plausibility to developer-produced refactorings while avoiding architecture-specific calibration, which would otherwise risk confounding the cross-architecture comparison. The policy is deliberately conservative: some outputs may be structurally plausible yet excluded because they diverge from typical human refactoring patterns. This trade-off is revisited in Section 6 as a threat to construct validity. The filter is used only to support conditional structural analysis. It does not establish semantic equivalence, behavioral preservation, or functional correctness. Its purpose is to exclude structurally implausible outputs before computing metrics intended to characterize architectural transformation behavior.

3.6 Evaluation Measures

Output Generation and Validity. Output generation and validity are evaluated at three successive stages over the full set of $N = 2,000$ instances per architecture. The *file generation rate* is the proportion of inputs for which the architecture produces any output file. The *syntactic validity rate* is the proportion for which the produced file parses successfully as Python. The *similarity acceptance rate* is the proportion that additionally exceeds the common plausibility threshold defined in Section 3.5. Together, these three measures distinguish failures to produce output, failures to produce syntactically valid output, and failures to produce structurally plausible output. Throughout the study, *validity* thus refers to syntactic parseability and similarity-based plausibility; it does not imply behavior preservation or successful application of the intended refactoring type (see Section 6).

Operational Cost and Execution Efficiency. From the system-level logs, we extract execution time, number of LLM calls, and token counts. These measures are collected for every execution regardless of output validity, since unsuccessful runs still consume time, calls, and tokens. In the analysis, they serve two roles. First, they operationalize *operational cost* for RQ2, where we estimate the main and interaction effects of the two architectural factors. Second, they support architecture-level comparison of *execution efficiency* for RQ3.

Structural Transformation. We compute eight static metrics commonly used in maintainability and complexity studies [10, 16, 35]: (i) average cyclomatic complexity (Avg_CC) [21]; (ii–iv) Halstead volume, effort, and difficulty (Hal_Vol, Hal_Eff, Hal_Diff) [8]; (v) lines of code (LoC) [27]; (vi) number of functions (Func_Count) [5]; (vii) comment density (Comm_Den) [26]; and (viii) average function length (Avg_Func_Len) [5]. Metrics are computed statically using *Radon* and *Lizard*; no code execution is required. Structural analyses are conducted on the metric deltas defined in Section 3.4 and only for outputs that pass both AST validation and similarity-based filtering.

3.7 Statistical Analysis

The statistical analysis is organized around the data structure and the three research questions. Three properties of the design guide our choice of procedures. First, Shapiro–Wilk tests reject normality ($p < 0.05$) for all paired differences, motivating non-parametric analyses throughout. Second, every input is processed by all four architectures, yielding a paired repeated-measures design whenever alignment is preserved. Third, all pairwise comparisons within a measure family are adjusted using Holm–Bonferroni [11] to control the family-wise error rate at $\alpha = 0.05$, retaining greater power than classical Bonferroni correction.

RQ1 Analysis. RQ1 examines how the four architectures differ in file generation success, syntactic validity, and similarity-based acceptance. Each outcome is binary at the instance level and summarized as a proportion over $N = 2,000$ executions per architecture. We use Pearson’s χ^2 test for the four-architecture comparison, and Fisher’s exact test for the six pairwise contrasts. Fisher’s exact is preferred for pairwise contrasts because expected counts may be small for the weakest configurations. Effect sizes are reported as Cohen’s h [3], with thresholds: negligible (< 0.2), small (0.2 to 0.5), medium (0.5 to 0.8), and large (≥ 0.8).

RQ2 Analysis. RQ2 examines the main effects of *Communication Structure* and *Coordination Strategy* on operational cost, and whether they interact. Because cost measures are recorded for every execution, the complete set of architecture-instance observations is available (4 architectures $\times$ 2,000 instances = 8,000 obs.). To obtain factor-level estimates under non-normality, we use the Aligned Rank Transform (ART) followed by standard two-way ANOVA [41], which aligns the data separately for each main effect and the interaction before ranking, thereby preserving factorial interpretability. Effect sizes are reported as partial η^2 : negligible (< 0.01), small (0.01 to 0.06), medium (0.06 to 0.14), and large (≥ 0.14).

RQ3 Analysis. RQ3 examines how the architectures differ in the structural transformations they apply and in their execution efficiency.

a) Structural transformation. Structural metrics are continuous, non-normal, and only available for outputs that survive both validation stages. Because this filtering yields different accepted subsets per architecture, we report two complementary analyses:

- *Paired analysis:* We restrict the sample to instances accepted by all four architectures, preserving the paired structure and controlling for input-specific variability. We apply Friedman tests followed by pairwise Wilcoxon signed-rank tests with Holm–Bonferroni correction. Effect sizes are reported as matched-pairs rank-biserial correlation r_{rb} : negligible (< 0.1), small (0.1 to 0.3), medium (0.3 to 0.5), and large (≥ 0.5).
- *Marginal analysis:* To avoid bias toward easier instances in the paired subset, we complement it with a marginal analysis using each architecture’s full accepted set as independent, unbalanced samples. We apply Kruskal–Wallis tests followed by Dunn’s test with Holm–Bonferroni correction. Effect sizes are reported as Cliff’s δ [32]: negligible (< 0.147), small (0.147 to 0.33), medium (0.33 to 0.474), and large (≥ 0.474).

