Architectures for Code Development with LLMs

Daniele Amato `334211`
Matteo Flores `337579`
Natale Mastrogiacomo `345071`
Ilaria Parodi `339184`
Giuseppe Ventrella `348517`

Abstract

Large Language Models (LLMs) have shown strong performance in code generation tasks, but with limited reliability and maintainability, especially when using a single agent. This work aims to investigate whether Multi-Agent Systems (MAS) can address these limitations through architectural decomposition. We propose a modular pipeline with four roles: *Planner*, *Coder*, *Reviewer*, and *Commenter*. Instead of using the heterogeneous ensemble approach used in previous works, we isolate the effect of the architecture by using the same lightweight model, **Gemini 2.5 Flash Lite** across both a Single-Agent baseline and a Multi-Agent configuration. The two architectures are evaluated on 20 programming tasks using a composite metric that combines Functional Correctness, Maintainability Index (MI), Cyclomatic Complexity (CC), and Execution Time. Results, supported by ablation studies, show the potential of MAS to improve quality and robustness of code, even when using efficient, low-latency models.

1 Introduction

The recent developments in LLMs have significantly influenced automated software engineering (Chen et al., 2021; Zhang et al., 2025). Despite the current state-of-the-art LLMs being able to generate good standalone code snippets, the standard approach of using Single Agents result often insufficient in providing acceptable depth of reasoning when handling complex software engineering tasks (Jiang et al., 2024). The absence of planning strategies and self-correcting abilities of Single Agents may generate code that, even if syntactically correct, might not cover all edge cases and may be more difficult to maintain.

To address these problems, the field of software engineering using LLMs has shifted towards Multi-Agent Systems (MAS) that try to follow the process of human software development using role specialization (Qian et al., 2023; Zhang et al., 2025). However, it remains an open question whether the benefits of MAS stem solely from using more powerful models or from the architecture itself.

This work investigates the impact of using Multi-Agent architectures with a lightweight efficiency-oriented LLM **Gemini 2.5 Flash Lite**. We implement a sequential pipeline with a static analysis feedback loop, assigning the model distinct personas (*Planner*, *Coder*, *Reviewer*, *Commenter*). After that, we compare this approach against a standard Single-Agent baseline using the exact same model instance.

1.1 Research Questions

This study aims to address the following research questions:

- **RQ1:** *Which architectures produce higher-quality and more maintainable code?*
- **RQ2:** *How do agent coordination strategies impact correctness?*
- **RQ3:** *Does modular role separation improve code generation?*

2 Background

Automated code generation has evolved in recent years, from simple autocompletion systems to more complex agentic workflows.

2.1 LLM-based Code Generation

LLMs trained on massive code repositories have shown strong performance in solving algorithmic problems (Chen et al., 2021). However, when operating as single agents, these models show limitations in handling complex requirements, as they are prone to generate code based on immediate probability without considering broader architectural

constraints or edge cases (Zhang et al., 2025). Techniques such as Chain-of-Thought (CoT) prompting attempt to mitigate this by encouraging intermediate reasoning steps, but they often lack rigorous verification.

2.2 Multi-Agent Collaboration

Multi-Agent Systems (MAS) try to surpass single-agent limitations by decomposing the workload across specialized roles. Frameworks like *ChatDev* (Qian et al., 2023) and *MetaGPT* (Hong et al., 2024) demonstrated that assigning roles (e.g., CEO, CTO, Programmer) improves software consistency. A key component of effective MAS is the **feedback loop**. Recent studies on *Self-Debugging* (Chen et al., 2023) and *Self-Refinement* suggest that models can correct their own errors if provided with error logs or static analysis reports. Our work builds on these findings by integrating a dedicated *Reviewer* agent that uses static analysis to guide a *Coder* agent, aiming to perform iterative refinement before the code is finalized.

3 System overview

We designed a modular MAS with a multi-step cooperative ordered approach to address the limitations of single-pass code generation (Zhang et al., 2025). As illustrated in Figure 1, the architecture is structured as a sequential pipeline with an iterative feedback loop, comprising four specialized agents: *Planner*, *Coder*, *Reviewer*, and *Commenter*.

