REALMS-CG: RESILIENT ENSEMBLE OF AUTONOMOUS LLM MULTI-AGENT WITH SPECTRAL CLUSTERING FOR CODE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language model (LLM) multi-agent systems are often built on the assumption that decomposing a task across multiple agents improves reliability. However, when one of the intermediate agents produces an error, it can propagate across stages rather than being corrected. We study this as *correlated failure*: the propagation of error across successive agent stages. We introduce Resilient Ensemble Of Autonomous LLM Multi-agent with Spectral clustering for Code Generation (REALMS-CG), a heterogeneous multi-LLM code-generation framework that combines model diversity with structure-aware candidate selection. REALMS-CG is implemented as a fully local pipeline using five compact LLMs ranging from 8B to 20B parameters, requiring less than 16 GB of memory each. REALMS-CG embeds each LLM output into a semantic similarity graph and applies spectral clustering to reduce redundant output before downstream processing. We evaluated five configurations; single agent, homogeneous multi-agent generation, a Mixture-of-Agents (MoA)-style approach, and two REALMS-CG variants on 378 MBPP+ and 342 LiveCodeBench tasks. Beyond Pass@1, we traced failures across multiple agent stages and compared resource efficiency and available context headroom using peak stage-average input-token count and runtime. REALMS-CG with an LLM final aggregator achieved the highest Pass@1 scores of 91.50% on MBPP+ and 43.27% on LiveCodeBench.

1 INTRODUCTION

Large language model (LLM) are widely used for automated code generation through systems such as ChatGPT, Claude, Copilot, and Gemini (OpenAI, 2022; Anthropic, 2024; GitHub, 2025; Google DeepMind, 2025). Multi-agent systems extend this capability through task decomposition and role-based collaboration (Wang et al., 2024; Idrisov et al., 2025; Liu et al., 2025), but their stages are not independent: an omitted constraint or incorrect interpretation can be inherited by successive agents. We study this behaviour as *correlated failure*, in which an error propagates across successive stages instead of being corrected.

A recent state-of-the-art (SOTA) heterogeneous approach has demonstrated that the output of multiple LLMs can be combined to improve the final output quality. For example, on standardised benchmarks, using a Mixture-of-Agents (MoA) approach where multiple proposer models pass their outputs to subsequent agents and a final aggregation layer to select the final output has shown that model diversity can achieve improved performance (Wang et al., 2025).

However, in the MoA approach, simply forwarding increasingly large collections of intermediate responses introduces two issues for multi-stage code generation. First, semantically similar outputs could rapidly increase the LLM context required and redundant information may be provided to an agent multiple times. Second, simply concatenating outputs could cause the amount of context processed by subsequent agents to grow rapidly with the number of models and stages, resulting in unsustainable growth for the resources required and limiting the scalability of such approach.

To address these issues, we propose REALMS-CG, a resource-efficient heterogeneous multi-LLM, multi-agent framework that treats intermediate-output selection as a structure-aware inference prob-

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

lem. Additionally, we identify and analyse the types of correlated failures, performance and scalability of existing multi-agent approaches, MoA and REALMS-CG.

We compared five configurations under standardised prompts and inference settings: a single agent baseline, a homogeneous multi-agent pipeline, a heterogeneous MoA-style approach, REALMS-CG with spectral clustering as the final selection mechanism, and REALMS-CG with an LLM final aggregator. Our experiments use MBPP+ (Liu et al., 2023) and LiveCodeBench (Naman Jain, 2024) with LiveCodeBench providing a freshness control benchmark for evaluating models whose knowledge cut-offs precede the evaluated problems. Beyond overall correctness, we trace intermediate outputs across agent stages to identify how correlated failures are introduced, propagated, or corrected throughout the pipeline. We also compare resource efficiency under the evaluated five-model, three-stage configuration by measuring peak stage-average input-token count and runtime to quantify the additional resources required.

Our contributions are:

- **Resource-efficient multi-LLM, multi-agent approach.** We introduce REALMS-CG, which combines heterogeneous LLM generation with structure-aware intermediate candidate reduction to prevent semantically redundant candidates from dominating downstream agent context and vastly improving the scalability of heterogeneous multi-LLM, multi-agent systems.
- **Direct characterisation of correlated failures.** We classify and provide trace-based evidence of correlated failures rather than inferring solely from aggregate accuracy or performance. Our analysis identifies recurring failure types and shows how incorrect interpretations, tests, and implementations can lead to an erroneous final output.
- **Failure-recovery and resource-efficiency analysis** We study how correlated failures are preserved or recovered by REALMS-CG and measure input-token consumption and runtime to quantify the trade-off between multi-agent diversity and computational overhead.

2 Related Work

LLM-based programming assistants are now commonly used to support code generation, testing, defect detection, and software recommendation (Xia et al., 2022; Xu et al., 2022; Yetistiren et al., 2022; Durrani et al., 2024). However, developers frequently reject or substantially modify generated suggestions because of correctness, controllability, and requirement-alignment problems (Chakraborty et al., 2020; Vaithilingam et al., 2022; Liang et al., 2024). Generated code also remains susceptible to hallucination, bias, and limited interpretability (Tang et al., 2023; Liu et al., 2024; Shree et al., 2024), while iterative prompting may not resolve deeper reasoning errors (Liu et al., 2024). This suggests that while LLM generate plausible code, outputs often lack reliability and require verification.

Multi-agent approaches attempt to improve reliability by assigning specialised roles for planning, generation, testing, and verification (Lu et al., 2024; Mathews & Nagappan, 2024; Qian et al., 2024; Wang et al., 2024; Abbasnejad et al., 2025; Idrisov et al., 2025; Liu et al., 2025; Zhu et al., 2026). Although these workflows provide structured reasoning, their dependent stages create opportunities for errors to accumulate. For example, Li et al. (2026) found that errors introduced by an agentic planner could affect subsequent execution and prevent the workflow from outperforming a non-agentic model.

Alternative approaches have explored model ensembling and parameter-level fusion to improve performance and robustness (Matena & Raffel, 2022; Chen et al., 2025) but generally operate during training or post-training and do not manage intermediate outputs among heterogeneous models at inference time. In contrast, Wang et al. (2025) combine heterogeneous proposer outputs through successive aggregation layers, demonstrating the value of model diversity through a Mixture-of-Agent (MoA) approach.

However, the MoA approach relies primarily on aggregating and forwarding the outputs produced by the previous layer. As the number of models and aggregation stages increases, the amount of intermediate information passed to subsequent agents can grow rapidly as each stage must process output generated by multiple models from the preceding stage. Repeated concatenation can therefore

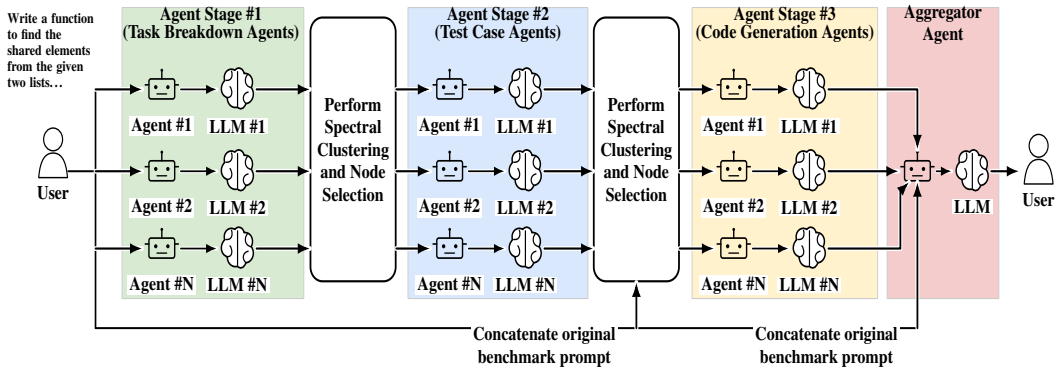


Figure 1: Overview of REALMS-CG using LLM as the final aggregator. After each agent stage, output are embedded into vector space using an embedding model. The outputs are represented as a node in a $N \times N$ matrix and spectral clustering is applied to determine which node should be forwarded to successive agent. A LLM is used to aggregate and return the final output to user.

result in multiplicative context growth, potentially approaching exponential growth as the number of models and stages increases.

This places increasing demands on the context window, token processing, memory, and resource requirements for downstream agents, limiting the scalability for deeper or wider multi-agent pipelines. Furthermore, semantically similar or redundant output may continue to be forwarded between stages, increasing these resource requirements without necessarily introducing additional useful information. Additionally, MoA does not explicitly examine how an incorrect intermediate output may influence later stages or identify correlated failures across a multi-stage code-generation workflow.

These limitations motivate the design of REALMS-CG, which combines heterogeneous LLMs with a structure-aware mechanism to manage intermediate outputs across multiple agent stages. While model diversity provides diverse reasoning perspectives, REALMS-CG further considers how these outputs should be organised and selectively forwarded to downstream agents to reduce redundancy and limit the propagation of correlated errors.

3 REALMS-CG Approach

Illustrated in Figure 1, REALMS-CG is constructed using heterogeneous LLMs. The same set of models is used in each agent stage, comprising the Task Breakdown, Test Case, and Code Generation stages. Similar to Wang et al. (2025), REALMS-CG leverages heterogeneous LLMs for diverse intermediate responses. The final aggregation stage is used to return one final output to the user.

Step 1—Generate alternatives. The standardised benchmark prompt is provided to each task breakdown agent. Each agent produces a natural-language summary of the task and multiple atomic subtasks which each contains the assumptions, requirements, constraints, expected inputs and outputs, acceptance criteria. Collectively, the models provide several alternative reasoning paths.

Step 2—Build the similarity graph. Each output is embedded into a common vector space, the similarity between every pair of outputs is calculated using Equation 1, and the resulting matrix $\mathbf{S}$ is converted into the normalised graph Laplacian in Equation 2. This graph reveals which outputs are semantically similar interpretations, even when their wording differs.

The unit represented by a graph node depends on the agent stage. At the task breakdown stage, each individual subtask is represented as one node. At the test case stage, each generated test case is represented as one node. At the code selection stage, each complete code candidate is represented as one node.

Step 3—Find groups of similar outputs. REALMS-CG decomposes the Laplacian using Equation 3, determines the number of groups from the eigengap in Equation 4, constructs the spectral representation in Equation 5, and applies k-means to assign each output to one cluster. Each distinct

cluster prevents over-representation of a large group of near-duplicate outputs compared to smaller alternative.

Step 4—Select representatives. Within each cluster, Equation 6 selects the medoid which is the actual response that is most representative to the other responses in that cluster. Equation 7 then determines the number q_c of additional candidates to retain, so that each cluster contributes its medoid plus q_c candidates. This preserves representation from every cluster while controlling how much to forward to the next stage.

Step 5—Forward to next agent. The selected medoid and additional candidates are forwarded to every test case agent. Each test case agent produces concrete test-case candidates containing test inputs, expected outputs, requirements, edge cases and the corresponding acceptance criterion. The generated test cases then repeat Step 2–4 and original benchmark prompt are supplied to the code generation agents.

Each code generation agent returns one complete runnable code candidate that follows the required input/output format or function signature and is intended to satisfy the supplied tests. All code generation agent output and the original benchmark prompt are directly forwarded to the aggregator agent which returns the best candidates output to the user.

3.1 Spectral Clustering and Node Selection

Using the output produced from each agent stage, each output is embedded into a shared vector space using the Qwen3-Embedding-8B model (Zhang et al., 2025), allowing semantically similar output to be represented closely in the embedding space. The normalised semantic similarity between two output vector embeddings a and b is then computed using cosine similarity, defined as:

$$\text{cos_similarity}_{\text{norm}}(\mathbf{a}, \mathbf{b}) = \frac{\text{cos_similarity}(\mathbf{a}, \mathbf{b}) + 1}{2} \tag{1}$$

The pairwise similarities form the similarity matrix $\mathbf{S} \in \mathbb{R}^{N \times N}$. Each candidate output is represented as a graph node and S_{ij} represents the weighted edge between candidates i and j. The normalised graph Laplacian is defined in equation 2, where I denotes an identity matrix, S denotes the normalised similarity matrix and D is the corresponding degree matrix.

$$\mathbf{L}_{\text{norm}} = \mathbf{I} - \mathbf{D}^{-\frac{1}{2}}\mathbf{S}\mathbf{D}^{-\frac{1}{2}}, D_{ii} = \sum_{j=1}^{N} S_{ij} \tag{2}$$

Spectral clustering is then performed by eigendecomposing the normalised Laplacian:

$$\mathbf{L}_{\text{norm}} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^{\top} \tag{3}$$

where $\mathbf{V} = [\mathbf{v}_1, \ldots, \mathbf{v}_N]$ contains the orthonormal eigenvector matrix and $\mathbf{\Lambda} = \text{diag}(\lambda_1, \ldots, \lambda_N)$ contains the corresponding eigenvalues in ascending order. Small eigenvalues capture the dominant connectivity structure of the similarity graph and therefore expose groups of semantically related candidate outputs.

The first eigenvector of the normalised graph Laplacian is the trivial eigenvector associated with its smallest eigenvalue. Selecting $k = 1$ would assign every candidate to a single cluster and would therefore provide no non-trivial separation between alternative semantic groups.