We do not apply factorial ART to the structural metrics, as the paired intersection contains only $N = 146$ instances, providing insufficient power for reliable interaction testing across eight metrics. This limitation is revisited in Section 6.

b) Execution efficiency. To complement the factorial cost analysis of RQ2 with a direct architecture-level view, execution-efficiency measures are analyzed under the full paired design over $N_{\text{paired}} = 2,000$ instances. This sample is preserved because these measures

are available for every execution regardless of output validity. We apply Friedman tests followed by pairwise Wilcoxon signed-rank tests with Holm–Bonferroni correction, reporting r_{rb} as effect size. Because each architecture is executed once per input, these measures characterize observed execution behavior under a single execution per input; the consequences for stochastic variability are discussed in Section 6.

4 Results and Discussion

4.1 RQ1: Output Generation and Validity

Table 1 reports execution outcomes per architecture across the three stages defined in Section 3.6. Three architectures (MAS-1, MAS-2, MAS-4) reach near-complete levels ($\geq 94\%$) at both *file generation* and *syntactic validity*. MAS-3 is the only configuration that breaks down at these early stages, generating output for only 49.9% of inputs and a syntactically valid file in 37.7%. At *similarity acceptance*, the three non-MAS-3 architectures separate into a clear ordering: MAS-2 (75.5%) > MAS-4 (62.6%) > MAS-1 (53.0%). The bottom row of Table 1 (SA/SV) reports retention among syntactically valid outputs, where the contrast is cleaner: Static configurations retain a substantially larger fraction than Dynamic ones under both communication structures (76.4% and 66.5% vs. 54.7% and 52.3%).

■ **Table 1** Output generation and validity per architecture. The first three rows show successful instances out of $N = 2,000$.

Outcome	MAS-1 (H+Dyn)	MAS-2 (H+Sta)	MAS-3 (C+Dyn)	MAS-4 (C+Sta)
File generation (FG)	1993 (99.7%)	2000 (100.0%)	997 (49.9%)	2000 (100.0%)
Syntactic validity (SV)	1935 (96.8%)	1976 (98.8%)	753 (37.7%)	1884 (94.2%)
Similarity acceptance (SA)	1059 (53.0%)	1509 (75.5%)	394 (19.7%)	1252 (62.6%)
SA / SV	54.7%	76.4%	52.3%	66.5%

Factorial labels: H/C = Hierarchical / Centralized; Dyn/Sta = Dynamic / Static.

Pearson’s χ^2 rejects equality of rates across architectures at every stage (Table 2), so we use pairwise Fisher exact tests to localize differences. At *file generation*, MAS-3 differs from every other architecture with large effects, while MAS-1, MAS-2, and MAS-4 remain indistinguishable. The same pattern holds at *syntactic validity*, except for a small but significant advantage of MAS-2 over MAS-4. At *similarity acceptance*, MAS-3 again differs from all others with medium-to-large effects; in addition, MAS-2 accepts more outputs than MAS-1 and MAS-4, while MAS-1 and MAS-4 are indistinguishable.

■ **Table 2** Pairwise comparisons for output generation and validity.

Pair	File gen.				Syntactic				Similarity			
	Sig	Dir	h	Mag	Sig	Dir	h	Mag	Sig	Dir	h	Mag
MAS _{1vs2}	✗	—	—	—	✗	—	—	—	✓	↓	-0.47	*
MAS _{1vs3}	✓	↑	1.46	***	✓	↑	1.46	***	✓	↑	0.71	**
MAS _{1vs4}	✗	—	—	—	✗	—	—	—	✗	—	—	—
MAS _{2vs3}	✓	↑	1.57	***	✓	↑	1.60	***	✓	↑	1.19	***
MAS _{2vs4}	✗	—	—	—	✓	↑	0.27	*	✓	↑	0.28	*
MAS _{3vs4}	✓	↓	-1.57	***	✓	↓	-1.33	***	✓	↓	-0.91	***

Sig (Significance): ✓ rejects equality after Holm–Bonferroni ($p_{adj} < 0.05$); ✗ otherwise. **Dir** (Direction): sign of h ; ↑ indicates that MAS_X exhibits a higher rate than MAS_Y in the comparison MAS_{XvsY}; ↓ the opposite. h : Cohen’s h effect size. **Mag** (Magnitude): $|h|$ classified as negligible (—), small (*), medium (**), or large (***).

The results reveal two distinct phenomena. Early in the pipeline, failures concentrate in a single configuration, MAS-3 (Centralized+Dynamic), rather than in all Dynamic or all Centralized architectures. A plausible explanation is the combination of global planning with local tool autonomy: the plan presumes actions such as writing the output file, but the ReAct agents retain discretion over tool invocation and may terminate without executing them, so a run can complete its steps without materializing an output; root-cause analysis of these failures is left to future work. Later, once outputs are syntactically valid, the dominant separation shifts to Coordination Strategy: Static configurations preserve a substantially larger share of plausible outputs under the common similarity filter, regardless of communication structure.

Answer to RQ1. At file generation and syntactic validity, only MAS-3 (C+Dyn) fails substantially, while MAS-1, MAS-2, and MAS-4 perform similarly. At similarity acceptance, the dominant separation is by Coordination Strategy: Static configurations retain a substantially larger fraction of syntactically valid outputs than Dynamic ones under both communication structures, with MAS-2 (H+Sta) achieving the highest final acceptance rate.

4.2 RQ2: Factorial Analysis of Operational Cost

Table 3 shows a strong descriptive pattern across cost measures. Within both communication structures, Dynamic coordination consumes more than Static on every measure. However, the size of this penalty differs sharply: moving from Static to Dynamic raises total tokens by approximately +138% under Hierarchical communication (from 15,104 to 35,844) but by only +31% under Centralized (from 20,156 to 26,338). MAS-2 (H+Sta) is the global cost minimum on every measure.

■ **Table 3** Median values per architecture for cost measures analyzed by ART-ANOVA.