3.1 Architectural Pipeline and Evolution

The proposed workflow starts with the **Planner**, which analyzes the task description to produce a high-level strategy, a function signature, and a set of edge cases, *strictly avoiding code implementation*. The **Coder** then implements the solution following the generated plan.

A key design decision concerns the verification stage. An early version included a **Tester** agent responsible for automatically generating and executing test suites in order to maximize coverage. This approach was later discarded, as automated test generation introduced significant additional complexity and instability. The final design instead relies on a **Reviewer** agent that performs a hybrid validation process, combining static analysis with the inspection of test execution outcomes. The Reviewer uses zero-shot prompting to detect common issues such as syntax errors, missing imports, and incomplete implementations, and provides structured feedback to guide iterative refinement, following a self-debugging paradigm (Chen et al., 2023).

If the Reviewer identifies flaws, the system enters a **feedback loop** (*limited to 10 retries*), passing the error log back to the Coder for refinement. Initially, the maximum number of loop-backs has been set to *5*, but we noticed that it was not sufficient for more complex tasks, while experimenting with higher values we noticed that the model *never converged after 10 retries*. Based on these observations, a limit of *10 iterations* was selected to balance effectiveness and stability.

Once the code passes the review, the **Commenter** adds Google-style docstrings and inline documentation before the final evaluation.

3.2 Dataset Construction

To ensure a robust evaluation across different domains and difficulty levels, we curated a composite dataset of 20 programming tasks derived from three established benchmarks and custom problems:

- **HumanEval:** 6 tasks (IDs 1, 2, 3, 12, 19, 20) focusing on algorithmic logic and string manipulation.
- **MBPP:** 4 tasks (IDs 4, 5, 6, 17) covering fundamental programming concepts.
- **CodeNet:** 8 tasks (IDs 7, 8, 9, 10, 13, 14, 16, 18) involving data processing, graph algorithms, and advanced data structures.
- **Custom:** 2 tasks (IDs 11, 15) designed to test other algorithms not present in standard datasets. In this case, we manually defined tests for the static evaluation of the reviewer.

3.3 Input Data Files for Tests

For each programming task, we provide the corresponding input file. Each of them contains sample input data that is used only to test the generated code implementations, acquiring the time needed for the execution.

4 Planner

The Planner agent takes as input a programming task and outputs a structures representation divided in 3 components: a function signature, a high-level solution plan and a list of edge cases. Its role is to analyze the problem and generate a solution strategy, without generating executable code.

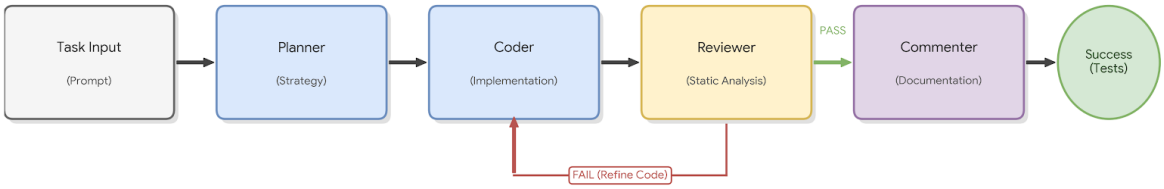

Figure 1: Overview of the Multi-Agent Architecture using Planner, Coder, Reviewer and Commenter agents.

To ensure consistency and comparability between configurations, the prompt enforces strict formatting and behavioral constraints. In particular, the Planner must not produce executable code, imports or docstrings, and is required to be concise, precise and avoid conversational language.

The decoding process is deterministic, with the temperature parameter set to 0. Planning is treated as a task that benefits from stability and reproducibility, rather than creativity. This configuration ensures fair comparison across models by minimizing output variance.

5 Coder

5.1 Agent settings

The Coder agent takes as input the original task description, the structured plan generated by the Planner, and in case of retry the current code with test feedback. Its role is to produce executable Python code that implements the solution strategy defined in the plan.

The Coder must output only valid Python code, including necessary imports at the beginning, without markdown formatting, comments, or explanatory text. The function signature extracted from the Planner output must be preserved exactly.

Two distinct prompt templates are used depending on the generation phase. The initial generation template instructs the model to follow the plan step by step and produce the complete function. The refinement template, used when previous code failed tests, provides the erroneous code with test feedback, instructing the model to correct only the logic or edge cases while preserving the original signature.

