Architectures for Code Development with LLMs:
A Comparative Study of Multi-Agent Approaches

Riccardo Marconi, Dorotea Monaco, Luigi Gonnella,
Kevser Gunaydin, Francesco Mina
Politecnico di Torino
{riccardo.marconi, dorotea.monaco, luigi.gonnella}@studenti.polito.it
{kevser.gunaydin, francesco.mina}@studenti.polito.it

Abstract

Large language models (LLMs) demonstrate impressive code generation capabilities, yet single-prompt interactions often fail on complex development tasks. We investigate whether multi-agent architectures improve code quality through role specialization and iterative refinement. We compare three approaches—Naive (one-shot generation), Single-Agent (5-stage pipeline with self-refinement), and Multi-Agent (Planner-Coder-Critic coordination)—across 25 programming tasks spanning five domains and three difficulty levels. Surprisingly, Multi-Agent and Naive achieve identical overall correctness (77.6%), outperforming Single-Agent (74.6%). However, Multi-Agent excels in algorithmically demanding domains (DSA: 85.9%, Logic: 75.7%) and medium-difficulty tasks (91.8%), justifying its computational overhead only for specific task types. Our findings reveal that architectural sophistication benefits complex algorithmic reasoning but can hurt performance on simple tasks through over-refinement.

1 Introduction

Large language models have revolutionized automated code generation, with models like Codex (Chen et al., 2021) and Code Llama (Roziere et al., 2023) demonstrating remarkable capabilities on programming benchmarks. However, single-prompt approaches often struggle with complex development tasks that require multi-step reasoning, systematic debugging, and quality assurance. Real-world software development involves iterative refinement cycles—planning, implementation, testing, and revision—that single interactions cannot capture.

Recent work in multi-agent systems (Hong et al., 2023; Qian et al., 2023) suggests that distributing responsibilities across specialized agents (e.g., planning, coding, reviewing) may improve software development outcomes by mimicking human team workflows. Yet, a fundamental question remains unanswered: does the computational overhead of multi-agent coordination consistently translate to higher code quality? No systematic evaluation exists comparing single-agent and multi-agent architectures for code generation across diverse task types and complexity levels, making it unclear when architectural sophistication is justified.

This work addresses this gap through a controlled empirical study comparing three architectural approaches across 25 programming tasks spanning five domains (data structures, logic, mathematics, string processing, list manipulation) and three difficulty levels. We evaluate a Naive baseline (one-shot generation), a Single-Agent pipeline with five iterative refinement stages, and a Multi-Agent system with role-specialized coordination (Planner-Coder-Critic). Our findings challenge the assumption that architectural sophistication uniformly improves performance, revealing that the benefits of multi-agent coordination are highly task-dependent and come at significant computational cost.

1.1 Research Questions

We address the following research questions from the assignment requirements:

1. **RQ1:** How do the architectures compare in terms of functional correctness and code quality?

2. **RQ2:** How do agent coordination strategies impact correctness?

3. **RQ3:** Does modular role separation improve code generation?

2 Background

LLM-based Code Generation. Neural code generation has progressed from early transformer models (Austin et al., 2021) to specialized systems like

Codex (Chen et al., 2021), StarCoder (Li et al., 2023), Code Llama (Roziere et al., 2023), and Qwen2.5-Coder (Qwen Team, 2024), achieving increasingly strong performance on benchmarks like HumanEval.

Multi-Agent Systems. ChatDev (Qian et al., 2023) and MetaGPT (Hong et al., 2023) apply role-based agent collaboration to software development, with specialized agents handling requirements, design, implementation, and testing. These systems demonstrate capabilities for high-level engineering workflows but focus on system design rather than low-level algorithmic correctness.

Iterative Refinement. Chain-of-thought prompting (Wei et al., 2022) and self-debugging (Chen et al., 2023) show that iterative refinement can improve LLM outputs. However, recent work (Huang et al., 2023) reveals that self-refinement sometimes degrades solution quality without external feedback, raising questions about when iteration helps versus hurts—a gap our comparative evaluation addresses.