Measure	Hierarchical		Centralized	
	MAS-1 (Dyn)	MAS-2 (Sta)	MAS-3 (Dyn)	MAS-4 (Sta)
Execution time (s)	82.2	60.9	124.7	70.9
LLM calls	12	3	11	8
Input tokens	26,247	11,697	19,838	15,986
Output tokens	9,606	3,362	6,843	3,793
Total tokens	35,844	15,104	26,338	20,156

ART-ANOVA (Table 4) confirms this pattern across measures. Coordination Strategy is the dominant main effect: large on LLM calls and on input, output, and total tokens, and medium on execution time. Communication Structure is more heterogeneous: large on LLM calls, medium on execution time, small on output tokens, and negligible on input and total tokens. The interaction is significant on all measures, but its magnitude varies from small on execution time to large on LLM calls. In all cases, the interaction shows that the cost impact of Dynamic coordination depends on the communication structure with which it is paired.

The interaction deserves emphasis because it changes the substantive reading of the main effects. Considering only main effects, Communication Structure would appear nearly irrelevant for token-based cost. Once the interaction is included, however, the Hierarchical structure emerges as the setting where Dynamic coordination becomes especially expensive. This explains why MAS-1 is the most token-intensive configuration even though the isolated main effect of structure on total tokens is negligible.

■ **Table 4** ART-ANOVA effects and interaction on cost measures.

Measure	Structure (C & H)		Coordination (Dyn & Sta)		Interaction	
	η_p^2	Mag	η_p^2	Mag	η_p^2	Mag
Execution time	0.092	**	0.137	**	0.037	*
LLM calls	0.343	***	0.774	***	0.504	***
Input tokens	0.002	—	0.238	***	0.084	**
Output tokens	0.034	*	0.377	***	0.063	**
Total tokens	0.007	—	0.278	***	0.080	**

η_p^2 : partial eta-squared effect size. **Mag** (Magnitude): η_p^2 classified as negligible (—), small (*), medium (**), or large (***).

Answer to RQ2. Coordination Strategy is the dominant factor, with large effects on LLM calls and token consumption, and a medium effect on execution time. Communication Structure has a heterogeneous profile: large on LLM calls, medium on execution time, small on output tokens, and negligible on input and total tokens. The two factors interact on every cost measure, with the strongest interaction on LLM calls (large) and the weakest on execution time (small). Across all interactions, Dynamic coordination is substantially more expensive under Hierarchical than Centralized communication, making MAS-1 (H+Dyn) the most costly configuration and MAS-2 (H+Sta) the least costly.

4.3 RQ3: Structural Transformations and Execution Efficiency

4.3.1 Structural Transformations

Structural metrics are analyzed at two levels. The *paired* analysis uses the intersection of instances accepted by all four architectures ($N_{\text{paired}} = 146$), preserving input alignment and controlling for input-level variability. The *marginal* analysis uses each architecture’s full accepted set ($N = 1059, 1509, 394, 1252$ for MAS-1 through MAS-4), providing higher power at the cost of comparing unbalanced, non-aligned samples.

Paired analysis. Friedman tests reject equality on four of the eight metrics (Avg_CC, Hal_Diff, LoC, Func_Count). Pairwise Wilcoxon comparisons (Table 5) reveal a single dominant pattern: MAS-1 (H+Dyn) separates from each of the other three architectures on Avg_CC, LoC, and Func_Count, with medium-to-large effects. Relative to the others, MAS-1 generates more functions, longer files, and lower average cyclomatic complexity per function. Hal_Diff differentiates only MAS-1 vs 4. All comparisons among MAS-2, MAS-3, and MAS-4 are non-significant.

Marginal analysis. Kruskal–Wallis rejects equality on all eight metrics. The paired result is largely reproduced: MAS-1 remains the main structural outlier, again generating more functions, longer files, and lower Avg_CC relative to the others (Table 6). These effects are smaller than in the paired analysis, as expected given the added input heterogeneity. The marginal analysis also detects differences among MAS-2, MAS-3, and MAS-4 not visible in the paired analysis, but these are uniformly small or negligible. Marginal comparisons should be read with caution, particularly those involving MAS-3, since its accepted subset contains only 394 of 2,000 instances and is thus biased toward inputs on which MAS-3 produces plausible outputs; this post-treatment selection bias is discussed in Section 6.

■ **Table 5** Paired pairwise comparisons (Wilcoxon signed-rank + Holm–Bonferroni) for significant structural metrics where $N_{\text{paired}} = 146$.

Pair	Avg_CC				Hal_Diff				LoC				Func_Count			
	Sig	Dir	r_{rb}	Mag	Sig	Dir	r_{rb}	Mag	Sig	Dir	r_{rb}	Mag	Sig	Dir	r_{rb}	Mag
MAS _{1vs2}	✓	↓	-0.58	***	✗	–	–	–	✓	↑	0.53	***	✓	↑	0.52	***
MAS _{1vs3}	✓	↓	-0.54	***	✗	–	–	–	✓	↑	0.46	**	✓	↑	0.46	**
MAS _{1vs4}	✓	↓	-0.39	**	✓	↓	-0.45	**	✓	↑	0.56	***	✓	↑	0.46	**
MAS _{2vs3}	✗	–	–	–	✗	–	–	–	✗	–	–	–	✗	–	–	–
MAS _{2vs4}	✗	–	–	–	✗	–	–	–	✗	–	–	–	✗	–	–	–
MAS _{3vs4}	✗	–	–	–	✗	–	–	–	✗	–	–	–	✗	–	–	–

Sig (Significance): ✓ rejects equality after Holm–Bonferroni ($p_{adj} < 0.05$); ✗ otherwise. **Dir** (Direction): sign of r_{rb} ; ↑ indicates that MAS_X exhibits a higher value than MAS_Y in the comparison MAS_{XvsY}; ↓ the opposite. r_{rb} : matched-pairs rank-biserial. **Mag** (Magnitude): $|r_{rb}|$ classified as negl. (–), small (*), medium (**), or large (***).