The temperature parameter varies based on the task: 0.2 for initial generation, favoring deterministic and structured output, and 0.4 for code fixing, allowing slightly more flexibility to explore alternative solutions. The maximum number of new tokens is set to 1024.

5.2 Output extraction.

Since language models frequently produce additional text alongside the requested code, a filtering function is applied to the raw output. Markdown code block delimiters are removed, and the extraction logic identifies the first line containing a code keyword (`import`, `from`, `def`, or `class`) as the start of the code block. Subsequent lines are collected as long as they are indented, empty, or contain additional code keywords. This procedure guarantees that only clean, executable Python code is passed to the Reviewer.

6 Reviewer

The Reviewer agent acts as the primary quality assurance gatekeeper. It's implementation follows a **Hybrid Dynamic-Static** approach to ensure both functional correctness and syntactic validity. The review process follows a two-step validation pipeline:

- **Dynamic Execution (Test-Driven):** The agent attempts to execute the generated code within a controlled namespace using Python's `exec` utility. It runs the candidate solution against the specific unit tests provided in the task definition (e.g., HumanEval assertions). This step verifies functional correctness and captures runtime errors and test failures.
- **LLM-Based Debugging:** If execution fails, the agent transitions to a debugging role. It synthesizes a structured debugging context aggregating three key components: the faulty

source code, the reference test suite, and the captured runtime traceback (e.g., specific `AssertionError` or stack trace). This comprehensive context allows the model to diagnose the root cause and generate precise, error-aware feedback for the Coder.

If the code passes all dynamic tests, it is immediately approved. Otherwise, the structured error analysis is fed back to the Coder for refinement, repeating the cycle for a maximum of 10 iterations (Section 3.1).

7 Commenter

The Commenter agent executes the final pipeline stage, enhancing the readability and documentation of the validated solution without altering its logic. To avoid redundant "over-commenting," the agent follows strict guidelines to optimize information density:

- **Non-Triviality:** Basic syntax explanations (e.g., imports, loop counters) are strictly forbidden to reduce visual noise.
- **Algorithmic Intent:** Inline comments are restricted to complex logic or edge cases, focusing solely on the "why" rather than the "what".
- **Compact Documentation:** Google-style docstrings are mandated, prioritizing concise, single-line summaries over verbose parameter descriptions.

This strategy ensures professional-grade output by balancing structural clarity with essential documentation.

8 Experimental results

To evaluate the quality of the generated code, we adopt a set of complementary metrics capturing maintainability, structural complexity, and execution efficiency.

- **Maintainability Index (MI)**, estimates how difficult the code is to understand, test, and maintain (Coleman et al., 1994). It is computed using the *radon* Python library.
  Since MI takes into account the lines of code, and the Multi Agent architecture generates a more commented code than the Single Agent, the code has been cleaned of comments before the evaluation to avoid bias.
- **Cyclomatic Complexity (CC)**, counts the number of linearly independent paths, where a lower value indicates a better score. For a single function, CC is computed according to (McCabe, 1976):

$$CC = E - N + 2P \tag{1}$$

  or, the simplified one (since we are dealing with just one function):

$$CC = E - N + 2 \tag{2}$$

  CC is computed using the *radon* library, which provides an equivalent implementation based on control-flow analysis.
- **Execution time**: measures the runtime efficiency of functionally correct implementations. It is measured by running each solution on a fixed input instance associated with the task.

The results are reported in Table 1.
In order to give a single-value comparison between the two results, a single **score** value has been computed, merging the normalized previously reported metrics, as follows:

- Maintainability Index is rescaled in range $[0, 1]$:

$$MI_{norm} = \frac{\min\{\max\{MI, 0\}, 100\}}{100} \tag{3}$$

- Cyclomatic Complexity is normalized in $(0, 1]$ as follows:

$$CC_{norm} = \min\{1.0, 1.0/CC\} \tag{4}$$

- Execution time has been rescaled in $(0, 1]$ considering $0.1\ s$ as the target execution time (i.e. for every $ET \leq 0.1\ s$, the maximum normalized score will be assigned):

$$ET_{norm} = \min\left\{\frac{0.1}{\max\{ET, 0.00001\}}, 1.0\right\} \tag{5}$$

The final score is defined as a weighted sum of the normalized metrics as:

$$FS = 0.4 \cdot ET_{norm} + 0.2 \cdot MI_{norm} + 0.4 \cdot CC_{norm} \tag{6}$$

Task ID	Single Agent (SA)				Multi Agent (MA)			
	MI	CC	Time (s)	Score	MI	CC	Time (s)	Score
Simple Tasks								
Task 01	69.57	4.00	531.77	0.2838	69.57	4.00	538.42	0.2838
Task 02	62.78	5.00	0.019712	0.6483	62.78	5.00	0.019848	0.6483
Task 03	73.95	3.00	0.027499	**0.7218**	72.92	5.00	0.023196	0.6788
Task 04	63.56	3.00	0.159390	0.5416	68.09	3.00	0.156769	**0.5594**
Task 05	80.65	2.00	0.005419	0.7613	80.65	2.00	0.014884	0.7613
Task 06	69.96	4.00	0.010942	0.6399	69.96	4.00	0.006220	0.6399
Task 07	76.55	2.00	0.000003	**0.7531**	74.17	2.00	0.000002	0.7483
Task 08	71.29	3.00	0.000011	**0.6759**	67.20	4.00	0.000002	0.6344
Task 09	81.99	1.00	0.000003	0.9640	81.99	1.00	0.000002	0.9640
Task 10	67.57	3.00	0.000003	**0.6685**	65.67	3.00	0.000002	0.6647
Complex Tasks								
Task 11	57.94	7.00	0.007359	0.5730	60.98	6.00	0.006748	**0.5886**
Task 12	64.11	5.00	0.000006	0.6082	64.92	5.00	0.000004	**0.6098**
Task 13	-	-	-	-	45.35	12.00	0.000032	**0.5240**
Task 14	42.27	4.00	0.000300	0.5845	42.39	3.00	0.000323	**0.6181**
Task 15	45.58	20.00	0.000137	0.5112	47.83	20.00	0.000027	**0.5157**
Task 16	55.23	9.00	0.000032	0.5549	56.05	9.00	0.000018	**0.5566**
Task 17	62.88	5.00	0.000230	0.6058	65.41	4.00	0.000419	**0.6308**
Task 18	47.45	6.00	0.000024	0.5616	48.59	5.00	0.000024	**0.5772**
Task 19	61.39	6.00	13.033335	0.1925	55.45	4.00	0.000175	**0.6109**
Task 20	60.48	9.00	0.003168	0.5654	57.93	6.00	0.000026	**0.5825**

Table 1: Metrics comparison: Single Agent vs Multi Agent. For Task13 the Single Agent did not produce executable code (indicated by '-'). Task19 reported the largest difference of execution time ($13s$)

Task ID	Single Agent				MA without Planner				MA with Planner			
	MI	CC	Time (s)	Score	MI	CC	Time (s)	Score	MI	CC	Time (s)	Score
Simple Tasks												
Task 01	69.57	4.00	531.77	**0.2838**	-	-	-	-	69.57	4.00	538.42	**0.2838**
Task 02	62.78	5.00	0.0197	**0.6483**	62.78	5.00	0.0086	0.6056	62.78	5.00	0.0198	**0.6483**
Complex Tasks												
Task 11	57.94	7.00	0.0074	0.5730	-	-	-	-	60.98	6.00	0.0067	**0.5886**
Task 12	64.11	5.00	0.0000	0.6082	63.21	5.00	0.0000	0.6064	64.92	5.00	0.0000	**0.6098**
Task 13	-	-	-	-	-	-	-	-	45.35	12.00	0.0000	**0.5240**
Task 14	42.27	4.00	0.0003	0.5845	39.32	4.00	0.0003	0.5786	42.39	3.00	0.0003	**0.6181**
Task 19	61.39	6.00	13.033	0.1925	55.32	6.00	0.0000	0.5773	55.45	4.00	0.0002	**0.6109**

Table 2: Comparative analysis of Single Agent, Multi-Agent without Planner, and Multi-Agent with Planner. The Planner architecture shows superior stability and success rates, particularly in complex tasks where other methods fail. Sometimes, the function used to compute the execution time returned 0.0000: it's a technical issue since the real execution time is $0.0000 < t < 0.0001$.