Because the clustering stage is intended to identify and preserve distinct candidate groups, REALMS-CG uses a minimum cluster count of $k_{\min} = 2$. The number of clusters is then determined adaptively by finding the largest separation between consecutive eigenvalues over the admissible range. Appendix A.1 provide a conceptual illustration.

$$k = \underset{i \in \{2, \ldots, N-1\}}{\arg\max} \; (\lambda_{i+1} - \lambda_i)\,. \tag{4}$$

Each row $\mathbf{U_0}[i,:]$ maps the i-th candidate into a k-dimensional spectral space. Row normalisation makes clustering depend on direction rather than magnitude, as illustrated in Appendix A.2. The resulting embedding $\mathbf{U}$ is partitioned using k-means with deterministic farthest-first initialisation, which iteratively selects the point maximising its minimum distance from existing centres.

$$\mathbf{U}_0 = [\mathbf{v}_1 \mid \mathbf{v}_2 \mid \cdots \mid \mathbf{v}_k] \in \mathbb{R}^{N \times k}. \tag{5}$$

3.2 Medoid-Based Candidate Forwarding

REALMS-CG selects representative outputs from each cluster rather than forwarding every generated output to the next agent stage. For each cluster C_c, its medoid m_c is defined as the actual candidate having the greatest aggregate semantic similarity to all other members of the same cluster:

$$m_c = \arg\max_{i \in C_c} \sum_{\substack{j \in C_c \\ j \neq i}} S_{ij}. \tag{6}$$

Several configurations using a configurable proportion $x\%$ of the candidates from each cluster were tested to determine whether selecting a few nodes closer or further from the medoid would result in a difference. For a cluster C_c containing $|C_c|$ candidates, the number of additional candidates considered for forwarding is:

$$q_c = \min\left\{|C_c| - 1,\ \left\lceil \frac{x}{100}|C_c| \right\rceil \right\}. \tag{7}$$

As shown in Appendix A.3, for the *closest-to-medoid* strategy, candidates are ranked according to their semantic similarity to the medoid. For the *furthest-from-medoid* ablation, the ordering is reversed and the q_c candidates least similar to the medoid are retained.

Hence, each cluster contributes its medoid together with a controlled proportion of nearby or distant candidates. The closest-to-medoid strategy favours candidates that are representative of the dominant semantic pattern within each cluster, whereas the furthest-from-medoid strategy is evaluated as an ablation to determine whether retaining greater intra-cluster diversity can improve downstream performance. In the experiments, value $x \in \{10, 30, 50\}$ are evaluated. This bounds the amount of intermediate information propagated through the pipeline while retaining representation from every identified semantic cluster.

4 Experiment

4.1 Requirements and Constraints

To support reproducibility on comparatively affordable hardware, all tools and frameworks used in the experiments are publicly available under open-source licences, and the software components are deployed in Docker containers to provide consistent execution environments across machines. The LLM model pool consists only of widely available, general-purpose open-source LLMs with no benchmark or domain-specific fine-tuning and each model requires less than 16 GB of memory.

4.2 Hardware

All experiments were conducted locally using two NVIDIA DGX Spark systems (NVIDIA, 2025). Each DGX Spark node is designed as a compact GPU AI workstation and integrates the NVIDIA GB10 Grace Blackwell GPU. Each system provides 128 GB of unified memory and 1 TB of storage, enabling efficient execution of large-scale local inference workloads. Local execution on DGX Spark provides a consistent and isolated environment for all experimental configurations while avoiding dependencies on external cloud services.

Table 1: LLMs used in the experiments.

Model	Provider	Params.	Memory	Cut-off
GPT-OSS	OpenAI	20B	14.0 GB	Jun. 2024
Phi 4	Microsoft	14B	9.1 GB	Jun. 2024
Gemma 3	Google	12B	8.1 GB	Aug. 2024
Granite 3.3	IBM	8B	4.9 GB	Apr. 2024
Llama 3.1	Meta	8B	4.9 GB	Dec. 2023

Table 2: Standardised inference parameters.

Parameter	Value
Temperature	0.2
Top-p	0.95
Context length	16,000
Maximum output	8,192
Seed	42

4.3 Software

The experiments are conducted using a fully local and containerised software environment. NVIDIA DGX OS version 7.2.3 is used as the host operating system, while Docker version 28.5.1 ensures that container services are easily reproducible. The multi-agent workflows are implemented and orchestrated using n8n version 2.0.3, which manages the execution of individual agents and the transfer of intermediate outputs between stages. Ollama version 0.13.2 is used as the local inference backend for hosting and executing the evaluated LLMs. Milvus version 3.5.18 is used to store and process vector embeddings required for semantic similarity analysis and candidate clustering.

4.4 Language Models

Five small-to-medium open-source LLMs (Abdin et al., 2024; Grattafiori et al., 2024; IBM, 2025; OpenAI, 2025; Team, 2025) are selected for the primary experiments. The selected models are general-purpose models from different providers and model families, providing model-level diversity without requiring domain-specific fine-tuning. Each model requires less than 16 GB of memory individually, satisfying the resource constraints defined for the experiments. To ensure a consistent comparison, the same inference parameters are applied across all models and experimental configurations. Tables 1 and 2 summarise the selected LLMs, their knowledge cut-off, and standardised inference parameters. Additionally, for models that support an explicit thinking or reasoning mode, such as GPT-OSS, the thinking capability is disabled or configured to its lowest available setting to minimise additional reasoning overhead and maintain a consistent inference setting across models.

4.5 Standardised Agent Prompts

To ensure a consistent comparison across models and experimental configurations, all agents within the same role were provided with the same standardised prompt structure. The prompts used for the baseline, task breakdown, test case generation, code generation, and aggregation stages are summarised in Appendix A.4.

4.6 Benchmarks

We evaluate all experimental configurations using two code-generation benchmarks: MBPP+ (Liu et al., 2023) and LiveCodeBench (Naman Jain, 2024). The two benchmarks are selected to provide both established functional-correctness evaluation and a freshness-controlled evaluation that reduces the risk of benchmark contamination.

MBPP+ extends the original Mostly Basic Python Problems (MBPP) (Austin et al., 2021) benchmark with substantially more test cases for each programming problem, providing a more rigorous evaluation of functional correctness. We use the MBPP+ dataset containing 378 programming tasks with EvalPlus 0.3.1 for evaluation and 342 LiveCodeBench (LCB) V5–V6 problems.

MBPP+ provides established functional-correctness coverage but may predate model knowledge cut-offs; the selected LiveCodeBench problems were released from October 2024 onwards and provide freshness control. We report Pass@1, peak stage-average input tokens, and runtime. Because every configuration is evaluated on the same tasks, significance is assessed using two-sided exact McNemar tests at $\alpha = 0.05$ with Holm correction across eleven comparisons per benchmark.

Table 3: Pass@1 results on MBPP+.

Model	Baseline	Multi-Agent	MoA-style	Configuration (Forwarding strategy)	REALMS-CG (SC Last Layer)	REALMS-CG (LLM Last Layer)
GPT-OSS 20B	91.00%	90.50%	72.80%	Top 10% closest to medoid	80.40%	**91.50%**
Granite 3.3 8B	68.30%	61.40%		Top 30% closest to medoid	81.20%	89.70%
Llama 3.1 8B	70.90%	54.20%		Top 50% closest to medoid	81.70%	89.20%
Gemma 3 12B	85.40%	83.10%		Top 50% furthest from medoid	83.60%	90.50%
Phi-4 14B	78.00%	82.50%		Top 30% furthest from medoid	81.70%	89.70%
				Top 10% furthest from medoid	82.00%	88.60%

Table 4: Pass@1 results on LiveCodeBench v5+v6.

Model	Baseline	Multi-Agent	MoA-style	Configuration (Forwarding strategy)	REALMS-CG (SC Last Layer)	REALMS-CG (LLM Last Layer)
GPT-OSS 20B	38.89%	36.26%	16.96%	Top 10% closest to medoid	15.78%	**43.27%**
Granite 3.3 8B	2.63%	3.80%		Top 30% closest to medoid	16.08%	42.39%
Llama 3.1 8B	7.31%	4.67%		Top 50% closest to medoid	12.86%	35.67%
Gemma 3 12B	20.18%	19.30%		Top 50% furthest from medoid	14.61%	33.33%
Phi-4 14B	9.06%	9.64%		Top 30% furthest from medoid	17.54%	41.81%
				Top 10% furthest from medoid	15.20%	41.52%

4.7 Experimental Configurations and Ablations

We evaluate five configurations under identical benchmark prompts and inference settings: (1) **Single agent baseline**, in which each model directly generates one solution; (2) **Multi-agent**, a homogeneous pipeline using the same model for task breakdown, test generation, and code generation; (3) **MoA-style**, which concatenates all preceding-stage outputs following Wang et al. (2025); (4) **REALMS-CG (LLM final)**, which clusters between stages and uses an LLM to select the final code; and (5) **REALMS-CG (SC final)**, which treats the final candidates as one cluster and returns its global medoid.

We organise the evaluation around four research questions (RQs):

- **RQ1:** How does REALMS-CG affect functional correctness?
- **RQ2:** How resource-efficient is REALMS-CG compared with MoA-style aggregation?
- **RQ3:** What correlated failures occur, and how do they propagate?
- **RQ4:** How does REALMS-CG mitigate correlated failures?

5 Results and Analysis

5.1 RQ1: How Does REALMS-CG Affect Functional Correctness?

Answer to RQ1. Tables 3 and 4 show that on MBPP+, compared to the baseline, four of the five models perform worse under the homogeneous multi-agent configuration, with only Phi-4 improving from 78.00% to 82.50%. On LiveCodeBench, three of the five homogeneous multi-agent configurations perform worse than their corresponding single-agent baselines, while Granite 3.3 8B and Phi-4 14B show small improvements.

The MoA-style approach achieves only 72.80% on MBPP+ and 16.96% on LCB, remaining below the strongest single-agent baseline on both benchmarks. These findings indicate that model diversity and additional agent stages alone are insufficient, which are consistent with the findings of Li et al. (2026).

With spectral clustering as the final layer, REALMS-CG achieves between 80.40% and 83.60% on MBPP+ and between 12.86% and 17.54% on LCB. More importantly, **REALMS-CG with an LLM as the final layer achieves the highest observed Pass@1 among all evaluated configurations**,

reaching 91.50% on MBPP+ and 43.27% on LCB.These results exceed the MoA-style approach by 18.70 and 26.32 percentage points on MBPP+ and LiveCodeBench, respectively.

At $\alpha = 0.05$, the exact paired McNemar tests also show that REALMS-CG is significantly better than the Granite, Llama, Gemma, and Phi-4 baseline and multi-agent configurations on both benchmarks. Compared with GPT-OSS multi-agent, the improvement is not significant on MBPP+ ($p = 0.5716$) but is significant on LiveCodeBench ($p = 0.00791$). The aggregate test differs only for the latter comparison, for which it gives $p = 0.0611$ because it does not use the matched task outcomes. Tables 7–10 in Appendix A.6 report all aggregate and paired comparisons. These findings show that the evaluated REALMS-CG pipeline, which combines heterogeneous candidate generation, spectral-clustering-based intermediate reduction, medoid-based forwarding, and LLM-based final selection, outperforms most evaluated individual-model, MoA-style, and homogeneous multi-agent configurations. The experiment evaluates these components as a complete pipeline and does not attribute the observed performance improvement to any single component in isolation. The small observed advantage over the strongest GPT-OSS baseline remains statistically inconclusive.

5.2 RQ2: How Resource-Efficient Is REALMS-CG Compared with MoA-Style Aggregation?

Answer to RQ2. As shown in Appendix A.5, REALMS-CG significantly reduces the amount of information propagated between stages through spectral clustering and node selection. Under the evaluated five-model, three-stage configuration and a 16,000-token context limit, the MoA-style approach reaches the context limit, whereas REALMS-CG maintains substantially smaller intermediate inputs.

Taken together with the Pass@1 results, the efficiency measurements show that REALMS-CG improves not only solution quality but also the resource efficiency of the multi-agent pipeline. In particular, the LLM-last-layer variant achieves the strongest Pass@1 performance while requiring substantially fewer input tokens and considerably shorter runtime than the MoA-style approach. This suggests that structure-aware candidate reduction can limit context expansion and computational requirements without sacrificing performance or efficiency.

5.3 RQ3: What Correlated Failures Occur and How Do They Propagate?

Answer to RQ3. We conducted an exploratory thematic analysis of a 10% sample of mixed pass/fail tasks from the homogeneous multi-agent runs. Sampling was balanced across different model-agreement patterns, and each trace was followed through task breakdown, test generation, and code generation. Wrong logic and incomplete or missing implementations were the dominant categories on both benchmarks; LiveCodeBench additionally showed more syntax and function-signature errors. Complete category distributions are reported in Appendix A.7, Appendix A.8 and Appendix A.9.

Two cases illustrate the propagation mechanism. In MBPP+/239, an ambiguous recursive stopping condition led to incorrect expected counts and an off-by-one implementation. In LiveCodeBench task abc400_a, failure to preserve the in-range non-divisor branch led to tests that omitted this case and code that applied unchecked floor division. The complete model-by-model traces are provided in Appendix A.10.

5.4 RQ4: How Does REALMS-CG Mitigate Correlated Failures?

Answer to RQ4. REALMS-CG mitigates correlated failures by preventing one model's interpretation from becoming the pipeline's only reasoning path. Heterogeneous models generate alternatives, spectral clustering and medoid-based forwarding reduce repetition while retaining distinct candidate groups, and the final aggregator compares the surviving code candidates with the original task. This process does not repair an incorrect reasoning chain but enables a correct alternative to survive and be selected.

REALMS-CG can therefore recover when at least one correct and complete candidate survives the earlier stages. Using task (MBPP+) MBPP+/239 and (LCB) abc400_a as an example, Appendix A.10 shows exactly what was available and what was selected for the two analysed tasks.

For MBPP+/239, recovery occurs because the candidate pool contains a correct interpretation of the rule that the homogeneous pipeline misunderstood. The selected implementation treats n consistently as the required sequence length, counts a sequence only after all n values have been chosen, and considers every next value satisfying the doubling rule. The aggregator can choose this implementation because it also receives the original task.

For abc400_a, the final candidates still differ in completeness. Three candidates contain a correct helper function but do not form a runnable program, while two candidates include standard input and output. The aggregator agent explicitly checks for this requirement and selects a complete candidate. The chosen program therefore returns -1 for $A = 3$, returns 40 for $A = 10$, and prints the result as required.

6 LIMITATIONS

Evaluation covers 720 Python problems from MBPP+ and LiveCodeBench. The findings do not establish performance for other languages, multimodal tasks, multi-file repositories, long-running development, or non-functional requirements such as security, maintainability, and readability.