■ **Table 6** Marginal pairwise comparisons (Dunn + Holm–Bonferroni) for significant structural metrics where $N_{\text{marginal}} = 1059, 1509, 394, 1252$.

Pair	Avg_CC				Hal_Vol				Hal_Eff				Hal_Diff			
	Sig	Dir	δ	Mag	Sig	Dir	δ	Mag	Sig	Dir	δ	Mag	Sig	Dir	δ	Mag
MAS _{1vs2}	✓	↓	-0.15	*	✓	↑	0.07	–	✓	↓	-0.05	–	✓	↓	-0.10	–
MAS _{1vs3}	✓	↓	-0.22	*	✓	↑	0.08	–	✓	↓	-0.17	*	✓	↓	-0.20	*
MAS _{1vs4}	✓	↓	-0.13	–	✓	↑	0.09	–	✓	↓	-0.05	–	✓	↓	-0.12	–
MAS _{2vs3}	✗	–	–	–	✗	–	–	–	✓	↓	-0.12	–	✓	↓	-0.09	–
MAS _{2vs4}	✗	–	–	–	✗	–	–	–	✗	–	–	–	✗	–	–	–
MAS _{3vs4}	✓	↑	0.09	–	✗	–	–	–	✓	↑	0.12	–	✓	↑	0.07	–

Pair	LoC				Comm_Den				Func_Count				Avg_Func_Len			
	Sig	Dir	δ	Mag	Sig	Dir	δ	Mag	Sig	Dir	δ	Mag	Sig	Dir	δ	Mag
MAS _{1vs2}	✓	↑	0.39	**	✓	↓	-0.09	–	✓	↑	0.35	**	✓	↓	-0.09	–
MAS _{1vs3}	✓	↑	0.31	*	✓	↓	-0.22	*	✓	↑	0.44	**	✓	↓	-0.17	*
MAS _{1vs4}	✓	↑	0.45	**	✗	–	–	–	✓	↑	0.34	**	✗	–	–	–
MAS _{2vs3}	✓	↓	-0.14	–	✓	↓	-0.12	–	✓	↑	0.09	–	✗	–	–	–
MAS _{2vs4}	✓	↑	0.09	–	✓	↑	0.06	–	✗	–	–	–	✓	↑	0.06	–
MAS _{3vs4}	✓	↑	0.23	*	✓	↑	0.17	*	✓	↓	-0.11	–	✓	↑	0.13	–

Sig (Significance): ✓ rejects equality after Holm–Bonferroni ($p_{adj} < 0.05$); ✗ otherwise. **Dir** (Direction): sign of δ ; ↑ indicates that MAS_X exhibits a higher value than MAS_Y in the comparison MAS_{XvsY}; ↓ the opposite. δ : Cliff’s delta effect size. **Mag** (Magnitude): $|\delta|$ classified as negligible (–), small (*), medium (**), or large (***).

Both analyses converge on the same interpretation: architectural separation in structural transformation is concentrated in MAS-1 (H+Dyn), which applies the most decomposition-heavy transformation profile, generating more functions and longer files while lowering average per-function complexity. MAS-2, MAS-3, and MAS-4 remain structurally close.

Answer to RQ3(a). MAS-1 (H+Dyn) is the main structural outlier, generating more functions, longer files, and lower average cyclomatic complexity per function than the other architectures. MAS-2, MAS-3, and MAS-4 are structurally close, with mostly small or negligible differences across both paired and marginal analyses.

4.3.2 Execution Efficiency

Friedman tests reject equality on every efficiency measure across all $N_{\text{paired}} = 2,000$ executions. Pairwise Wilcoxon comparisons (Table 7) reveal a highly regular structure: for LLM calls, input, output, and total tokens, the four architectures follow a strict ordering, MAS-2 (H+Sta)

$< \text{MAS-4 (C+Sta)} < \text{MAS-3 (C+Dyn)} < \text{MAS-1 (H+Dyn)}$, with every pairwise contrast significant and nearly all effects large, consistent with RQ2. Execution time follows a different ordering: $\text{MAS-2} < \text{MAS-4} < \text{MAS-1} < \text{MAS-3}$, inverting the positions of MAS-1 and MAS-3. This inversion is substantively important: despite being the most token-expensive, MAS-1 is faster than MAS-3, whose elapsed-time penalty reflects inefficient execution driven by its severe early-stage output generation failures rather than by token volume alone.

■ **Table 7** Pairwise comparisons on execution efficiency (Wilcoxon signed-rank + Holm–Bonferroni, $N_{\text{paired}} = 2,000$). All comparisons are significant after Holm–Bonferroni ($p_{\text{adj}} < 0.05$).

Pair	Exec. time			LLM calls			In. tokens			Out. tokens			Total tok.		
	Dir	r_{rb}	Mag	Dir	r_{rb}	Mag	Dir	r_{rb}	Mag	Dir	r_{rb}	Mag	Dir	r_{rb}	Mag
MAS _{1vs2}	↑	0.45	**	↑	0.99	***	↑	0.92	***	↑	0.98	***	↑	0.97	***
MAS _{1vs3}	↓	-0.47	**	↑	0.67	***	↑	0.52	***	↑	0.47	**	↑	0.58	***
MAS _{1vs4}	↑	0.25	*	↑	0.96	***	↑	0.72	***	↑	0.94	***	↑	0.85	***
MAS _{2vs3}	↓	-0.88	***	↓	-0.99	***	↓	-0.43	**	↓	-0.99	***	↓	-0.65	***
MAS _{2vs4}	↓	-0.33	**	↓	-0.99	***	↓	-0.57	***	↓	-0.61	***	↓	-0.66	***
MAS _{3vs4}	↑	0.78	***	↑	0.98	***	↑	0.38	**	↑	0.89	***	↑	0.56	***

Dir (Direction): sign of r_{rb} ; ↑ indicates that MAS_X exhibits a higher value than MAS_Y in the comparison MAS_{XvsY}; ↓ the opposite. **r_{rb}** : matched-pairs rank-biserial correlation. **Mag** (Magnitude): $|r_{rb}|$ classified as negligible (–), small (*), medium (**), or large (***).