8.1 Ablation Studies

Due to API quota limitations, ablation studies were conducted on a subset of tasks, including two Simple Tasks (Task 01-02) and five Complex Tasks (Tasks 11–14 and 19). Task19 is particularly informative due to the large execution time gap observed between Single-Agent and Multi-Agent configurations.

8.1.1 The impact of the Planner

To assess the necessity of explicit planning, we conducted an ablation study removing the Planner agent from the MA architecture. As shown in Table 2, this often results to non-executable or incomplete code even for simple tasks. This highlights that explicit planning can improve robustness even when task complexity is limited.

8.1.2 Reviewer Feedback Loop

In our proposed architecture the number of times the code can generate and fix its own code plays a crucial role, as previously explained (Section 3.1). To isolate its effect, we conducted an additional ablation study removing the loop-back. The new Reviewer's role, without loop-back, consists in establishing if the generated code passes or not all the test cases without further refinement.

Task ID	Single Agent				MA without loop-back				MA with loop-back			
	MI	CC	Time (s)	Score	MI	CC	Time (s)	Score	MI	CC	Time (s)	Score
Simple Tasks												
Task 01	69.57	4.00	531.77	**0.2838**	66.18	5.00	197.129828	0.2126	69.57	4.00	538.42	**0.2838**
Task 02	62.78	5.00	0.0197	**0.6483**	62.78	5.00	0.006753	0.6056	62.78	5.00	0.019848	**0.6483**
Complex Tasks												
Task 11	57.94	7.00	0.0074	0.5730	60.98	6.00	0.004135	**0.5886**	60.98	6.00	0.006748	**0.5886**
Task 12	64.11	5.00	0.0000	0.6082	-	-	-	-	64.92	5.00	0.000004	**0.6098**
Task 13	-	-	-	-	45.22	13.00	0.000045	0.5212	45.35	12.00	0.000032	**0.5240**
Task 14	42.27	4.00	0.0003	0.5845	-	-	-	-	42.39	3.00	0.000323	**0.6181**
Task 19	61.39	6.00	13.033	0.1925	52.47	6.00	0.000130	0.5716	55.45	4.00	0.000175	**0.6109**

Table 3: Ablation Study: Impact of the Reviewer's Loop-back mechanism. Iterative refinement is crucial for correctness.

The results in Table 3 show that while the effect has limited impact on simple tasks, the absence of iterative refinement often results in non-executable or lower-quality code for complex tasks. These results confirm that feedback-driven correction is essential for handling complex programming tasks.

9 Conclusion

This work aims to investigate whether architectural decomposition alone can improve code generation quality when using lightweight Large Language Models. We conducted a controlled comparison between a Single-Agent and a Multi-Agent architecture, isolating the effect of architectural design by using the same underlying model in both configurations. The results are presented in Table 1.

RQ1: ***Which architectures produce higher-quality and more maintainable code?*** The code generated by the two architectures obtained similar results in terms of performance for Simple Tasks (IDs from 01 to 10). For Complex Tasks (IDs from 11 to 20), the MA code performs significantly better in terms of Execution Time and Cyclomatic Complexity, and slightly better in terms of Maintainability Index. Since functional correctness has been used as a metric for the correctness evaluation by the Reviewer, it has not been used for result comparison, to avoid introducing bias.

RQ2: ***How do agent coordination strategies impact correctness?*** Iterative feedback between the Reviewer and the Coder is essential for correcting initial errors and obtaining executable codes in complex tasks. Furthermore, the ablation study without loop-back confirms that in absence of iterative refinement, the system frequently fails to converge to correct implementations.

RQ3: ***Does modular role separation improve code generation?*** Modular role separation significantly improves robustness and correctness for complex tasks, while offering limited benefits for simpler ones. Lightweight models struggle to handle complex tasks within a single generation step due to the lack of explicit planning and long-horizon reasoning mechanisms. Ablation results further confirm that code generation alone is not always sufficient to achieve stable and high-quality outputs, and that explicit planning and review phases play an important role within the overall architecture.