Results are limited to five 8B–20B models and Qwen3-Embedding-8B for semantic representation. Larger or proprietary models, different model pools or alternative embedding model may result in different findings. Runtime measurements apply to the two DGX Spark systems and stated software stack, limiting generalisation across hardware

Finally, Semantically similar candidates can share errors, and correct but unusual candidates may be discarded. The aggregator cannot recover if all surviving candidates are wrong or incomplete, and may overlook a correct survivor. Repeated runs across broader model pools and programming domains remain necessary.

7 CONCLUSION

This work identifies correlated failure in multi-stage LLM code-generation pipelines and introduces REALMS-CG, which combines heterogeneous generation, structure-aware intermediate reduction, medoid-based forwarding, and final LLM selection. Rather than repairing every failed reasoning chain, the pipeline preserves alternative interpretations and allows a correct surviving candidate to be selected.

Across the evaluated configurations, REALMS-CG with an LLM final layer achieves the highest observed Pass@1: 91.50% on MBPP+ and 43.27% on LiveCodeBench. The exact paired McNemar tests confirm significant improvements over the evaluated MoA-style approach on MBPP+ ($p = 8.38 \times 10^{-16}$) and LiveCodeBench ($p = 2.65 \times 10^{-20}$).

Beyond Pass@1 results, compared with the MoA-style approach, REALMS-CG reduces the peak stage-average input-token count from 16,000 to 893.77 on MBPP+ and to 2,603 on LiveCodeBench. The corresponding peak stage-average runtimes decrease from 1,866 to 126.49 seconds and from 2,532 to 201.19 seconds, respectively.

A manual post-experiment evaluation of six illustrative tasks is presented in Appendix A.11, with separate Task Breakdown and Test Case visualisations for each task. Incorrect test expectations identified by this evaluation are marked with red $\times$ symbols. These annotations were added after the experiments and did not influence the pipeline.

Across the raw-eigenvector views, separation commonly reflects functional roles or output formats, while incorrect expectations can occupy the same semantic group as correct ones. Geometric separation and proximity to a medoid therefore do not provide a reliable correctness criterion.

Future work should investigate methods that automatically detect, characterise, classify, and remove undesirable nodes, including incorrect or insufficiently specified outputs. Such methods could combine spectral structure with task-specific correctness and coverage checks, while preserving valid minority interpretations and useful test coverage.

AI USE STATEMENT

Generative AI tools were used to assist in creating and refining explanatory illustrations, improving the clarity and presentation of selected passages, proofreading, and preparing LaTeX formatting. They were also used to provide feedback on the research methodology and experiments. All AI-generated suggestions were treated as advisory rather than as independent evidence. The experimental measurements, analyses, interpretations, and scientific conclusions were conducted, reviewed and verified by the authors against the underlying data and trace logs.

ETHICS STATEMENT

This work uses publicly available benchmark datasets, open-source language models, and publicly available software exclusively for academic research. No human subjects were involved, and no personal, private, or sensitive data were collected or processed. We used all datasets, models, libraries, and software tools in accordance with their applicable licences, terms of use, and attribution requirements. The experiments were conducted locally, and the study does not release personal information or introduce a new dataset containing user data.

REPRODUCIBILITY STATEMENT

The datasets used in this study (MBPP+ (Liu et al., 2023) and LiveCodeBench (Naman Jain, 2024)) are publicly available and can be accessed from their respective official sources. The source code, prompts, scripts and all experiment results are available at Zenodo (DOI: 10.5281/zenodo.22907479) and Figshare to support transparency and allow verification of the reported findings.

ACKNOWLEDGMENTS

Intentionally left empty for double-blind anonymity.

REFERENCES

Iman Abbasnejad, Xuefeng Liu, and Atunu Roy. Deciding the path: Leveraging multi-agent systems for solving complex tasks. In *2025 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 4216–4225, 2025. doi: 10.1109/CVPRW67362.2025.00405.

Marah Abdin, Jyoti Aneja, Harkirat Behl, Sébastien Bubeck, Ronen Eldan, Suriya Gunasekar, Michael Harrison, Russell J. Hewett, Mojan Javaheripi, Piero Kauffmann, James R. Lee, Yin Tat Lee, Yuanzhi Li, Weishung Liu, Caio C. T. Mendes, Anh Nguyen, Eric Price, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Xin Wang, Rachel Ward, Yue Wu, Dingli Yu, Cyril Zhang, and Yi Zhang. Phi-4 technical report, 2024. URL https://arxiv.org/abs/2412.08905.

Anthropic. Anthropic home. https://www.anthropic.com/, 2024. Accessed: 2025-01-23.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

Joymallya Chakraborty, Suvodeep Majumder, Zhe Yu, and Tim Menzies. Fairway: a way to build fair ml software. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, pp. 654–665, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450370431. doi: 10.1145/3368089.3409697. URL https://doi.org/10.1145/3368089.3409697.

Luming Chen, Ziwei Xiang, Kai Lei, and Xu-Yao Zhang. Multi-task model fusion via adaptive merging. In *ICASSP 2025 - 2025 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 1–5, 2025. doi: 10.1109/ICASSP49660.2025.10887718.

Usman Khan Durrani, Mustafa Akpinar, Muhammed Fatih Adak, Abdullah Talha Kabakus, Muhammed Maruf Öztürk, and Mohammed Saleh. A decade of progress: A systematic literature review on the integration of ai in software engineering phases and activities (2013-2023). *IEEE Access*, 12:171185–171204, 2024. doi: 10.1109/ACCESS.2024.3488904.

GitHub. GitHub Copilot: Your ai pair programmer. https://github.com/features/copilot, 2025. Accessed: 2025-01-23.

Google DeepMind. Gemini. https://deepmind.google/models/gemini/, 2025. Accessed: 2025-01-23.

Aaron Grattafiori et al. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

IBM. Granite 3.3 language models. https://github.com/ibm-granite/granite-3.3-language-models, 2025. IBM Research.

Baskhad Idrisov, Esther Eisenacher, and Tim Schlippe. Program code generation: Single llms vs. multi-agent systems. In *2025 7th International Conference on Natural Language Processing (ICNLP)*, pp. 121–127, 2025. doi: 10.1109/ICNLP65360.2025.11108400.

Chunyang Li, Yilun Zheng, Xinting Huang, Tianqing Fang, Jiahao Xu, Lihui Chen, Yangqiu Song, and Han Hu. Webdevjudge: Evaluating (m)LLMs as critiques for web development quality. In *The Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=CCSPm6V5EF.

Jenny T. Liang, Chenyang Yang, and Brad A. Myers. A large-scale survey on the usability of ai programming assistants: Successes and challenges. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 616–628, 2024.

Aofan Liu, Haoxuan Li, Bin Wang, Ao Yang, and Hui Li. Ra-gen: A controllable code generation framework using react for multi-agent task execution. In *2025 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–9, 2025. doi: 10.1109/IJCNN64981.2025.11228706.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatGPT really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. URL https://openreview.net/forum?id=1qvx610Cu7.

Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. No need to lift a finger anymore? assessing the quality of code generation by chatgpt. *IEEE Transactions on Software Engineering*, 50(6):1548–1584, 2024. doi: 10.1109/TSE.2024.3392499.

Jiawei Lu, Haoye Wang, Zhongxin Liu, Keyu Liang, Lingfeng Bao, and Xiaohu Yang. Instructive code retriever: Learn from large language model's feedback for code intelligence tasks. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pp. 191–203, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712487. doi: 10.1145/3691620.3694997. URL https://doi.org/10.1145/3691620.3694997.

Michael Matena and Colin Raffel. Merging models with fisher-weighted averaging. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc. ISBN 9781713871088.

Noble Saji Mathews and Meiyappan Nagappan. Test-driven development and llm-based code generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pp. 1583–1594, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712487. doi: 10.1145/3691620.3695527. URL https://doi.org/10.1145/3691620.3695527.

Alex Gu Wen-Ding Li Fanjia Yan Tianjun Zhang Sida Wang Armando Solar-Lezama Koushik Sen Ion Stoica Naman Jain, King Han. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint*, 2024.

NVIDIA. NVIDIA DGX Spark. https://www.nvidia.com/en-us/products/workstations/dgx-spark/, 2025. Accessed: 2025-01-23.

OpenAI. Introducing chatgpt. https://openai.com/index/chatgpt/, November 2022. Accessed: 2024-12-1.

OpenAI. gpt-oss-120b gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. Chatdev: Communicative agents for software development, 2024. URL https://arxiv.org/abs/2307.07924.

Sunny Shree, Krishna Khadka, Yu Lei, Raghu N. Kacker, and D. Richard Kuhn. Constructing surrogate models in machine learning using combinatorial testing and active learning. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, pp. 1645–1654, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400712487. doi: 10.1145/3691620.3695532. URL https://doi.org/10.1145/3691620.3695532.

Yutian Tang, Zhijie Liu, Zhichao Zhou, and Xiapu Luo. Chatgpt vs sbst: A comparative assessment of unit test suite generation, 2023. URL https://arxiv.org/abs/2307.00588.

Gemma Team. Gemma 3. 2025. URL https://goo.gle/Gemma3Report.

Priyan Vaithilingam, Tianyi Zhang, and Elena L. Glassman. Expectation vs.nbsp;experience: Evaluating the usability of code generation tools powered by large language models. In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI EA '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391566. doi: 10.1145/3491101.3519665. URL https://doi.org/10.1145/3491101.3519665.

Junlin Wang, Jue WANG, Ben Athiwaratkun, Ce Zhang, and James Zou. Mixture-of-agents enhances large language model capabilities. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=h0ZfDIrj7T.

Luqiao Wang, Yangtao Zhou, Huiying Zhuang, Qingshan Li, Di Cui, Yutong Zhao, and Lu Wang. Unity is strength: Collaborative llm-based agents for code reviewer recommendation. In *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 2235–2239, 2024.

Tianpei Xia, Rui Shu, Xipeng Shen, and Tim Menzies. Sequential model optimization for software effort estimation. *IEEE Transactions on Software Engineering*, 48(6):1994–2009, 2022. doi: 10.1109/TSE.2020.3047072.

Frank F. Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, pp. 1–10, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392730. doi: 10.1145/3520312.3534862. URL https://doi.org/10.1145/3520312.3534862.

Burak Yetistiren, Isik Ozsoy, and Eray Tuzun. Assessing the quality of github copilot's code generation. In *Proceedings of the 18th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE 2022, pp. 62–71, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450398602. doi: 10.1145/3558489.3559072. URL https://doi.org/10.1145/3558489.3559072.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models. *arXiv preprint arXiv:2506.05176*, 2025.

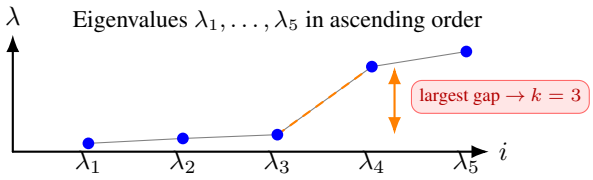


Figure 2: Illustration of the eigengap heuristic for automatic cluster number selection.

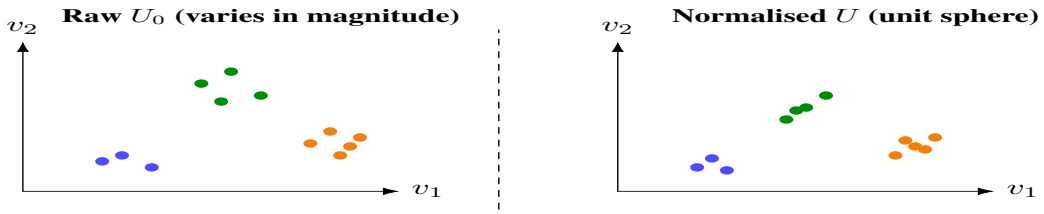


Figure 3: Effect of row normalisation on spectral embeddings. Raw embeddings vary in magnitude, while normalised embeddings form tighter clusters.

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*, 52(2):631–650, 2026. doi: 10.1109/TSE.2025.3642621.

A Appendix

A.1 Eigengap Illustration

Figure 2 illustrates the eigengap heuristic used when the similarity graph does not contain multiple disconnected components. The eigenvalues are ordered from smallest to largest, and the cluster count k is selected at the largest gap between consecutive eigenvalues.

A.2 Row Normalisation On Spectral Embedding

Figure 3 illustrates why each row of the spectral embedding is normalised before applying k-means. In the raw embedding $\mathbf{U}_0$, two candidates can point in similar directions but have different magnitudes, causing Euclidean distance to reflect scale as well as their position in the spectral space. Dividing each non-zero row by its ℓ_2 norm produces the normalised embedding $\mathbf{U}$, in which every row has unit length. Consequently, k-means groups candidates mainly according to direction, which better reflects the connectivity patterns captured by the selected eigenvectors. The figure is conceptual and illustrates the effect of row normalisation rather than the embedding of a particular benchmark task.

A.3 Visualisation of node selection

Figure 4 provides a conceptual visualisation of the medoid-based node-selection strategy used in REALMS-CG. Within each spectral cluster, the medoid represents the most central candidate output, while the remaining candidates are ranked according to their semantic similarity to the medoid. The concentric regions illustrate the different forwarding strategies evaluated in the experiments, including the top 10%, 30%, and 50% of candidates closest to or furthest from the medoid.

The closest-to-medoid strategies favour representative candidates that are semantically similar to the dominant pattern within a cluster, whereas the furthest-from-medoid strategies retain more diverse candidates from the same cluster. These selected candidates are subsequently forwarded to the next agent stage. The figure is intended as a conceptual illustration of the selection process; the physical position of each node does not represent an exact two-dimensional embedding distance.