Answer to RQ3(b). Token-based and call-based measures follow a stable ordering ($\text{MAS-2} < \text{MAS-4} < \text{MAS-3} < \text{MAS-1}$), whereas elapsed time follows a different ordering in which MAS-3 (C+Dyn) becomes the slowest. Static configurations are the most efficient and stable, while Dynamic configurations diverge into two behaviors: high-cost decomposition-heavy transformation (MAS-1, H+Dyn) and severe output generation failures with poor runtime efficiency (MAS-3, C+Dyn).

4.4 Discussion

Coordination Strategy is the dominant execution-control dimension. Across the three research questions, the clearest and most reproducible pattern is the advantage of Static over Dynamic coordination, visible in the retention of plausible outputs after syntactic validation, in the dominant main effects on nearly all cost measures, and in the stable architecture ordering for token- and call-based efficiency. This empirically supports the broader LLM-MAS literature, which treats coordination and tool orchestration as core architectural variables rather than secondary implementation details [7, 9, 18], and shows that whether tool use is prescribed in advance or delegated adaptively at runtime matters more than how control is organized among agents.

Architectural effects are interaction-sensitive rather than purely additive. The strongest methodological result is not the main effect of Coordination Strategy alone, but its interaction with Communication Structure on all cost measures. Hierarchical communication does not uniformly increase or decrease cost; instead, it specifically amplifies the penalty of Dynamic coordination. As shown in Table 3, moving from Static to Dynamic raises total tokens by +138% under Hierarchical communication, but only +31% under Centralized. MAS-1 is thus the worst token-consuming architecture not because Hierarchical structure is generally costly, but because it is costly when paired with Dynamic tool orchestration. These interactions arise even under a fixed model, task, and tooling context, complementing recent comparative work that found limited differences across other agent-configuration axes [15].

Dynamic coordination does not fail in one way; it fails in two. The two Dynamic architectures do not form a coherent family in the way the two Static ones do. MAS-1 succeeds at file generation and syntactic validity but is expensive and structurally aggressive, whereas MAS-3 is comparatively less token-expensive yet fails to produce usable outputs for a large fraction of instances and exhibits the worst runtime behavior. This divergence suggests that Dynamic coordination increases sensitivity to the surrounding control structure rather than exerting a uniform effect, widening the behavioral envelope toward either expensive productive transformation or unstable execution depending on how control is organized.

Structural improvement is not one-dimensional. MAS-1 lowers average cyclomatic complexity per function while simultaneously increasing lines of code and number of functions, indicating a decomposition-heavy transformation style consistent with the broader refactoring literature, which has long shown that improvements in one structural dimension may coexist with increases in others [16, 22]. MAS-1 thus tends toward a specific structural strategy, breaking code into more, simpler units at the price of larger files and higher fragmentation, rather than performing uniformly better refactoring overall.

Implications for empirical evaluation of LLM-MAS. The results also speak to evaluation methodology. Recent surveys note that most existing LLM-MAS studies evaluate complete systems end-to-end, making architectural attribution difficult [9, 18]. This study shows that a controlled factorial design can expose interaction effects that partial comparisons would miss: comparing only MAS-2 against MAS-3 might suggest centralized control is problematic, while comparing only MAS-1 against MAS-2 might suggest Dynamic coordination is universally harmful, yet the full crossed design shows that what matters is the combination of both dimensions, not either in isolation.

Practical reading of the four configurations. For automated Python refactoring under the conditions studied here, MAS-2 (Hierarchical+Static) is the most attractive default, combining the best similarity acceptance rate with the lowest operational cost while avoiding the structural extremity of MAS-1 and the output generation failures of MAS-3. MAS-4 (Centralized+Static) is a credible alternative when explicit planning is desired, but consistently weaker than MAS-2 on validity and cost. The two Dynamic configurations are harder to justify as defaults: one is operationally expensive, the other operationally unstable. While this does not mean Dynamic coordination is never useful, in the present refactoring setting its benefits are not evident enough to offset its cost and output generation liabilities.

5 Related Work

LLM-MAS Design Space in Software Engineering. Prior work explores a wide range of LLM-MAS designs for automating SE activities that exceed the scope of single-prompt assistance, including development workflows [12, 30], issue resolution and bug fixing [1, 2, 19, 20, 37], and code review and generation [36, 47]. Across these tasks, systems differ in how responsibilities are distributed, how agents communicate, how execution is planned, and how tools are incorporated: ChatDev [30] and MetaGPT [12] decompose work into specialized roles with structured interaction protocols; Coder [2], MASAI [1], MARSCoDe [19], Alibaba Lingma [20], and MAGIS [37] combine task decomposition, repository exploration, and tool-supported interaction in different ways; and CodeAgent [36] and AdaCoder [47] vary in supervisory control, with CodeAgent centralizing oversight through a dedicated QA-Checker

agent and AdaCoder deferring planning adaptively. However, most of these systems are evaluated as complete systems under task-specific configurations, which makes architectural effects difficult to separate from model choice, prompt design, or tooling, a limitation that recent surveys identify with respect to SE-specific applications [9,18] and general multi-agent collaboration mechanisms [7,17,39,45], and that our results confirm: the interaction between Communication Structure and Coordination Strategy would remain undetectable under end-to-end evaluation.