Our work provides a systematic comparison of architectural paradigms (single-shot, iterative pipeline, multi-agent coordination) using controlled conditions to isolate the impact of architectural decisions on code correctness across task types and difficulty levels.

3 Methodology

We implement and evaluate three architectures of increasing complexity for LLM-based code generation: a *naive baseline*, a *single-agent pipeline*, and a *multi-agent system*. All approaches use Ollama for local model inference with temperature fixed at 0.0 for deterministic output. Table 1 summarizes the key differences.

Table 1: Comparison of the three approaches

	Naive	Single	Multi
LLM Calls	1	5–8	10–20+
Reasoning Phases	0	5	5+6+4
Feedback Loop	No	Yes	Yes
Max Iterations	0	3	3 (default)
Agent Identities	1	1	3

3.1 Naive Baseline

The naive baseline represents the simplest approach: single-shot code generation with no structured reasoning. The task specification (function signature and docstring) is formatted into a prompt,

the LLM generates code, and the output is extracted and executed. There is no analysis, planning, or refinement.

Figure 1: Naive baseline: single-pass generation

This approach serves as a control condition, measuring what the LLM can achieve without architectural support.

3.2 Single-Agent Pipeline

The single-agent approach introduces structured multi-phase reasoning using Lang-Graph (LangChain AI, 2024), while maintaining a unified agent identity. The pipeline consists of five phases with an iterative refinement loop.
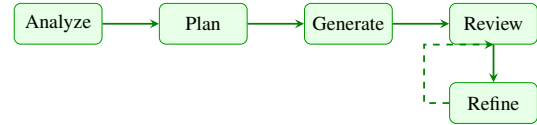

Figure 2: Single-agent pipeline with refinement loop (max 3 iterations)

Phase 1: Analysis. Extracts structured understanding from the task specification without proposing solutions: required behavior, input/output types, constraints, edge cases, ambiguities, and common pitfalls. The prompt explicitly forbids code generation.

Phase 2: Planning. Formulates an implementation strategy based on the analysis: algorithmic approach, step-by-step implementation sequence, edge case handling, data structures, and complexity analysis (time and space).

Phase 3: Generation. Synthesizes Python code guided by the accumulated context. The prompt enforces exact signature matching, no explanatory text, and comprehensive edge case handling. A robust parser extracts code from the LLM output.

Phase 4: Review. Evaluates the generated code through execution in an isolated namespace and static analysis via Radon (Campagna, 2024). The review distinguishes between correctness issues (which trigger refinement) and quality issues (which do not).

Phase 5: Refinement. When correctness issues are found, generates improved code based on review feedback. The loop terminates when correct

or after 3 iterations. Priorities: correctness first, then edge cases, then quality.

3.3 Multi-Agent System

The multi-agent approach distributes responsibilities across three specialized agents, following the sequential chain architecture. Each agent has a distinct identity and internal reasoning pipeline.
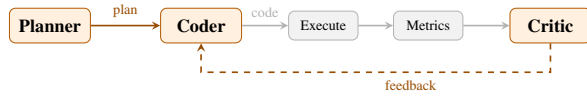

Figure 3: Multi-agent orchestration: Planner runs once, then Coder-Critic loop iterates until correct or max iterations

3.3.1 Planner Agent

The Planner is implemented as a *multi-node planning agent* in LangGraph (LangChain AI, 2024), with each node realized as a class in `planner/nodes/`. This modularization enables phase-level observability, targeted refinement, and explicit state tracking. The planner transforms a raw user request into a structured, machine-consumable plan (JSON) that serves as the contract between planning and coding, with a quality gate to prevent low-quality plans from propagating downstream.

Concretely, the Planner graph consists of six nodes:

1. **Intent Analysis**: Extracts the core intent, task type, domain context, success metrics, and assumptions.

2. **Requirements Engineering**: Produces testable functional and non-functional requirements, constraints, and edge cases.

3. **Architecture Design**: Decomposes the solution into components, selects patterns and data structures, and outlines algorithms with complexity notes.

4. **Implementation Planning**: Produces step-by-step coding instructions, validation rules, and representative test cases.