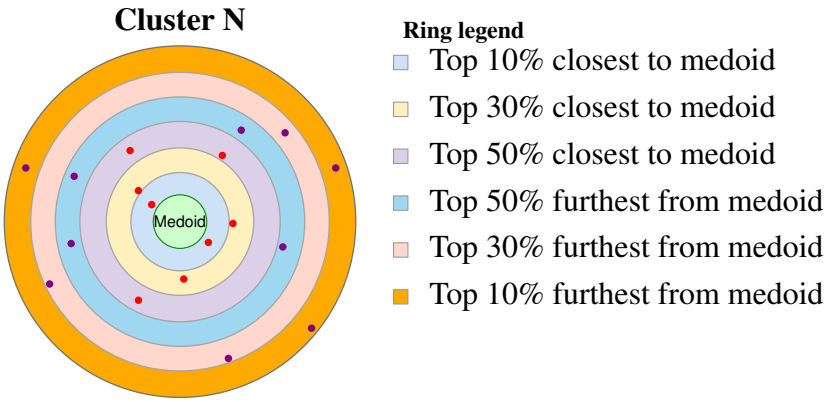


Figure 4: Illustration of medoid-based candidate selection within a spectral cluster. Candidates are ranked according to their semantic distance from the cluster medoid. REALMS-CG forwards the closest or furthest 10%, 30%, and 50% of candidates from each cluster.

Table 5: Standardised agent prompts and information flow across experimental configurations.

Configuration	Task Breakdown Agent	Test Case Agent	Code Generation Agent	Aggregator Agent
Baseline	N.A.	N.A.	**System prompt:** You are a code generator. You will receive … Return a complete runnable solution satisfying the task, required function signature, and supplied test cases. **User input:** Original benchmark prompt.	N.A.
Multi-Agent	**System prompt:** You are a task breakdown agent. You will receive … Convert the programming problem into requirements and subtasks containing assumptions, constraints, inputs, outputs, and acceptance criteria. **User input:** Original benchmark prompt.	**System prompt:** You are a test case agent. You will receive … Generate concrete test cases for each implementation subtask based on its expected inputs, outputs, constraints, and acceptance criteria. **User input:** Task Breakdown Agent output.	**System prompt:** You are a code generator. You will receive … Generate a complete runnable solution using the original task together with the requirements and test information produced by the preceding agents. **User input:** Original benchmark prompt + Test Case Agent output.	N.A.
MoA-style Approach				**System prompt:** You are a code selection agent. You will receive … Select the candidate solution that best matches the programming task. **User input:** Original benchmark prompt + Code Generation output.
REALMS-CG (LLM Last Layer)				
REALMS-CG (Spectral Cluster Last Layer)				N.A.

A.4 Standardised Prompts for the Experiment

The standardised prompts used across the experimental configurations are summarised in Table 5. To ensure a fair comparison, the same role-specific instructions and input structure are maintained for each corresponding agent stage, while only the information available to the agent changes according to the pipeline design. The Task Breakdown Agent converts the original programming problem into structured requirements and subtasks, the Test Case Agent generates concrete tests from these intermediate outputs, and the Code Generation Agent produces the final runnable solution using the original task together with the information propagated from earlier stages.

For configurations that include a final LLM-based aggregation layer, the Code Selection Agent receives the original benchmark prompt and candidate code outputs and selects the solution that best matches the programming task. This standardisation reduces prompt-related variation and allows

Table 6: Peak stage-average input-token count and duration across the MoA-style approach and REALMS-CG configurations.

Benchmark	Metrics (Peak stage-average)	MoA-style	REALMS-CG (SC)	REALMS-CG (SC + LLM)
MBPP+	Input tokens	16,000	788.44	893.77
	Duration	1,866 sec (∼31 min)	268.12 sec (∼4.5 min)	126.49 sec (∼2.1 min)
LiveCodeBench	Input tokens	16,000	1,852	2,603
	Duration	2,532 sec (∼42 min)	431.84 sec (∼7.2 min)	201.19 sec (∼3.4 min)

the observed differences in performance to be attributed more directly to the agent configuration, model diversity, and intermediate-output selection strategy.

A.5 Detailed Resource-Efficiency Results

Table 6 compares the peak stage-average input tokens and processing duration of the MoA-style approach against the two REALMS-CG variants across all agent stages. The MoA-style approach consistently reaches the maximum 16,000-token context length for both MBPP+ and LiveCodeBench, indicating substantial context growth from concatenating and forwarding multiple intermediate outputs across agent stages.

A.6 Statistical Significance Tests

A.6.1 Supplementary Tests Using Aggregate Pass@1

We use a pooled two-proportion z-test to compare REALMS-CG with the individual-model baseline configurations, the homogeneous multi-agent configurations, and the evaluated MoA-style approach. The tested REALMS-CG configuration uses the LLM final layer and forwards the top 10% of candidates closest to each cluster medoid. Let $\widehat{p}_R$ and $\widehat{p}_C$ denote the REALMS-CG and comparator Pass@1 proportions reported in Tables 3 and 4. Because the two systems being compared use the same benchmark and therefore the same assumed sample size N, the pooled proportion, standard error, and test statistic are

$$\widehat{p} = \frac{\widehat{p}_R + \widehat{p}_C}{2}, \qquad \mathrm{SE} = \sqrt{\widehat{p}(1-\widehat{p})\frac{2}{N}}, \qquad z = \frac{\widehat{p}_R - \widehat{p}_C}{\mathrm{SE}}. \tag{8}$$

We test $H_0 : p_R = p_C$ against the two-sided alternative $H_1 : p_R \neq p_C$. The two-sided p-value is $2[1 - \Phi(|z|)]$, where Φ is the cumulative distribution function of the standard normal distribution. REALMS-CG is considered significantly better when its Pass@1 is higher than the comparator and $p < 0.05$. The calculations use the Pass@1 values printed in the paper, with $N = 378$ for MBPP+ and $N = 342$ for LiveCodeBench; no task-level pass/fail outcomes are used.

At $\alpha = 0.05$, REALMS-CG performs significantly better than the Granite, Llama, Gemma, and Phi-4 Baseline and multi-agent configurations and the evaluated MoA-style approach on both benchmarks. Its difference from GPT-OSS is not statistically significant for either the Baseline configuration (MBPP+: $p = 0.8078$; LiveCodeBench: $p = 0.2443$) or the multi-agent configuration (MBPP+: $p = 0.6310$; LiveCodeBench: $p = 0.0611$). These conclusions are unchanged after a Holm correction across the eleven comparisons within each benchmark. Because the same benchmark tasks are used for every configuration, this aggregate proportion test does not model task-level pairing; it treats the reported Pass@1 values as independent binomial proportions.

Table 7: Two-proportion z-tests for MBPP+ ($N = 378$). REALMS-CG uses the LLM final layer and top-10%-closest forwarding strategy with Pass@1 of 91.50%.

Comparator	Comparator Pass@1	REALMS-CG Pass@1	z	Two-sided p	Significant
GPT-OSS (Baseline)	91.00%	91.50%	0.24	0.8078	No
Granite 3.3 (Baseline)	68.30%	91.50%	7.96	1.74×10^{-15}	Yes
Llama 3.1 (Baseline)	70.90%	91.50%	7.25	4.22×10^{-13}	Yes
Gemma 3 (Baseline)	85.40%	91.50%	2.62	0.00870	Yes
Phi-4 (Baseline)	78.00%	91.50%	5.16	2.44×10^{-7}	Yes
GPT-OSS (Multi-Agent)	90.50%	91.50%	0.48	0.6310	No
Granite 3.3 (Multi-Agent)	61.40%	91.50%	9.75	1.80×10^{-22}	Yes
Llama 3.1 (Multi-Agent)	54.20%	91.50%	11.53	9.28×10^{-31}	Yes
Gemma 3 (Multi-Agent)	83.10%	91.50%	3.47	5.24×10^{-4}	Yes
Phi-4 (Multi-Agent)	82.50%	91.50%	3.68	2.34×10^{-4}	Yes
MoA-style	72.80%	91.50%	6.71	1.90×10^{-11}	Yes

Table 8: Two-proportion z-tests for LiveCodeBench ($N = 342$). REALMS-CG uses the LLM final layer and top-10%-closest forwarding strategy with Pass@1 of 43.27%.

Comparator	Comparator Pass@1	REALMS-CG Pass@1	z	Two-sided p	Significant
GPT-OSS (Baseline)	38.89%	43.27%	1.16	0.2443	No
Granite 3.3 (Baseline)	2.63%	43.27%	12.64	1.31×10^{-36}	Yes
Llama 3.1 (Baseline)	7.31%	43.27%	10.82	2.82×10^{-27}	Yes
Gemma 3 (Baseline)	20.18%	43.27%	6.49	8.72×10^{-11}	Yes
Phi-4 (Baseline)	9.06%	43.27%	10.18	2.49×10^{-24}	Yes
GPT-OSS (Multi-Agent)	36.26%	43.27%	1.87	0.0611	No
Granite 3.3 (Multi-Agent)	3.80%	43.27%	12.17	4.67×10^{-34}	Yes
Llama 3.1 (Multi-Agent)	4.67%	43.27%	11.82	2.94×10^{-32}	Yes
Gemma 3 (Multi-Agent)	19.30%	43.27%	6.76	1.38×10^{-11}	Yes
Phi-4 (Multi-Agent)	9.64%	43.27%	9.97	2.06×10^{-23}	Yes
MoA-style	16.96%	43.27%	7.50	6.40×10^{-14}	Yes

A.6.2 Tests Using Paired Per-Task Pass/Fail Outcomes

The aggregate Pass@1 test does not use information about whether the two systems pass or fail the *same* task. We therefore additionally use an exact McNemar test on the paired task-level outcomes. For each comparison, a is the number of tasks passed by both systems, b is the number passed only by REALMS-CG, c is the number passed only by the comparator, and d is the number failed by both systems. The paired outcome table is

	Comparator pass	Comparator fail
REALMS-CG pass	a	b
REALMS-CG fail	c	d

. (9)

Under the null hypothesis, either system is equally likely to be the successful system among the $b + c$ discordant tasks. Consequently, if $X \sim \mathrm{Binomial}(b + c, 0.5)$, the two-sided exact McNemar p-value is

$$p_{\mathrm{McN}} = \min\left\{1, 2\sum_{j=0}^{\min(b,c)} \binom{b+c}{j} 2^{-(b+c)}\right\}. \tag{10}$$

This test uses only discordant task pairs because tasks passed or failed by both systems provide no evidence about which system performs better. For MBPP+, a pass is determined from each task's base-test result. For LiveCodeBench, the per-task Pass@1 indicator is used. The same $\alpha = 0.05$ threshold is applied.

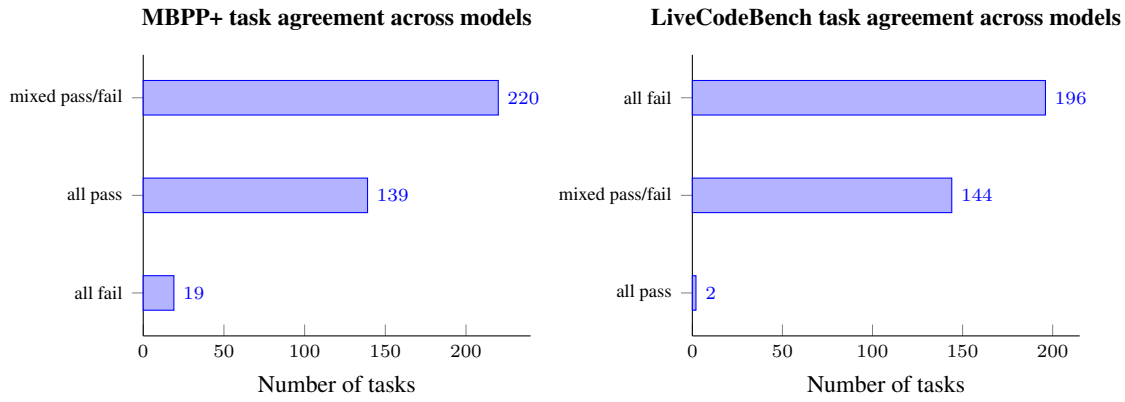
Table 9: Exact paired McNemar tests for MBPP+ ($N = 378$). The cells a, b, c, and d follow Equation 9. REALMS-CG uses the LLM final layer and top-10%-closest forwarding strategy.

Comparator	a	b	c	d	**Two-sided exact p**	**Significant**
GPT-OSS (Baseline)	331	15	13	19	0.8506	No
Granite 3.3 (Baseline)	250	96	8	24	2.77×10^{-20}	Yes
Llama 3.1 (Baseline)	262	84	6	26	1.08×10^{-18}	Yes
Gemma 3 (Baseline)	309	37	14	18	0.00177	Yes
Phi-4 (Baseline)	288	58	7	25	4.27×10^{-11}	Yes
GPT-OSS (Multi-Agent)	330	16	12	20	0.5716	No
Granite 3.3 (Multi-Agent)	229	117	3	29	4.33×10^{-31}	Yes
Llama 3.1 (Multi-Agent)	200	146	5	27	4.44×10^{-37}	Yes
Gemma 3 (Multi-Agent)	309	37	5	27	4.43×10^{-7}	Yes
Phi-4 (Multi-Agent)	303	43	9	23	2.04×10^{-6}	Yes
MoA-style	267	79	8	24	8.38×10^{-16}	Yes

Table 10: Exact paired McNemar tests for LiveCodeBench ($N = 342$). The cells a, b, c, and d follow Equation 9. REALMS-CG uses the LLM final layer and top-10%-closest forwarding strategy.