Multi-Agent Approaches to Automated Refactoring. Within automated refactoring, the literature is smaller and more fragmented: MANTRA [44] combines retrieval-augmented generation with multi-agent collaboration for method-level Java refactoring; Siddeeq et al. [34] propose a multi-agent Haskell refactoring pipeline; RefactorGPT [14] structures refactoring as a sequence of specialized ChatGPT agents; MUARF [43] introduces a developer-reviewer workflow as a notable example of runtime-adaptive coordination; Rajendran et al. [31] describe a centralized conceptual framework without empirical evaluation; and prior empirical work on Python refactoring demonstrates feasibility but evaluates systems end to end [28]. The most salient architectural variation concerns two questions: how control and information flow are organized among agents (*Communication Structure*), and whether tool invocation and execution order are fixed in advance or resolved at runtime (*Coordination Strategy*). Yet existing evaluations bundle these choices with model selection, prompt design, and task-specific tooling [14,28,34,43,44], making attribution to either factor impossible. Our results show that this attribution matters: Coordination Strategy emerged as the dominant factor across all outcome dimensions, and its interaction with Communication Structure would not have been isolable without a controlled crossed design.

Empirical Comparison of Agent Architectures. The gap is not the absence of LLM-MAS for refactoring, but the absence of controlled empirical evidence isolating architectural effects from confounding factors, a limitation explicitly identified in prior surveys [7,9,18]. Closest to our work, Kim [15] compares agent configurations varying in specialization granularity across nine SE tasks, finding small performance differences and attributing most behavior to the underlying model. Our findings complement and extend that result: while Kim finds limited architectural signal across granularity configurations, we find strong signal when Communication Structure and Coordination Strategy are crossed. This suggests that the relevant axis of variation matters and that factorial designs crossing multiple dimensions may reveal effects that single-axis comparisons miss.

6 Threats to Validity

We structure this section following Wohlin et al. [42], addressing conclusion, internal, construct, and external validity.

Conclusion validity. *Low statistical power* arises most clearly in the structural analysis: after AST validation and similarity-based filtering, the jointly accepted intersection contains only $N = 146$ instances, insufficient for reliable interaction testing across eight structural metrics; we thus restrict factorial inference to cost measures and analyze structural outcomes through complementary paired and marginal non-parametric comparisons. *Random variation in treatment execution* is a concern because each architecture runs only once per input – given the stochastic nature of LLM inference, repeated executions could yield different

outputs, costs, or failure rates. *Violated assumptions in factor-level analysis* remain a risk for RQ2, as ART-ANOVA does not explicitly model the instance-level dependence induced by the repeated-measures design; we partially mitigate this by complementing the factorial analysis with paired Friedman/Wilcoxon comparisons and reporting effect sizes throughout. Finally, to reduce *multiplicity*, all pairwise comparisons are adjusted with Holm–Bonferroni correction.

Internal validity. The main threat concerns *confounding in treatment implementation*: Communication Structure and Coordination Strategy are manipulated through specific instantiations involving fixed agent roles, prompts, planning artifacts, and LangGraph workflows, so the observed effects should be read as properties of these operationalizations rather than of all possible hierarchical, centralized, static, or dynamic architectures. A second threat is *post-treatment selection*: structural metrics are available only for outputs passing both validation stages, yielding unequal accepted subsets. We mitigate this by reporting both a paired analysis on the jointly accepted subset and a marginal analysis on each architecture’s full accepted set.

Construct validity. The principal threat is *incomplete operationalization of refactoring quality*: the eight static metrics capture complexity, size, modularity, and documentation-related properties but do not establish semantic correctness or behavior preservation, and the similarity-based filter excludes structurally implausible outputs without guaranteeing that retained ones are behavior-preserving or that the intended refactoring type was applied. A second subthreat is *operationalization bias* in the use of `human_ref.py`: while this choice avoids architecture-specific calibration, the resulting human-anchored threshold (0.8521) is conservative and may disproportionately exclude outputs from architectures whose similarity distributions are shifted downward, most notably MAS-3 (source-specific threshold 0.6226). A third threat concerns *cost measurement*: execution time, token usage, and call counts do not directly model monetary cost, energy consumption, or deployment-time latency.

External validity. Several subthreats limit generalizability. Under *interaction of treatment and subjects*, the dataset is restricted to file-level Python refactoring instances drawn from ML repositories in *MLRefScanner*. Findings may not generalize to other languages, domains, or refactoring granularities; tasks with broader repository context or longer action horizons, such as issue resolution, may reward adaptive coordination more than the file-level setting studied here. Under *interaction of treatment and setting*, all configurations use a single foundation model (`open-mixtral-8x22b`) and orchestration substrate (LangGraph). Other model families or frameworks may interact differently with hierarchical versus centralized communication or static versus dynamic coordination: models with stronger native tool-calling could narrow the observed Static–Dynamic gap, although the interaction between the two factors, being structural rather than model-specific, is less tied to this experimental setting. Finally, under *interaction of treatment and construct operationalization*, the study examines only cooperative architectures spanning two manipulated dimensions and four configurations. Other regions of the LLM-MAS design space – decentralized communication, non-cooperative interaction, alternative planning mechanisms, and different tools – remain outside this paper’s scope.