5. **Quality Review**: Scores plan completeness (0–10), enumerates issues and improvements, and sets an approval flag; we require a threshold of ≥ 8 to proceed.

6. **Consolidation**: Assembles a unified final plan artifact for the downstream Coder agent.

The Planner maintains an explicit state object containing each phase output (intent, requirements, architecture, implementation plan, and quality review), as well as metadata such as iteration count and accumulated errors. If the plan is not approved (quality score is below threshold), a bounded refinement loop routes feedback back to the *architecture* node (rather than re-running intent/requirements), for up to two re-planning iterations, after which the system emits a best-effort consolidated plan.

3.3.2 Coder Agent

The Coder agent, structured in `coder/` and its `nodes/` submodule, implements a six-phase, stateful pipeline that converts plans into executable code with strict plan and signature adherence, supports iterative refinement via critic feedback, enables modular ablations and advanced reasoning backends, and incorporates robust error handling for resilience against planning or LLM failures.

1. **Input Validation**: Validates function signature syntax (regex-based checks for `def`, parentheses, colon) and plan structure completeness, terminating early if critical requirements are missing.

2. **Edge Case Analysis**: Extracts type hints from the signature via regex parsing and identifies type-specific boundary conditions (empty collections, zero/negative numerics, null strings) informed by the implementation plan.

3. **Chain-of-Thought Generation**: Produces step-by-step reasoning that decomposes the problem into atomic steps, references identified edge cases, and plans algorithmic approach, incorporating feedback from previous iterations if refinement is active.

4. **Code Generation**: Synthesizes Python code by accumulating context from all prior phases (signature, plan, edge cases, CoT reasoning) plus any critic feedback and execution summaries from previous iterations.

5. **Code Validation**: Performs syntax validation via AST parsing and basic logic checks (infinite loop heuristics, unreachable code detection); accepts code with warnings but rejects on syntax errors.

6. **Code Optimization**: Applies LLM-based refinement to improve variable naming, algorithmic efficiency, and adherence to PEP 8 style conventions; falls back to validated code if optimization fails.

3.3.3 Critic Agent

The Critic agent, organized in `critic/` and its `nodes/` submodule, performs a four-phase, node-based review process with a shared state that tracks issues, scores, and structured feedback, enabling extensible validation criteria and clearly separating correctness from quality to guide systematic code refinement.

1. **Input Validation**: Verifies presence of required artifacts (code, plan, signature) and halts if prerequisites are missing.

2. **Correctness Analysis**: Evaluates functional correctness by cross-referencing code logic against plan requirements, interpreting execution error traces when available, and validating constraint satisfaction (time/space complexity, edge case handling).

3. **Quality Review**: Assesses maintainability via static metrics (cyclomatic complexity, maintainability index from Radon), code structure clarity, and adherence to coding standards.

4. **Feedback Synthesis**: Consolidates correctness and quality analyses into prioritized, actionable feedback for the Coder, explicitly prioritizing correctness issues over stylistic concerns to guide refinement effectively.

3.3.4 Orchestration

The master workflow, defined at the orchestration level, coordinates the agents by passing explicit state objects between them: (1) Planner creates the implementation plan; (2) Coder generates code; (3) code is executed and metrics computed; (4) Critic reviews and provides feedback; (5) if the Critic identifies issues, Coder regenerates with feedback. The loop continues until the Critic approves the code or maximum iterations (default: 3) are reached. This design ensures traceability, reproducibility, and ease of extension for future research.

3.4 Shared Infrastructure

All three approaches share common components ensuring fair comparison:

- **LLM Runtime**: Ollama interface with 8192-token context, 4096-token output limit, and automatic retry with backoff.

- **Code Extraction**: Parser handling markdown blocks (` ```python`), generic blocks, and raw functions; validates syntax via AST.

- **Execution**: Isolated namespace with captured stdout/stderr, exception handling, and function extraction verification.

- **Quality Metrics**: Radon-based static analysis computing Maintainability Index (0–100), Cyclomatic Complexity, LOC, and Halstead metrics.