10 Future work

An initial implementation of this work involved using different models for each specific agent of the Multi-Agent architecture in order to exploit the best performing model for each role. In this setting, combinations of models from different families, such as Mistral, Qwen, DeepSeek were evaluated. Despite using models with comparable or larger parameters counts, these heterogeneous architectures performed poorly, even on simple tasks, often failing to converge or produce executable code (as shown in Appendix A). This suggests that the observed degradation was not due to task complexity, but rather to limitations in model coordination. A possible explanation is that using different models introduce cross-model misalignment in reasoning style and output structure, leading to error amplification rather than correction. Future work should investigate whether larger model capacity can mitigate these limitations, or whether homogeneous model architectures remain more stable for multi-agent code generation.

References

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, H. P. d. O. Pinto, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*.

Don Coleman, Dan Ash, Bruce Lowther, and Paul Oman. 1994. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zhiyong Wang, and 1 others. 2024. Metagpt: Meta programming for multi-agent collaborative framework. In *International Conference on Learning Representations (ICLR)*.

Xue Jiang, Yihong Dong, Liang Wang, Zhuobin Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenjie Jiao. 2024. Self-planning code generation with large language models. *ACM Transactions on Software Engineering and Methodology*, 33(7):1–30.

Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320.

Chen Qian, Wei Liu, Hong Liu, Nuo Chen, Yifei Dang, Jia Li, Cheng Yang, Weize Chen, Yusheng Su, and 1 others. 2023. Chatdev: Communicative agents for software development. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 15174–15186.

Li Zhang, Wei Chen, and 1 others. 2025. A multi-agent framework for complex code generation. *arXiv preprint arXiv:2508.00083*.

A Results of the multi model analysis

The tasks used for this case study are different from those used in this report, and we can consider all them as Simple Tasks . In fact, they are as follows:

- 6 tasks from HumanEval (extracted randomly)
- 6 tasks from MBPP (extracted randomly)
- 6 tasks from CodeNet (extracted randomly)
- 2 custom tasks

In particular, since the Planner and the Coder may be the most important roles, we chose 2 models for each of them and tried every single combination. The Reviewer used a Qwen model and the Commenter used a small Mistral model.
Therefore, we analyzed 5 architectures:

- Single Agent
- Multi Agent, using *Mistral* as Planner and *Qwen* as Coder
- Multi Agent, using *Mistral* as Planner and *Nemo* as Coder
- Multi Agent, using *DeepSeek* as Planner and *Qwen* as Coder
- Multi Agent, using *DeepSeek* as Planner and *Nemo* as Coder

The results are shown in Table 4.

Task ID	Single Agent	Mistral → Qwen	Mistral → Nemo	DeepSeek → Qwen	DeepSeek → Nemo
Task 01	**0.7256**	-	0.6483	0.6689	0.6462
Task 02	**0.7306**	0.6056	-	0.5887	-
Task 03	**1.0000**	0.7333	-	**1.0000**	**1.0000**
Task 04	-	**0.6336**	-	-	-
Task 05	**0.6800**	-	-	0.6561	0.6038
Task 06	**0.6819**	0.6374	0.6378	0.6443	0.6395
Task 07	0.7692	**0.7922**	-	0.7120	-
Task 08	**0.7311**	-	-	0.6746	0.6780
Task 09	-	-	-	-	-
Task 10	-	-	-	-	-
Task 11	**0.8000**	0.6863	0.6767	0.7402	0.7402
Task 12	-	**0.6862**	-	**0.6862**	**0.6862**
Task 13	**0.6833**	0.6704	-	0.6695	0.6451
Task 14	**1.0000**	0.6969	-	-	-
Task 15	**0.6582**	0.6293	-	0.6600	0.5666
Task 16	0.6186	-	**0.7298**	-	0.6757
Task 17	-	0.5514	-	**0.7441**	0.6020
Task 18	-	-	-	**0.7051**	-
Task 19	**1.0000**	0.6742	0.7560	**1.0000**	-
Task 20	**0.6028**	**0.6028**	0.5645	0.6012	-

Table 4: Score (0-1) comparing every task, for each architecture. Values set in bold point out the best architecture for each task; the indicator "-" means that the architecture has generated no executable code.