7 Conclusion

This paper presented a controlled 2×2 factorial comparison of four LLM-MAS for automated Python refactoring, crossing *Communication Structure* (Hierarchical vs. Centralized) with *Coordination Strategy* (Dynamic vs. Static) across 2,000 real-world instances under identical model, dataset, orchestration, prompting, and tool conditions.

Coordination Strategy emerged as the dominant architectural factor: Static configurations consistently retained a larger fraction of syntactically valid outputs under the plausibility filter and incurred substantially lower token-based and call-based costs than Dynamic configurations. *Communication Structure* showed a heterogeneous profile, strongly affecting call behavior, with a medium effect on execution time, a small effect on output tokens, and a negligible effect on input/total token consumption. At the architecture level, MAS-2 (H+Sta) provided the best tradeoff between validity and cost, MAS-1 (H+Dyn) was the most expensive configuration and the main structural outlier, and MAS-3 (C+Dyn) was the only configuration to fail on a substantial fraction of inputs.

Static coordination is the preferable default when output validity and operational cost are primary concerns, and the controlled factorial design exposes interaction effects that remain hidden in standard end-to-end benchmarking. Future work should incorporate behavioral correctness checks, root-cause analysis of architecture-specific failures, repeated executions under controlled randomness, additional LLM families, alternative orchestration frameworks, other languages and repository domains, and broader regions of the LLM-MAS design space, including decentralized and non-cooperative architectures.

Data Availability Statement

The replication package accompanying this study is openly available on Zenodo under a CC-BY 4.0 license (source code additionally under MIT) at: <https://doi.org/10.5281/zenodo.20042886>. The package includes: (i) the source code of the four LLM-MAS architectures (MAS-1 through MAS-4), implemented with LangGraph, with documentation and scripts to execute each architecture individually or all four in sequence; (ii) the experimental dataset of $N = 2,000$ Python refactoring instances collected during the experiment, comprising the original, human-refactored, and four agent-refactored versions of each file, together with the per-execution system logs; (iii) the analysis scripts covering the full statistical pipeline (output validity tests, structural comparisons, and cost analysis) used to produce all results reported for RQ1, RQ2, and RQ3.

References

- 1 Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. Masai: Modular architecture for software-engineering ai agents. *arXiv preprint*, 2024. doi:10.48550/arXiv.2406.11638.
- 2 Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. Coder: Issue resolving with multi-agent and task graphs. *arXiv preprint*, 2024. arXiv:2406.01304.
- 3 Jacob Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, Hillsdale, NJ, 2nd edition, 1988.
- 4 Siyue Feng, Wenqi Suo, Yueming Wu, Deqing Zou, Yang Liu, and Hai Jin. Machine learning is all you need: A simple token-based approach for effective code clone detection. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024. doi:10.1145/3597503.3639114.

- 5 Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.
- 6 Najlah Gali, Radu Marinescu-Istodor, Damien Hostettler, and Pasi Fränti. Framework for syntactic string similarity measures. *Expert Systems with Applications*, 129:169–185, 2019. doi:10.1016/J.ESWA.2019.03.048.
- 7 Taicheng Guo, Xiuying Chen, Yaqi Wang, Ruidi Chang, Shichao Pei, Nitesh V Chawla, Olaf Wiest, and Xiangliang Zhang. Large language model based multi-agents: A survey of progress and challenges. *arXiv preprint*, 2024. doi:10.48550/arXiv.2402.01680.
- 8 Maurice H Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.
- 9 Junda He, Christoph Treude, and David Lo. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–30, 2025. doi:10.1145/3712003.
- 10 Ilja Heitlager, Tobias Kuipers, and Joost Visser. A practical model for measuring maintainability. In *6th international conference on the quality of information and communications technology (QUATIC 2007)*, pages 30–39. IEEE, 2007. doi:10.1109/QUATIC.2007.8.
- 11 Sture Holm. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics*, pages 65–70, 1979.
- 12 Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint*, 2023. arXiv:2308.00352.
- 13 Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. Explainable automated debugging via large language model-driven scientific debugging. *Empirical Software Engineering*, 30(2):45, 2025. doi:10.1007/S10664-024-10594-X.
- 14 Muhammed Abdulhamid Karabiyik. Refactorgpt: a chatgpt-based multi-agent framework for automated code refactoring. *PeerJ Computer Science*, 11:e3257, 2025. doi:10.7717/PEERJ-CS.3257.
- 15 Dae-Kyoo Kim. Artifact validity under varying agent configurations in llm-assisted software development: A comparative analysis. *Information and Software Technology*, 192:108022, 2026. doi:10.1016/J.INFSOF.2026.108022.
- 16 Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. An empirical study of refactoring challenges and benefits at microsoft. *IEEE Transactions on Software Engineering*, 40(7):633–649, 2014. doi:10.1109/TSE.2014.2318734.
- 17 Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth*, 1(1):9, 2024.
- 18 Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint*, 2024. doi:10.48550/arXiv.2409.02977.
- 19 Yizhou Liu, Pengfei Gao, Xinchun Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. Marscode agent: Ai-native automated bug fixing. *arXiv preprint*, 2024. doi:10.48550/arXiv.2409.00899.
- 20 Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. Alibaba lingmaagent: Improving automated issue resolution via comprehensive repository exploration. In *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, pages 238–249, 2025. doi:10.1145/3696630.3728549.
- 21 Thomas J McCabe. A complexity measure. *IEEE Transactions on software Engineering*, SE-2(4):308–320, 1976. doi:10.1109/TSE.1976.233837.
- 22 Tom Mens and Tom Tourwé. A survey of software refactoring. *IEEE Transactions on software engineering*, 30(2):126–139, 2004. doi:10.1109/TSE.2004.1265817.
- 23 Gonzalo Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001. doi:10.1145/375360.375365.