4 Experimental Results

4.1 Dataset

We constructed a benchmark of 25 programming tasks spanning five domains: string manipulation, list operations, logic problems, mathematical computation, and data structures & algorithms (DSA). Tasks were sourced from HumanEval (Chen et al., 2021) and competitive programming platform Leet-Code (LeetCode, 2024). Each domain contains Easy (1 tasks), Medium (2 tasks), and Hard (2 tasks) difficulty levels, stratified by algorithmic complexity and edge case density. All tasks include function signatures, specifications, input-output examples, and comprehensive test suites, with an average of 13 test cases per task.

4.2 Model Selection and Configuration

To identify the most suitable base model for our pipeline comparisons, we first evaluated three code-specialized LLMs in a single-agent setting: CodeLlama-13B-Instruct (Roziere et al., 2023), DeepSeek-Coder-v2-16B-Instruct (DeepSeek-AI, 2024), and Qwen2.5-Coder-7B-Instruct (Qwen Team, 2024) (Table 3). Qwen2.5-Coder-7B achieved the highest overall pass rate (74.6%), outperforming DeepSeek-Coder-v2-16B (71.0%) and CodeLlama-13B (39.6%). Given its superior performance and parameter efficiency, we selected Qwen2.5-Coder-7B as the foundation for all subsequent architectural experiments.

All experiments utilize locally-hosted models via Ollama. For the naive and single-agent approaches, we use `qwen2.5-coder:7b-instruct`. For the multi-agent system, we employ a heterogeneous configuration:

Table 2: Architecture Comparison: Functional Correctness (%) by Domain and Difficulty. Best per category in **bold**.

| Architecture | By Domain | | | | | | By Difficulty | | |
	String	List	Logic	Math	Dsa	Avg	Easy	Med	Hard
Naive	**93.2**	71.4	67.6	56.9	83.5	**77.6**	78.5	85.8	**69.7**
Single-Agent	85.2	**72.9**	64.9	62.7	76.5	74.6	**89.2**	83.5	59.7
Multi-Agent	90.9	61.4	**75.7**	**64.7**	**85.9**	**77.6**	76.9	**90.6**	66.2
Δ (M-S)	+5.7	−11.4	+10.8	+2.0	+9.4	+3.0	−12.3	+7.1	+6.5

M=Multi, S=Single. Δ shows Multi-Agent improvement over Single-Agent (percentage points).

`qwen2.5-coder:7b-instruct` for the Planner and Coder agents, and `deepseek-coder-v2:16b` for the Critic. This configuration leverages the larger model's reasoning capabilities to provide independent, rigorous validation of the generated code.

Table 3: Single-Agent Performance Comparison Across LLM Models. Pass rates (%) by domain. Best in **bold**.

Model	String	List	Logic	Math	Dsa	Avg
CodeLlama-13B	71.6	27.1	21.6	21.6	34.3	39.6
DeepSeek-16B	86.4	65.7	**70.3**	60.8	65.9	71.0
Qwen2.5-7B	**85.2**	**72.9**	64.9	**62.7**	**76.5**	**74.6**

4.3 RQ1: Code Correctness and Quality Comparison

We evaluate the architectures by analyzing functional correctness across all the tasks and assessing code quality metrics on a subset of complex tasks.

4.3.1 Functional Correctness

Table 2 presents our main results for functional correctness. Surprisingly, the Naive and Multi-Agent architectures achieve the same overall pass rate of 77.6%, effectively tying for the best performance, while the Single-Agent approach follows closely at 74.6%. However, these aggregate metrics hide substantial performance differences across domains and difficulty levels.

Domain-Level Analysis: Performance varies substantially by task domain. Multi-Agent demonstrates clear advantages in algorithmically complex domains: DSA (85.9%, +9.4pp over Single), Logic (75.7%, +10.8pp over Single), and Math (64.7%, +2.0pp over Single). However, it underperforms on Lists tasks (61.4%, −11.4pp over Single). The Naive baseline achieves the highest Strings performance (93.2%), suggesting pattern-matching tasks do not benefit from complex reasoning architectures.