Comparator	a	b	c	d	**Two-sided exact p**	**Significant**
GPT-OSS (Baseline)	104	44	29	165	0.1006	No
Granite 3.3 (Baseline)	9	139	0	194	2.87×10^{-42}	Yes
Llama 3.1 (Baseline)	22	126	3	191	1.05×10^{-33}	Yes
Gemma 3 (Baseline)	55	93	14	180	1.76×10^{-15}	Yes
Phi-4 (Baseline)	28	120	3	191	5.84×10^{-32}	Yes
GPT-OSS (Multi-Agent)	98	50	26	168	0.00791	Yes
Granite 3.3 (Multi-Agent)	13	135	0	194	4.59×10^{-41}	Yes
Llama 3.1 (Multi-Agent)	15	133	1	193	1.24×10^{-38}	Yes
Gemma 3 (Multi-Agent)	55	93	11	183	2.49×10^{-17}	Yes
Phi-4 (Multi-Agent)	28	120	5	189	1.15×10^{-29}	Yes
MoA-style	49	99	9	185	2.65×10^{-20}	Yes

The paired results agree with the aggregate Pass@1 tests for all MBPP+ comparisons: REALMS-CG is not significantly better than the GPT-OSS baseline or GPT-OSS multi-agent configuration, but it is significantly better than every other comparator. On LiveCodeBench, both tests find no significant improvement over the GPT-OSS Baseline and significant improvements over the remaining Baseline models and MoA-style approach. The only difference is the GPT-OSS multi-agent comparison. The aggregate test gives $p = 0.0611$, whereas the paired test identifies 50 tasks passed only by REALMS-CG and 26 passed only by GPT-OSS multi-agent, giving $p = 0.00791$. This result is significant because the paired test uses the direction of the disagreements on the same tasks. The paired-test conclusions are unchanged after applying a Holm correction across the eleven comparisons within each benchmark. Because the task-level outcomes are available and matched, the paired McNemar results provide the more appropriate primary significance analysis; the aggregate tests are retained to show the result obtained from the reported Pass@1 proportions alone.



(a) MBPP+ ($N = 378$). (b) LiveCodeBench v5+v6 ($N = 342$).

Figure 5: Distribution of task-level agreement among the five models under the multi-agent configuration. Tasks are grouped into cases where all models passed, all models failed, or a mixture of pass and fail outcomes was observed. The mixed pass/fail tasks form the population from which the 10% random sample was selected for correlated-failure analysis.

Table 11: Correlated failure categories identified from the sampled MBPP+ tasks.

No.	Failure Type	Occurrences	Percentage
1	Wrong logic	11	28.95%
2	Incomplete / missing function body	9	23.68%
3	Requirements mismatch	4	10.53%
4	Wrong input / input-type handling	3	7.89%
5	Wrong function name	2	5.26%
6	Over-restrictive validation	2	5.26%
7	Function signature mismatch	1	2.63%
8	Output mismatch	1	2.63%
9	Too much wrapper	1	2.63%
10	Missing import	1	2.63%
11	Invalid validation type	1	2.63%
12	Undefined helper function	1	2.63%
13	Improper global variable	1	2.63%
	Total	**38**	**100%**

A.7 Task agreement across models for MBPP+ and LiveCodeBench

Figure 5 shows the overall distribution of task-level agreement among the five models under the multi-agent configuration. Tasks are divided into three groups: all models pass, all models fail, and mixed pass/fail outcomes. The mixed category is particularly important for the correlated-failure analysis because it contains tasks where the same pipeline produces different outcomes across models.

For MBPP+, 220 of the 378 tasks exhibit mixed pass/fail outcomes, compared with 139 all-pass and 19 all-fail tasks. For LiveCodeBench, 144 of the 342 tasks exhibit mixed outcomes, while 196 are all-fail and only two are all-pass. A 10% random sample was drawn from the mixed-outcome tasks, with an approximately even distribution across cases involving different numbers of passing models, to support the detailed failure analysis reported above.

A.8 MBPP+ Failure Categories

Table 11 summarises the observed failure categories, their frequencies and percentages of the 38 recorded failure occurrences. The most common failures were wrong logic and incomplete or missing function bodies, while the remaining categories represent less frequent implementation, interface, and validation-related errors.

Table 12: Correlated failure categories identified from the sampled LiveCodeBench tasks.

No.	Failure Type	Occurrences	Percentage
1	Wrong logic	10	20.83%
2	Incomplete / missing function body	10	20.83%
3	Syntax error	6	12.50%
4	Function signature mismatch	4	8.33%
5	Missing imports	3	6.25%
6	Requirements mismatch	2	4.16%
7	Unused variables	2	4.16%
8	Invalid validation	2	4.16%
9	Output mismatch	2	4.16%
10	Invalid indentation	2	4.16%
11	Wrong function name	1	2.08%
12	Wrong code placement	1	2.08%
	Total	**48**	**100%**

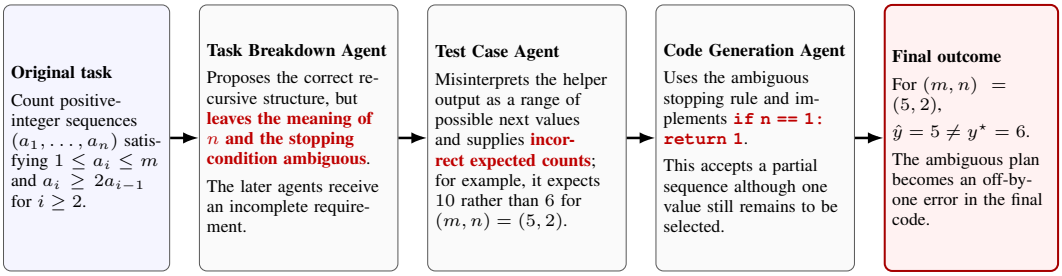
A.9 LiveCodeBench Failure Categories

Table 12 reports the 48 observed failure occurrences. Wrong logic and incomplete or missing function bodies were again the most common categories, while syntax errors and function-signature mismatches occurred more frequently than in MBPP+. This reflects the greater implementation complexity of the LiveCodeBench tasks.

A.10 Correlated Failure and Recovery Illustrations

Figures 6 and 7 provide detailed illustrations of the two case studies discussed in the correlated failure analysis. Figure 6 shows how an early ambiguity or omission can pass from task breakdown to test generation and then become an incorrect final implementation. Figure 7 contrasts these homogeneous failures with REALMS-CG: different model paths preserve alternative solutions, clustering reduces repeated candidates, and the final aggregator compares the remaining programs with the original task before selecting a correct and complete answer.

Example 1: MBPP+/239 — wrong logic



Example 2: LiveCodeBench `abc400_a` — wrong logic

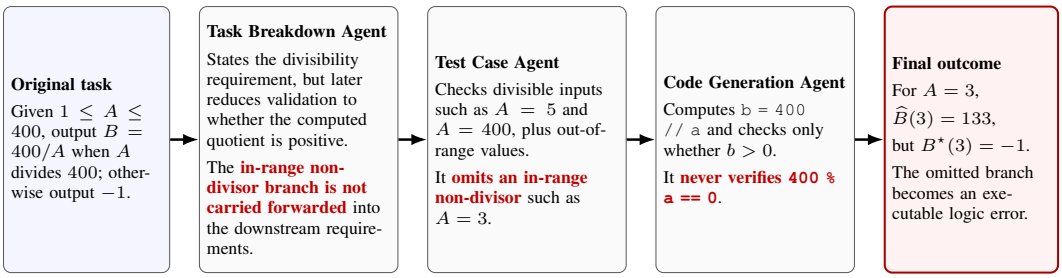


Figure 6: Examples of correlated failure propagation in the homogeneous multi-agent pipeline. In MBPP+/239, an ambiguous recursive stopping condition is accompanied by incorrect tests and becomes an off-by-one error in the generated code. In LiveCodeBench task abc400_a, the task breakdown agent does not preserve the in-range non-divisor branch, the tests omit that case, and the generated code applies unchecked floor division. In both examples, the stage dependency allows an earlier omission to survive to the final answer.

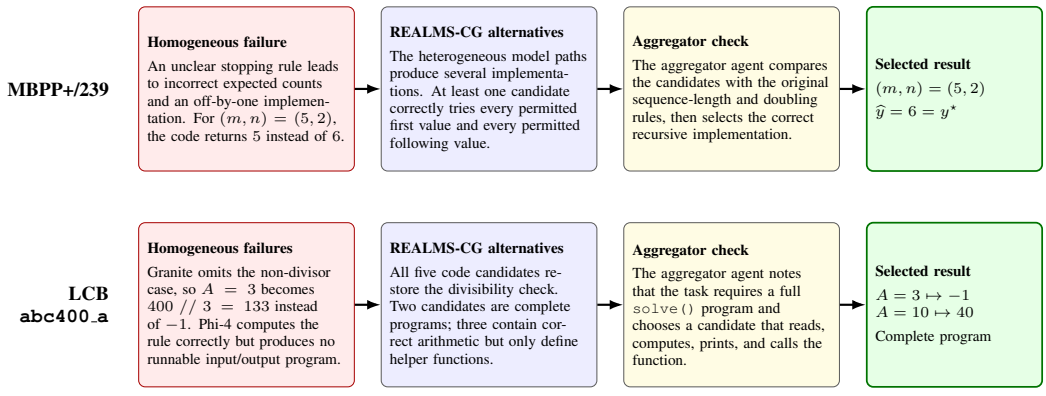


Figure 7: How REALMS-CG recovers from the correlated failures observed in MBPP+/239 and LiveCodeBench task abc400_a. Each row gives the failed homogeneous path, the alternatives available under REALMS-CG, the final aggregator's check, and the selected result. The stage labels make the interpretation independent of colour.

A.11 POST-EXPERIMENT SPECTRAL VISUALISATIONS

Figures 8–13 show the saved spectral-clustering results for LiveCodeBench tasks `abc400_a`, `3764`, and `abc374_a`, and MBPP+ tasks 239, 259, and 59. The twelve stage visualisations are grouped into six figures, with each task presented in two parts under the same figure number: Task Breakdown followed by Test Case on a separate page. The upper-left panel shows the non-trivial Laplacian eigenvalues; the upper-right panel shows the raw eigenvector coordinates (v_2, v_3); the lower-left panel shows the full retained row-normalised spectral embedding; and the lower-right panel shows the reconstructed similarity matrix. In the raw-eigenvector panels, colours indicate the saved clusters, marker shapes identify the generating models, and black stars mark the medoids. Cluster identifiers are local to each stage and do not imply correspondence between stages.

A manual post-experiment evaluation, supported by task-specific checks, identified generated test cases whose expected outputs contradict the relevant task or helper-function specification. These nodes are marked with red $\times$ symbols. The annotations were applied after the experiments and did not influence clustering, forwarding, or final selection. Unmarked nodes should not all be interpreted as verified correct: some contain incomplete expected outputs or insufficient information for a conclusive assessment. The red line segment in each eigenvalue panel instead marks the eigengap used to select the cluster count. The similarity matrix labelled $\mathbf{W}$ in the figures corresponds to $\mathbf{S}$ in Equation 2.

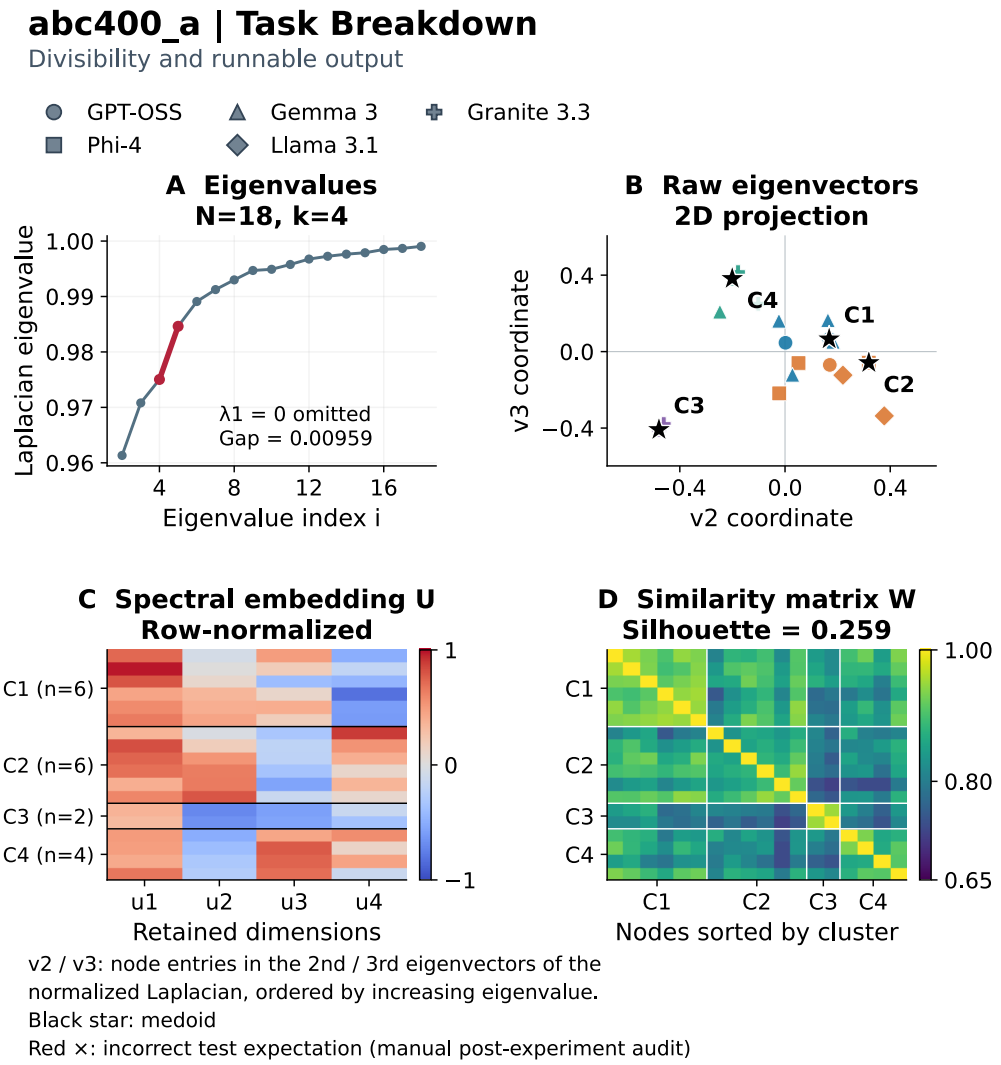


Figure 8: Spectral visualisation of LiveCodeBench task `abc400_a`: Task Breakdown stage. The four clusters mix divisibility and quotient calculation, decision and output logic, positive-integer validation, and examples or input setup. Different model outputs overlap, and cluster membership reflects semantic concerns rather than a correctness classification.