- 24 Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. Using of jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, volume 1, pages 380–384, 2013.
- 25 Shayan Noei, Heng Li, and Ying Zou. Detecting refactoring commits in machine learning python projects: A machine learning-based approach. *ACM Transactions on Software Engineering and Methodology*, 34(3):1–25, 2025. doi:10.1145/3705309.
- 26 Alberto S Nuñez-Varela, Héctor G Pérez-Gonzalez, Francisco E Martínez-Perez, and Carlos Soubervielle-Montalvo. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164–197, 2017. doi:10.1016/J.JSS.2017.03.044.
- 27 Roger S Pressman. Software engineering: a practitioner’s approach. *Pressman and Associates*, 2005.
- 28 Alexander Puma Pucho, Alexandre Mello Ferreira, Elder José Reioli Cirilo, and Bruno B. P. Cafeo. Refactoring python code with llm-based multi-agent systems: An empirical study in ml software projects. In *Simpósio Brasileiro de Engenharia de Software (SBES)*, pages 678–684, Porto Alegre, RS, Brasil, 2025. SBC. doi:10.5753/sbes.2025.11033.
- 29 Alexander Puma Pucho, Alexandre Mello Ferreira, Elder José Reioli Cirilo, and Bruno B. P. Cafeo. Refactoring python code with llm-based multi-agent systems: An empirical study in ml software projects [replication package]. Zenodo, August 2025. doi:10.5281/zenodo.16988695.
- 30 Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. *arXiv preprint*, 2023. arXiv:2307.07924.
- 31 Vasanth Rajendran, Dinesh Besiahgari, Sachin C Patil, Manjunath Chandrashekaraiiah, and Vishnu Challagulla. A multi-agent llm environment for software design and refactoring: a conceptual framework. In *SoutheastCon 2025*, pages 488–493. IEEE, 2025.
- 32 Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen’s d for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, volume 177, 2006.
- 33 Sandip Sarkar, Dipankar Das, Partha Pakray, and Alexander Gelbukh. Junitmz at semeval-2016 task 1: Identifying semantic similarity using levenshtein ratio. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)*, pages 702–705, 2016. doi:10.18653/V1/S16-1108.
- 34 Shahbaz Siddeeq, Muhammad Waseem, Zeeshan Rasheed, Md Mahade Hasan, Jussi Rasku, Mika Saari, Henri Terho, Kalle Mäkelä, Kai-Kristian Kemell, and Pekka Abrahamsson. Llm-based multi-agent system for intelligent refactoring of haskell code. In *International Conference on Product-Focused Software Process Improvement*, pages 408–418. Springer, 2025. doi:10.1007/978-3-032-12089-2_26.
- 35 Konstantinos Stroggylos and Diomidis Spinellis. Refactoring—does it improve software quality? In *Fifth International Workshop on Software Quality (WoSQ’07: ICSE Workshops 2007)*, pages 10–10. IEEE, 2007.
- 36 Xunzhu Tang, Kisub Kim, Yewei Song, Cedric Lothritz, Bei Li, Saad Ezzini, Haoye Tian, Jacques Klein, and Tegawendé Bissyandé. Codeagent: Autonomous communicative agents for code review. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 11279–11313, 2024. doi:10.18653/V1/2024.EMNLP-MAIN.632.
- 37 Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. Magis: Llm-based multi-agent framework for github issue resolution. *Advances in Neural Information Processing Systems*, 37:51963–51993, 2024.
- 38 Vikas Thada and Vivek Jaglan. Comparison of jaccard, dice, cosine similarity coefficient to find best fitness value for web retrieved documents using genetic algorithm. *International Journal of Innovations in Engineering and Technology*, 2(4):202–205, 2013.

- 39 Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O’Sullivan, and Hoang D Nguyen. Multi-agent collaboration mechanisms: A survey of llms. *arXiv preprint*, 2025. doi:10.48550/arXiv.2501.06322.
- 40 Yaoshu Wang, Jianbin Qin, and Wei Wang. Efficient approximate entity matching using jaro-winkler distance. In *International conference on web information systems engineering*, pages 231–239. Springer, 2017. doi:10.1007/978-3-319-68783-4_16.
- 41 Jacob O Wobbrock, Leah Findlater, Darren Gergle, and James J Higgins. The aligned rank transform for nonparametric factorial analyses using only anova procedures. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 143–146, 2011. doi:10.1145/1978942.1978963.
- 42 Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. *Experimentation in Software Engineering*. Springer Science & Business Media, 2012. doi:10.1007/978-3-642-29044-2.
- 43 Yisen Xu. Muarf: Leveraging multi-agent workflows for automated code refactoring. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 226–227. IEEE, 2025. doi:10.1109/ICSE-COMPANION66252.2025.00071.
- 44 Yisen Xu, Feng Lin, Jinqiu Yang, Nikolaos Tsantalis, et al. Mantra: Enhancing automated method-level refactoring with contextual rag and multi-agent llm collaboration. *arXiv preprint*, 2025. arXiv:2503.14340.
- 45 Bingyu Yan, Zhibo Zhou, Litian Zhang, Lian Zhang, Ziyi Zhou, Dezhuang Miao, Zhoujun Li, Chaozhuo Li, and Xiaoming Zhang. Beyond self-talk: A communication-centric survey of llm-based multi-agent systems. *arXiv preprint*, 2025. arXiv:2502.14321.
- 46 Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*, 2022.
- 47 Yueheng Zhu, Chao Liu, Xuan He, Xiaoxue Ren, Zhongxin Liu, Ruwei Pan, and Hongyu Zhang. Adacoder: An adaptive planning and multi-agent framework for function-level code generation. *IEEE Transactions on Software Engineering*, 2025.