Difficulty-Level Analysis: The architectures show distinct profiles across difficulty. Single-Agent excels on Easy tasks (89.2%, +10.7pp over Naive), likely due to its systematic analysis phase. Multi-Agent demonstrates value on Medium (90.6%, +7.1pp over Single) and Hard tasks (66.2%, +6.5pp over Single), where Planner-Coder-Critic coordination enables sophisticated decomposition.

Notably, Naive outperforms both on Hard tasks (69.7%). We attribute this counter-intuitive result to "over-refinement," where the complex validation loops in agentic systems inadvertently introduce regressions on the most difficult problems.

4.3.2 Code Quality Analysis

Beyond correctness, we analyzed the code quality of solutions generated by the three architectures for five hard difficulty tasks (longest_substring_without_repeating, triples_sum_to_zero, find_median_sorted_arrays, solve_n_queens, largest_prime_factor) using four key metrics: Maintainability Index (MI), Cyclomatic Complexity (CC), Lines of Code (LOC), and Halstead Volume (HV). Table 4 summarizes the results.

The Naive baseline generates the simplest code, with the lowest cyclomatic complexity (5.40) and lines of code (18.40), likely reflecting its tendency to produce direct, but sometimes less robust solutions. The Single-Agent and Multi-Agent architectures generate more complex code (CC 7.20 and 8.20 respectively) with higher Halstead volumes, reflecting their more comprehensive handling of edge cases and input validation. Crucially, despite the added complexity, the agentic approaches maintain or improve the Maintainability Index (Single Agent: 61.61, Multi Agent: 61.41) compared to Naive (59.43), suggesting that the additional logic is structured effectively. The Multi-Agent system's higher complexity correlates with its superior performance on these hard tasks,

indicating that the problem difficulty necessitates more sophisticated logic that simple solutions cannot capture.

Table 4: Average Code Quality Metrics on Hard Tasks. Arrows indicate desired direction (↑ higher is better, ↓ lower is better).

Architecture	MI (↑)	CC (↓)	LOC	HV
Naive	59.43	**5.40**	**18.40**	**147.09**
Single-Agent	**61.61**	7.20	23.20	218.84
Multi-Agent	61.41	8.20	26.40	273.83

4.4 RQ2: Impact of Coordination Strategies

The coordination strategies employed in the Multi-Agent system present a clear trade-off between computational efficiency and algorithmic robustness. While the increase in LLM calls per task (10-20 vs. 5-8 for the Single-Agent approach) yields diminishing returns on simple tasks, it proves essential in high-complexity domains. We observe that the overhead of context switching and message passing between agents actively degrades performance on tasks where direct generation suffices. For simple pattern-matching, the iterative consensus mechanism creates noise rather than signal, leading to the performance regression seen in the Easy category. Conversely, for logic-heavy tasks, this same mechanism acts as a necessary filter; the Planner-Critic loop effectively catches conceptual errors that a single pass misses, justifying the latency.

Furthermore, the adoption of a heterogeneous configuration significantly enhances the validation process. By separating the critique function (DeepSeek-V2) from the generation engine (Qwen2.5), the architecture introduces a layer of independence. This structural diversity mitigates the risk of confirmation bias where a homogeneous model validates its own hallucinations. This suggests that effective coordination relies as much on model diversity as it does on iterative loops.

4.5 RQ3: Effect of Role Separation

Modular role separation improves code generation conditionally rather than universally. The benefits are concentrated in tasks necessitating careful planning (e.g., Logic constraints, DSA optimization) and systematic validation. For simpler tasks, however, this form of architectural layering introduces overhead that may be counterproductive. Our results indicate that domains such as string manipulation or trivial edge-case handling derive little benefit from multi-stage reasoning; in these cases, the Planner's detailed decomposition can over-constrain the solution space, while the Critic's feedback mechanism may reject valid yet unconventional solutions.

At the same time, however, the quality analysis reveals that role separation successfully regulates code complexity. Although the Multi-Agent system yields solutions with the highest Cyclomatic Complexity and Halstead Volume, it maintains a Maintainability Index (61.41) comparable to the Single-Agent approach (61.61). This indicates that the specialized agents—structuring, implementing, and reviewing—ensure that the increased complexity required for hard tasks is compartmentalized, preventing it from degrading the overall maintainability of the codebase.