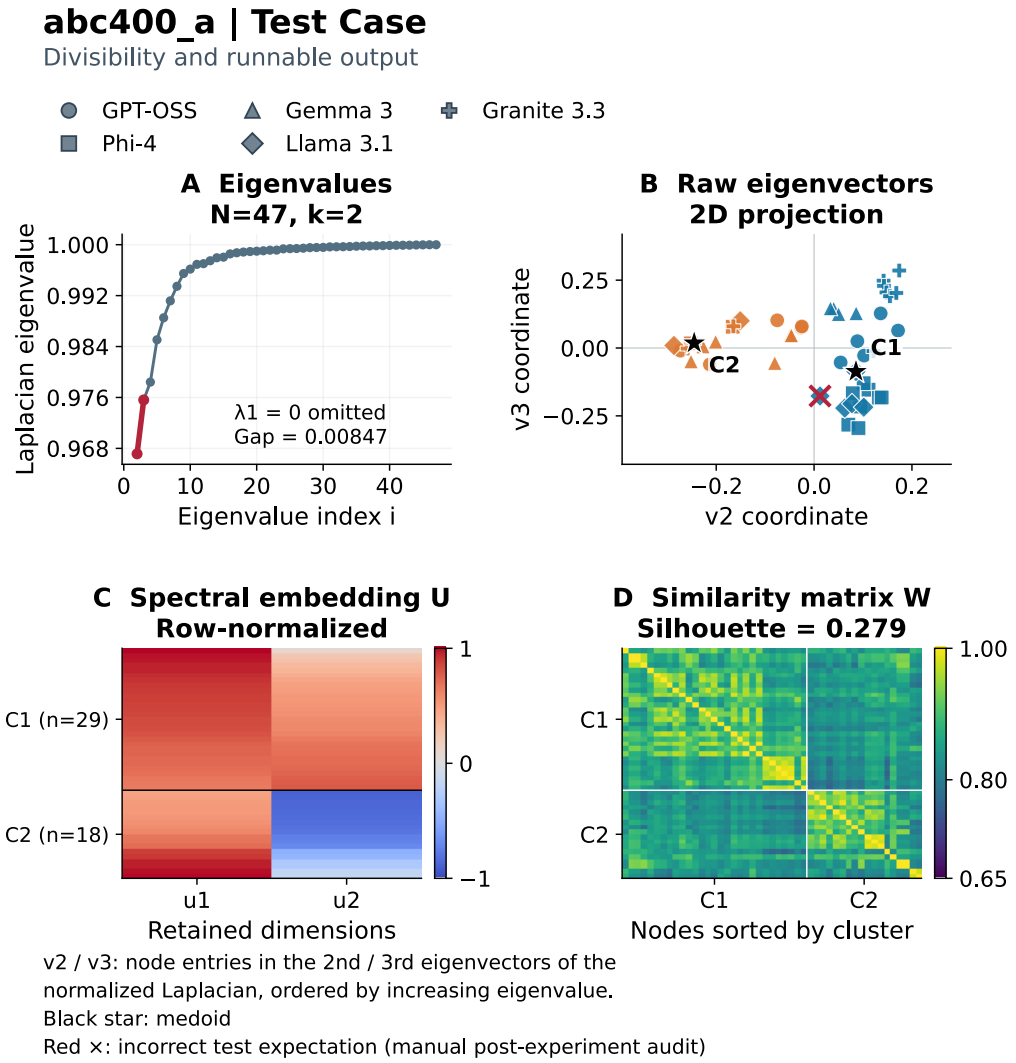


Figure 8: Spectral visualisation of LiveCodeBench task `abc400_a` (continued): Test Case stage. The two clusters mainly distinguish quotient answers from validation helpers and non-divisor cases. The red × marks a test expecting -1 for $A = 5$, although the correct answer is 80. Its membership in a cluster containing correct quotient tests illustrates that semantic similarity does not ensure a correct expected output.

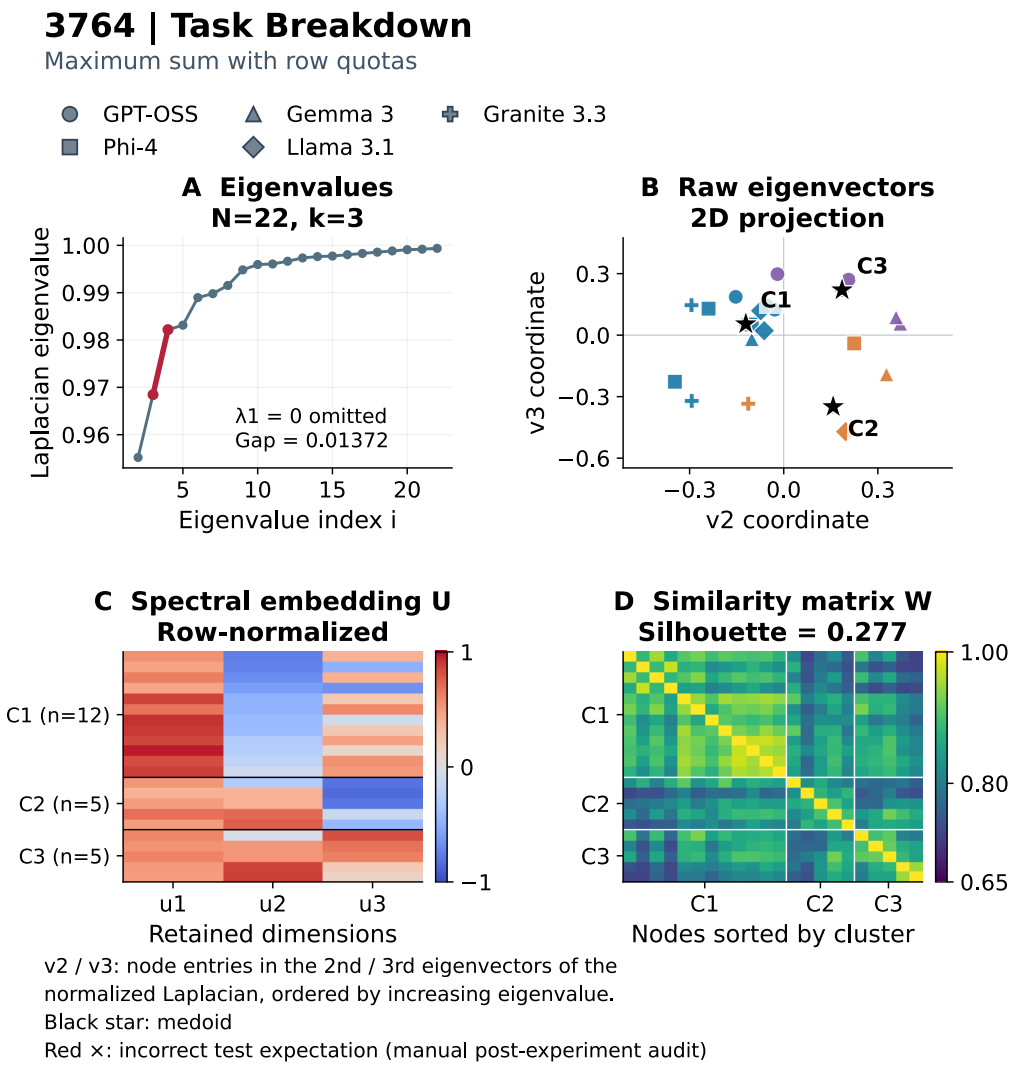


Figure 9: Spectral visualisation of LiveCodeBench task `3764`: Task Breakdown stage. The three clusters cover mixed input and selection logic, initialisation and partial sums, and aggregation. Correct greedy ideas, exhaustive enumeration, and ambiguous interpretations of the global selection limit can occur in the same semantic group.

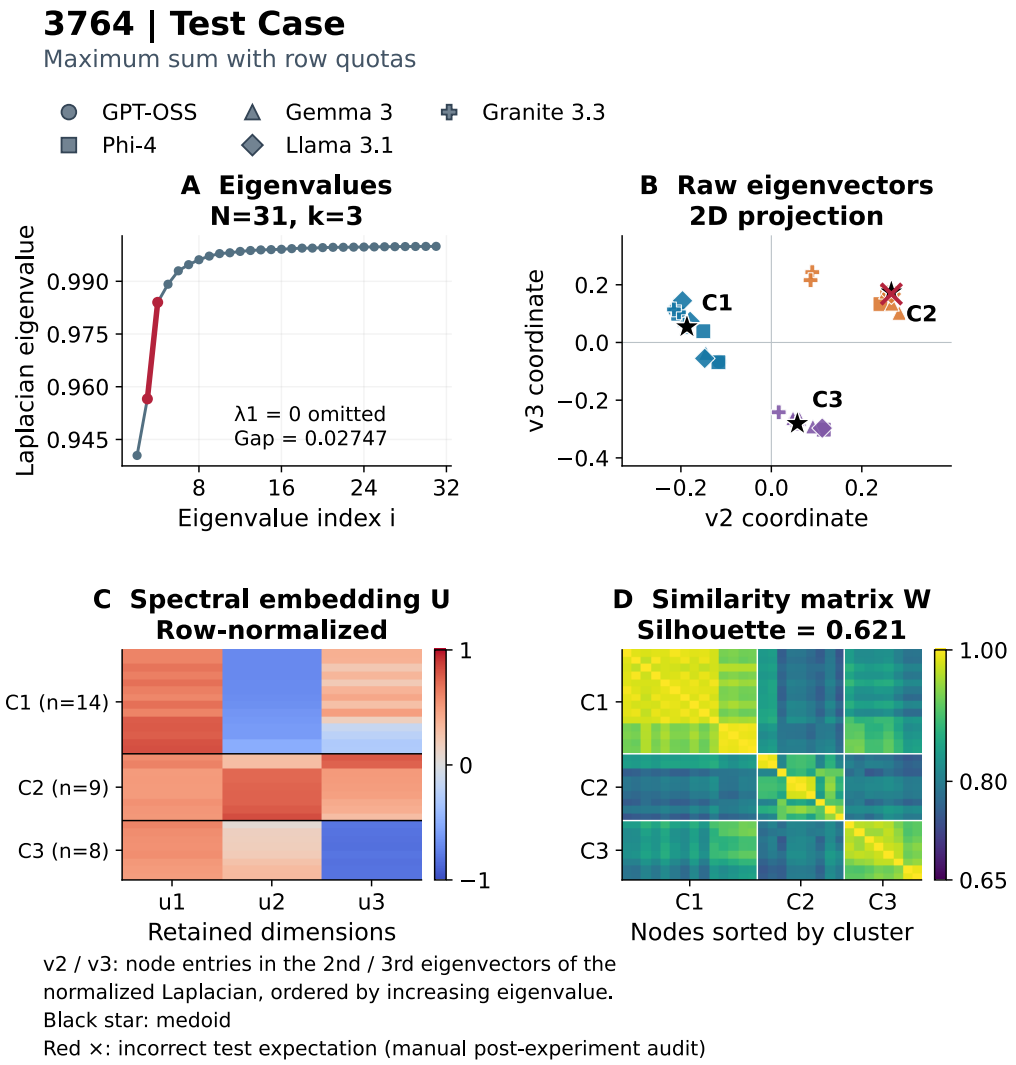


Figure 9: Spectral visualisation of LiveCodeBench task 3764 (continued): Test Case stage. The clusters separate input validation, summation of already-selected elements, and grid initialisation, while the saved tests omit a complete check of the constrained maximum-sum problem. The marked test expects 50 for the sum of $[[10, 20], [30, 40]]$, whose correct sum is 100.

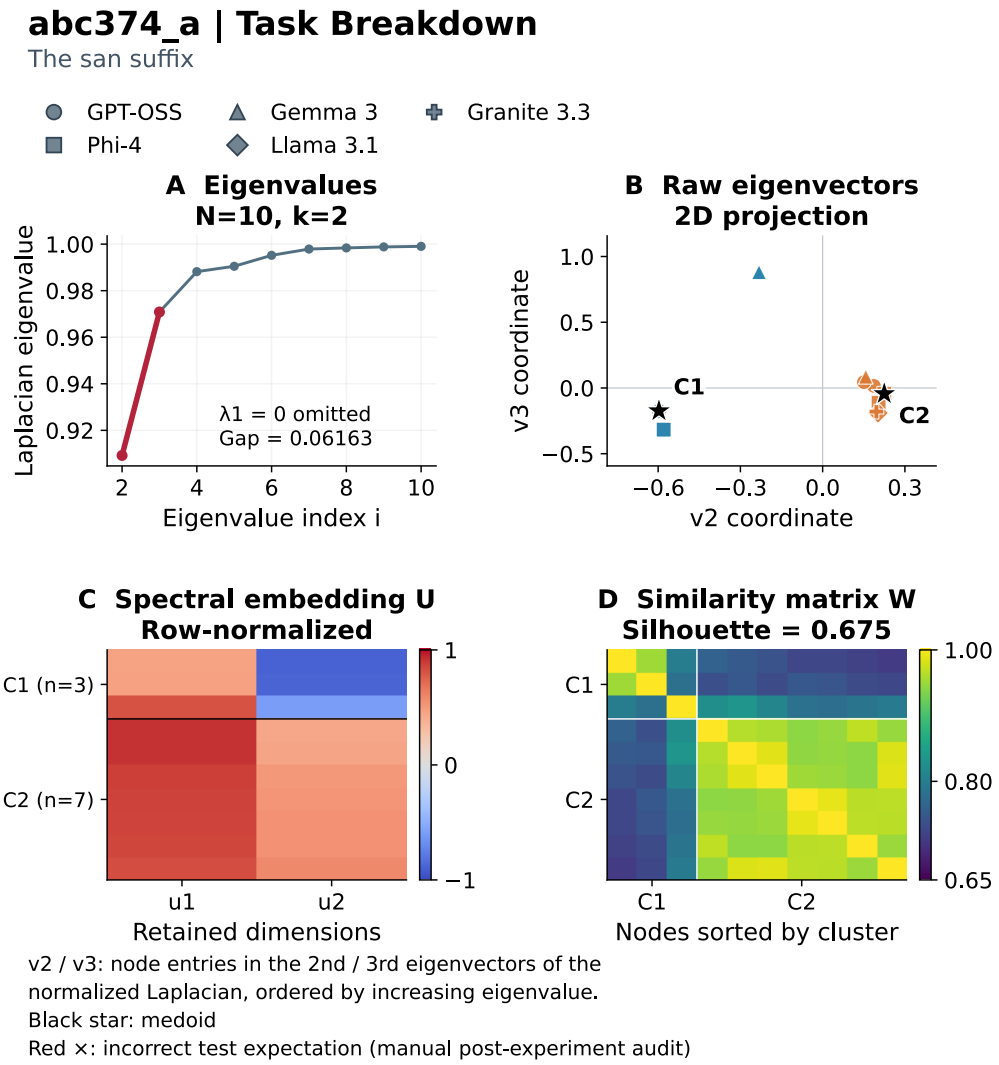


Figure 10: Spectral visualisation of LiveCodeBench task `abc374_a`: Task Breakdown stage. The principal separation is between reading or handling the input string and checking the suffix or formatting the output. This functional separation does not establish that every member fully captures the task requirements.

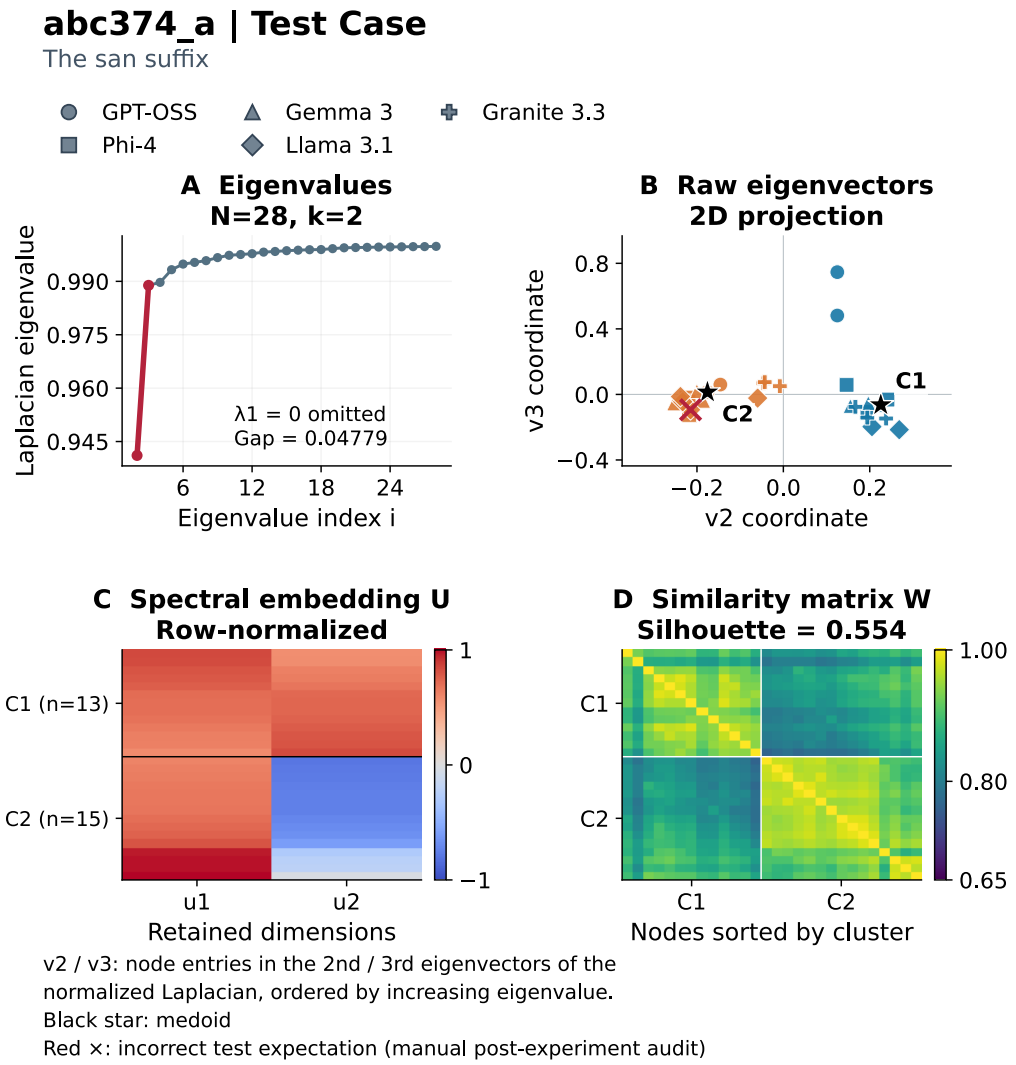


Figure 10: Spectral visualisation of LiveCodeBench task `abc374_a` (continued): Test Case stage. The clusters broadly separate input-reading tests from suffix-predicate tests. The suffix-checking cluster contains a test incorrectly expecting `True` for `abcd`. A clear functional partition can therefore contain an incorrect expectation.

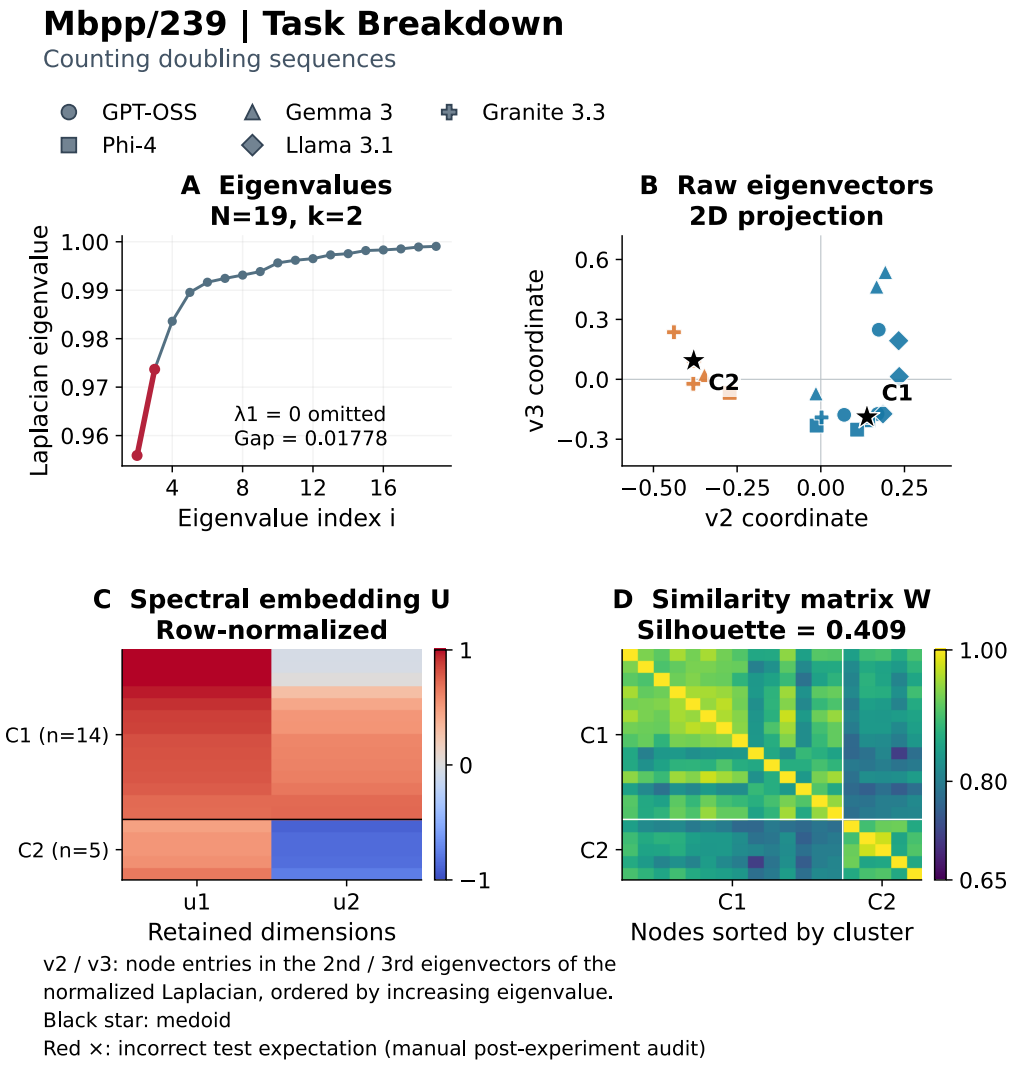


Figure 11: Spectral visualisation of MBPP+ task 239: Task Breakdown stage. The clusters distinguish whole-task counting and integration from recursive-helper and base-case language. The groups contain mixed interpretations of the sequence-length and doubling constraints, so semantic agreement alone does not resolve correctness.

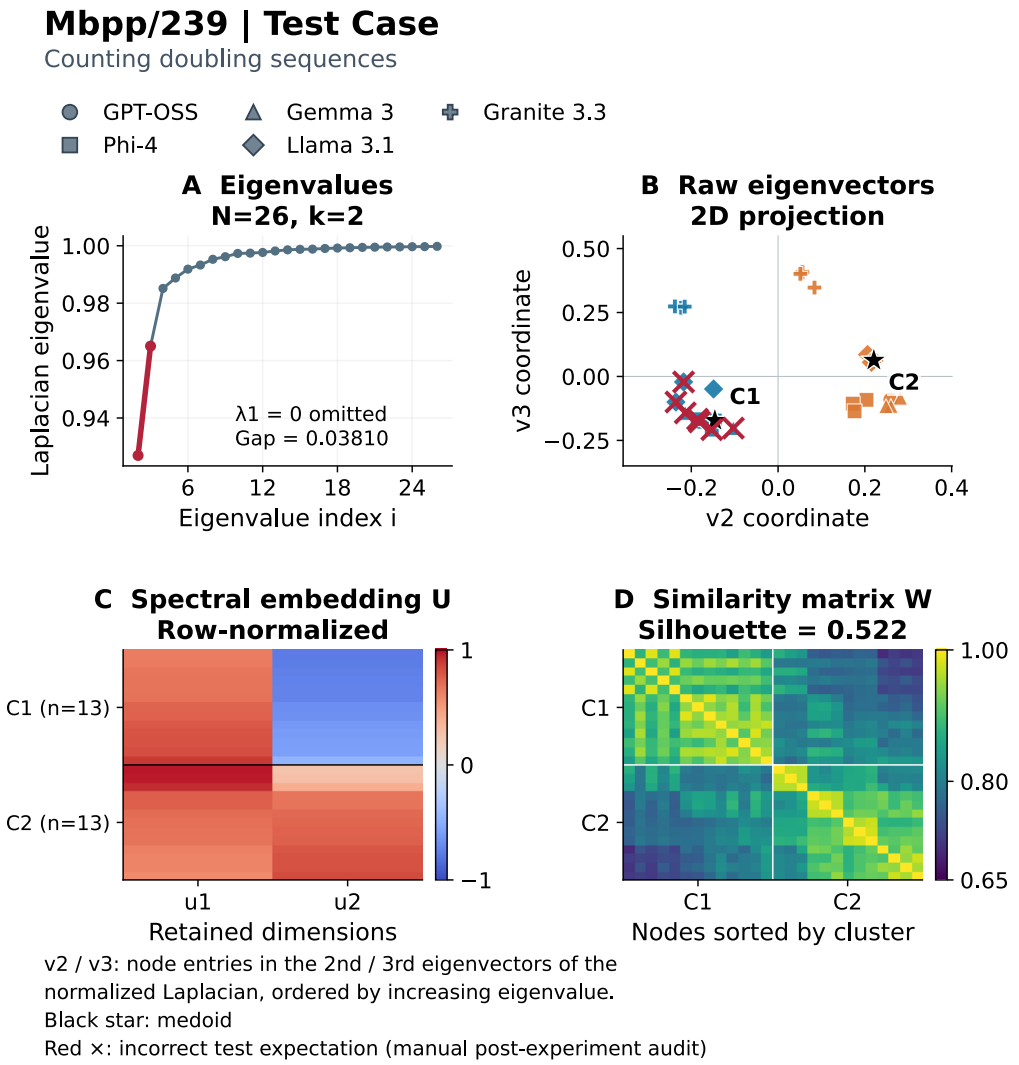


Figure 11: Spectral visualisation of MBPP+ task 239 (continued): Test Case stage. The clusters distinguish whole-task sequence counts from recursive-helper counts. The whole-task cluster contains seven incorrect counts among ten concrete numeric expectations, including $(m, n) = (3, 2) \mapsto 9$ instead of 2. The recursive-helper tests omit information needed to determine a unique correct count.