5 Discussion

Our results highlight a trade-off between *agentic structure* and *raw efficiency*. While the Naive and Multi-Agent settings reach the same aggregate functional correctness (77.6%), they do so with different operating regimes. Multi-Agent orchestration yields the clearest gains on reasoning-intensive tasks (Logic +10.8pp, DSA +9.4pp), suggesting that explicit decomposition and critique are most beneficial when problems require multi-step constraint satisfaction rather than surface pattern matching.

Conversely, for simpler tasks, additional coordination and repeated LLM calls introduce overhead and opportunities for error accumulation, which can reduce accuracy relative to the Naive baseline. This finding is particularly relevant under *low-compute* and latency-constrained settings, where increased inference budgets may be impractical despite potential quality benefits (Strubell et al., 2019; Bommasani et al., 2021).

6 Limitations

Scale and representativeness. Our evaluation spans 25 tasks and three model families; while sufficient for a controlled comparison, it limits statistical power and the generalizability of domain-specific conclusions.

Compute realism. Multi-agent pipelines can be substantially more expensive than single-pass generation, which may hinder adoption in realistic deployment contexts (e.g., on-device or small-server

inference, or strict budget/latency constraints).

Operational constraints. We do not model several practical challenges of production code generation systems, including prompt/plan drift across iterations, brittle structured-output parsing, evolving dependency ecosystems, and the need for robust failure-handling and monitoring in long-running agentic workflows.

Future work. To address these limitations and improve practical usability, we propose: (i) larger-scale benchmarking on established suites (e.g., APPS, CodeContests) with stratified analysis by task type and difficulty; (ii) compute-aware evaluation reporting latency and token cost alongside correctness, and exploring early-exit/anytime stopping criteria; (iii) adaptive routing that selects Naive/Single-/Multi-Agent policies per task using lightweight predictors; (iv) heterogeneous agent configurations (e.g., smaller models for intent/requirements and larger models for architecture) to better balance cost and quality; and (v) robustness improvements, including resilient JSON extraction, explicit error taxonomies, and ablations isolating which planning/review nodes contribute most.

7 Conclusion

Overall, our findings support *adaptive architecture selection*: Naive for rapid prototyping and pattern-heavy tasks; Single-Agent pipelines for general-purpose development with systematic edge-case checking; and Multi-Agent systems for algorithmic reasoning and specification-heavy problems where planning and critique justify their higher compute cost. We view compute-efficiency and deployment robustness as first-class objectives for future agentic code-generation research.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, and 1 others. 2021. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*.

Michele Campagna. 2024. Radon: A python tool to compute code metrics. https://radon.readthedocs.io/.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, 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. *arXiv preprint arXiv:2304.05128*.

DeepSeek-AI. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. https://github.com/deepseek-ai/DeepSeek-Coder-V2.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Ceyao Zhang, Zili Wang, Steven Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, and 1 others. 2023. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*.

Jie Huang, Xinyun Chen, Swaroop Mishra, Huaixiu Steven Zheng, Adams Wei Yu, Xinying Song, and Denny Zhou. 2023. Large language models cannot self-correct reasoning yet. *arXiv preprint arXiv:2310.01798*.

LangChain AI. 2024. Langgraph: Building stateful, multi-actor applications with llms. https://langchain-ai.github.io/langgraph/.

LeetCode. 2024. Leetcode. https://leetcode.com/.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: May the source be with you! *arXiv preprint arXiv:2305.06161*.

Chen Qian, Xin Cong, Wei Liu, Cheng Yang, Weize Chen, Yusheng Su, Yufan Dang, Jiahao Li, Juyuan Liu, Dahai Tang, and 1 others. 2023. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*.

Qwen Team. 2024. Qwen2.5-coder technical report. https://qwenlm.github.io/blog/qwen2.5-coder/.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, and 1 others. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and policy considerations for deep learning in NLP. *arXiv preprint arXiv:1906.02243*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.