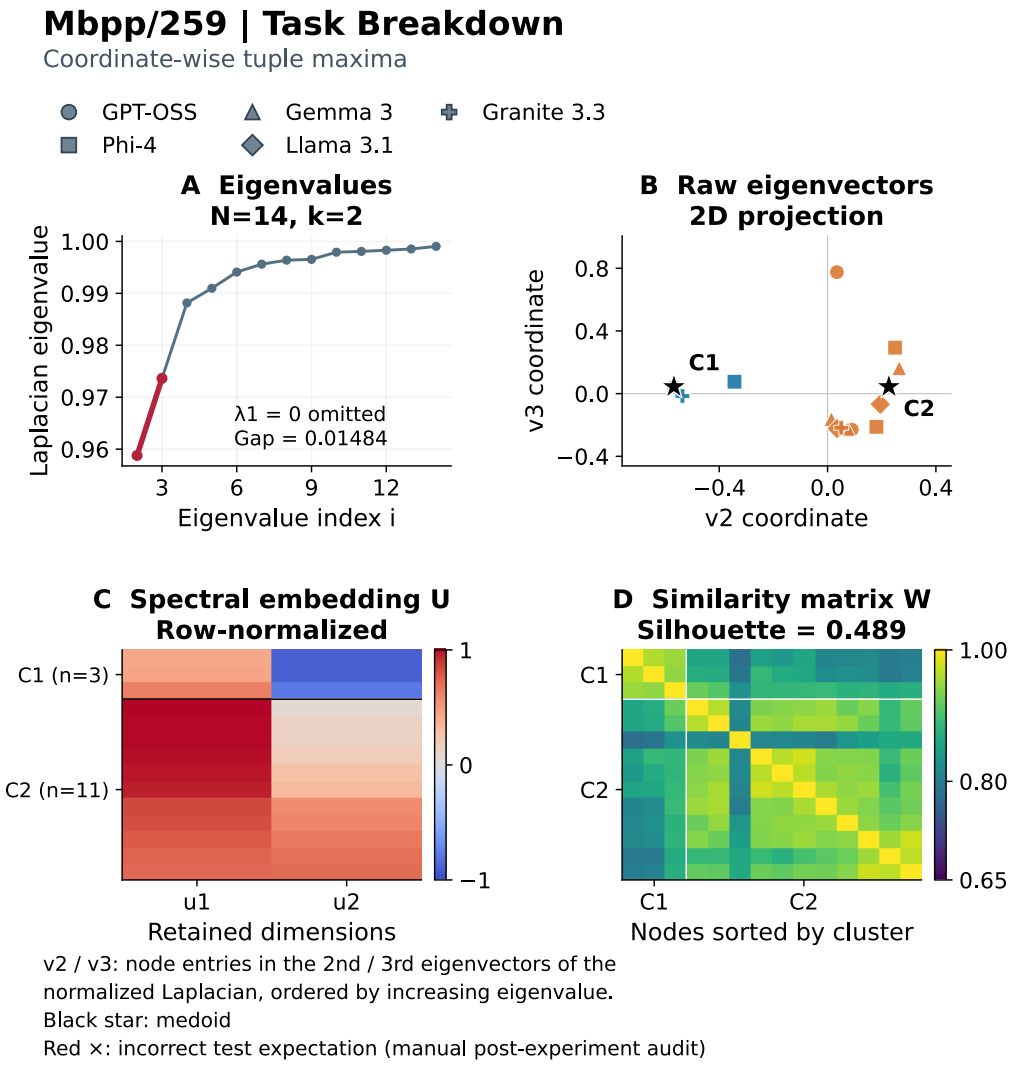


Figure 12: Spectral visualisation of MBPP+ task 259: Task Breakdown stage. The clusters mainly distinguish input-structure validation from coordinate-wise maximum computation and result assembly. Agreement about these broad roles can coexist with ambiguity about data types and the precise comparison operation.

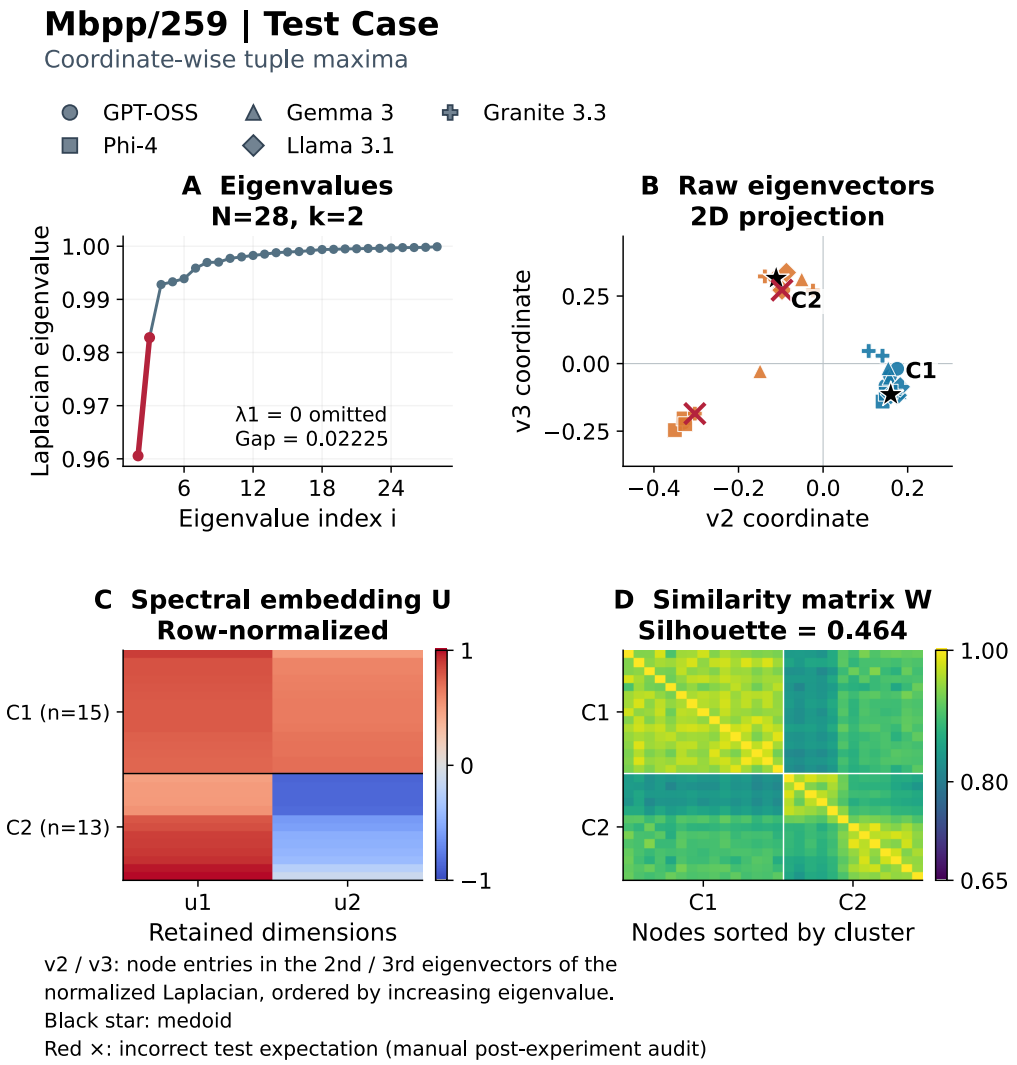


Figure 12: Spectral visualisation of MBPP+ task 259 (continued): Test Case stage. The tests focus on validation and returning an already-computed `max_tuples` input, leaving coordinate-wise maximum computation untested. The two marked nodes contain inconsistent expected outputs under the stated return/assembly helper contract. Both correctness and task coverage require evaluation beyond semantic grouping.

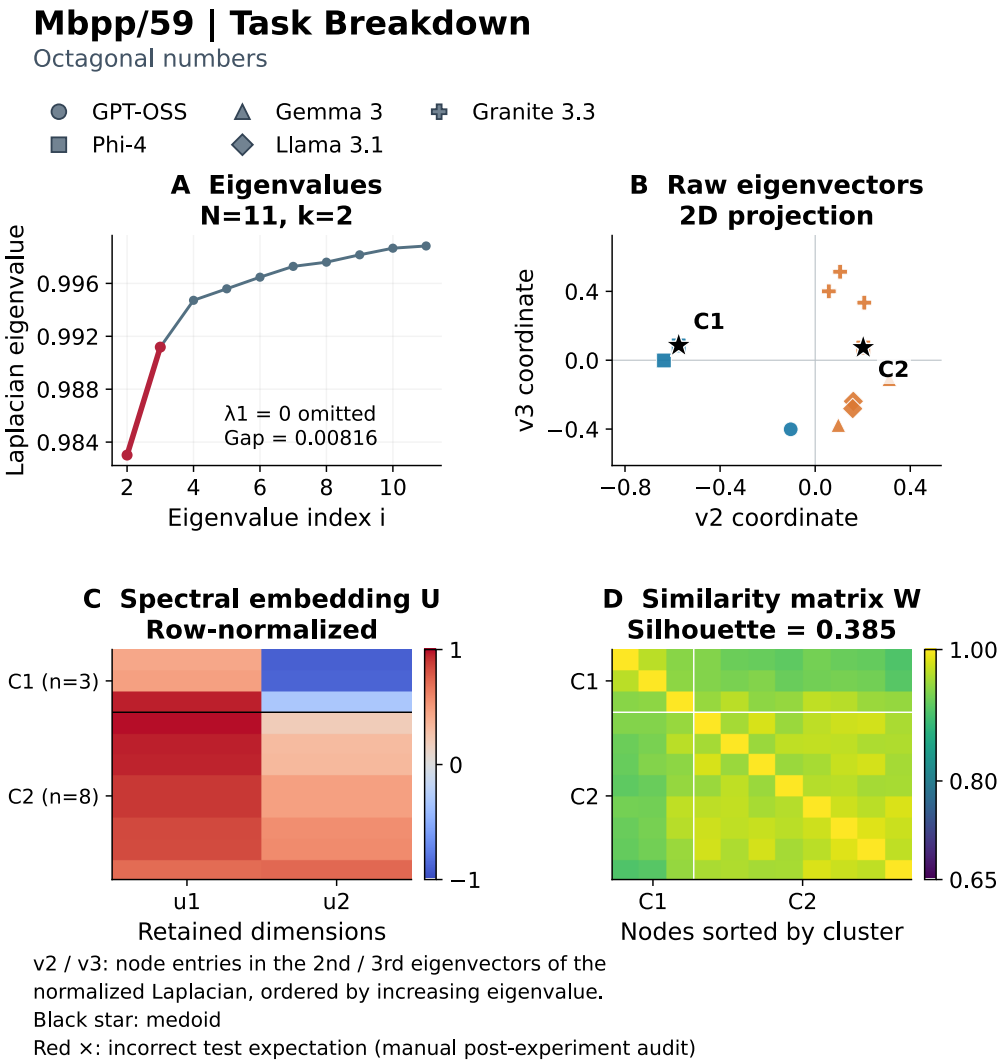


Figure 13: Spectral visualisation of MBPP+ task 59: Task Breakdown stage. The clusters separate validation and integration from computing the octagonal-number formula $n(3n-2)$. The partition captures functional roles rather than a general distinction between desirable and undesirable outputs.

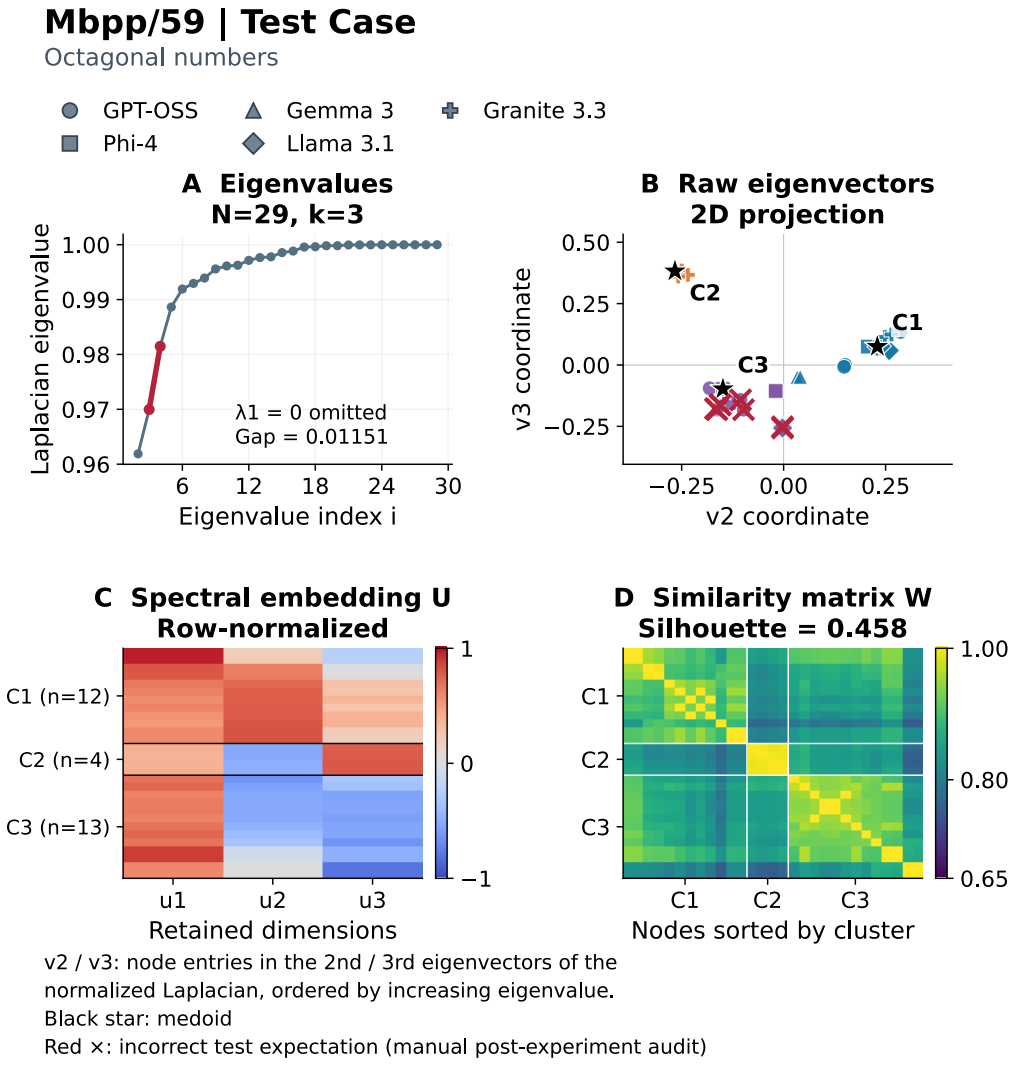


Figure 13: Spectral visualisation of MBPP+ task 59 (continued): Test Case stage. The clusters distinguish boundary or invalid-input behaviour, descriptive placeholders, and numeric formula answers. The numeric-answer cluster contains eight incorrect expectations among thirteen nodes; some tests give 71 or 73 for $n = 5$, whereas $n(3n - 2) = 65$. A separate Granite-only cluster contains descriptions without concrete numeric expectations.