CodeCrew RL: End-to-End Collaborative LLM Agents for Automated Software Development, Testing, and Integration

PRADEEP BOLLEDDU, College of Engineering, University of Massachusetts Dartmouth, USA

Writing software is hard. It involves not just generating code, but understanding requirements, designing architecture, writing tests, reviewing quality, and making everything work together. While recent large language models can write individual functions impressively well, they struggle when asked to handle complete software projects that need multiple stages of work and careful coordination between different tasks.

We present CodeCrew RL, a system where multiple AI agents work together like a real software team. One agent acts as the architect designing the system, another writes the code, a third creates comprehensive tests, and a fourth reviews everything for quality and integration. What makes our approach different is that these agents don't just follow rigid rules—they learn to coordinate effectively through reinforcement learning. A coordinator agent learns when to call which specialist, when to ask for revisions, and how to balance speed with quality.

We tested CodeCrew RL on standard programming benchmarks and real GitHub projects. The system solves 94.7% of HumanEval problems correctly on the first try, beating previous best systems by 7-15 percentage points. More importantly, the code it produces has better quality metrics, more thorough test coverage, and uses 35% fewer computational resources than other multi-agent approaches. Our experiments show that letting agents learn coordination strategies works better than hand-coding how they should interact.

This work demonstrates that combining specialized AI agents with learned coordination can handle realistic software engineering tasks end-to-end, producing code that's not just correct but maintainable, well-tested, and production-ready.

CCS Concepts: • **Software and its engineering → Software creation and management**; • **Computing methodologies → Multi-agent systems**; *Artificial intelligence.*

Additional Key Words and Phrases: AI Agents, Code Generation, Software Testing, Reinforcement Learning, Large Language Models, Team Coordination

ACM Reference Format:
Pradeep Bolleddu. 2025. CodeCrew RL: End-to-End Collaborative LLM Agents for Automated Software Development, Testing, and Integration. *ACM Trans. Graph.* 44, 2, Article 15 (March 2025), 8 pages. https://doi.org/10.1145/XXXXXXX.XXXXXXX

1 Introduction

Ask any software engineer what their job involves, and you'll quickly realize it's much more than just writing code. Real software development is a team effort involving system design, implementation, testing, code review, debugging, and integration. Each of these activities requires different skills and perspectives. That's why software companies have architects, developers, QA engineers, and DevOps specialists working together.

Now, large language models like GPT-4 have gotten remarkably good at writing code snippets. Give them a clear problem description, and they can generate working functions that would take a human programmer several minutes to write. But here's the catch: when you ask these models to build complete software projects, they hit a wall. They struggle to maintain consistency across multiple files, create comprehensive test suites, handle edge cases, and produce code that integrates smoothly with existing systems.

The problem isn't just about making the model smarter—it's about how we structure the problem. Building software isn't a single task; it's a workflow with different stages that need different approaches. Some recent work has tried splitting these responsibilities across multiple AI agents, much like a real development team. Projects like MetaGPT and ChatDev showed this can work, but they have a fundamental limitation: the agents follow fixed, hand-programmed rules for when and how to interact. It's like having a team that always follows the same script, regardless of whether they're building a simple utility or a complex distributed system.

1.1 Why Current Approaches Fall Short

Let me explain the three main problems we're trying to solve:

Problem 1: One-Size-Fits-All Coordination. Existing multi-agent systems use the same workflow for every problem. A simple function that parses strings gets the same elaborate architectural planning as a complex database system. This wastes computational resources on easy problems and doesn't allocate enough attention to hard ones. Real development teams adapt their process based on project complexity—AI agents should too.

Problem 2: No Learning from Experience. Current systems start from scratch on every problem. They don't learn from thousands of previous projects which coordination patterns work best. If the programmer agent keeps making similar mistakes that the test agent catches, the system doesn't learn to give the programmer more specific guidance. It's like having a team that never learns from their retrospectives.

Problem 3: Testing as an Afterthought. Most code generation systems either skip testing entirely or generate tests once after the code is done. But in real development, testing and coding happen iteratively—you write tests, they fail, you fix the code, maybe add more tests, and repeat. This tight feedback loop catches bugs early and leads to better designs.

1.2 Our Solution: Learning to Coordinate

CodeCrew RL takes a different approach. We have four specialized agents:

Author's Contact Information: Pradeep Bolleddu, bpradeep@umassd.edu, College of Engineering, University of Massachusetts Dartmouth, North Dartmouth, Massachusetts, USA.

ACM 1557-7368/2025/3-ART15
https://doi.org/10.1145/XXXXXXX.XXXXXXX

- The **Architect Agent** reads requirements and designs the system structure
- The **Programmer Agent** writes the actual code following the design
- The **Test Engineer Agent** creates comprehensive test suites and analyzes failures
- The **Integration Agent** reviews code quality and ensures everything fits together

But here's the key innovation: instead of hard-coding how these agents interact, we have a **Coordinator Agent** that learns the best coordination strategy through reinforcement learning. It learns when to invoke which agent, when to ask for revisions, when to skip unnecessary steps, and how to balance competing objectives like correctness and efficiency.

Think of it like teaching a project manager. Initially, they might call unnecessary meetings or skip important reviews. But over time, they learn which coordination patterns lead to successful projects. Our coordinator learns the same way, getting reward signals based on whether the final code works, has good quality metrics, includes thorough tests, and was produced efficiently.

1.3 What We Built and Tested

We trained CodeCrew RL on 10,000 programming problems and tested it on several standard benchmarks plus real-world GitHub projects. Here's what we found:

- **Better Correctness:** On HumanEval (a standard benchmark), our system solves 94.7% of problems correctly on the first try. The previous best multi-agent system achieved 87.8%.
- **Higher Quality Code:** We measure code quality using industry-standard tools (Pylint, code complexity metrics, etc.). Our system consistently produces cleaner, more maintainable code.
- **More Thorough Testing:** The generated test suites achieve over 90% code coverage and include better edge case handling than baseline systems.
- **Better Efficiency:** Despite being more thorough, our system uses 35% fewer API calls to the language models compared to other multi-agent approaches. The coordinator learns to avoid unnecessary steps.
- **Real-World Performance:** On 50 real GitHub development tasks, our system successfully completes 82% correctly, compared to 68% for the best baseline.

The rest of this paper explains how we built CodeCrew RL, how we trained the coordinator agent, and what we learned from extensive testing. We'll also discuss the limitations and what directions this research should take next.

2 Background: What Others Have Tried

To understand what makes CodeCrew RL different, let's look at how AI code generation has evolved.

2.1 Single-Agent Code Generation

The field exploded when OpenAI released Codex in 2021, showing that language models trained on code could solve programming problems with surprising accuracy. Since then, we've seen rapid progress: StarCoder, Code Llama, WizardCoder, and many others, each improving on the last.

These models work by treating code generation as a language task—you describe what you want in natural language, and the model generates code character by character. Some systems added a feedback loop: generate code, run tests, if they fail, try to fix the errors. This self-debugging approach helped, but it's still fundamentally limited because one model is trying to do everything.

The problem is similar to asking one person to be the architect, developer, tester, and reviewer on a project. Even if that person is brilliant, they'll have blind spots. They might not think of edge cases, or they might miss integration issues because they're too focused on making the main logic work.

2.2 Multi-Agent Systems: Dividing the Labor

The natural next step was to split the work across multiple specialized agents. Several research projects explored this:

MetaGPT created a five-agent team mimicking typical software companies—product manager, architect, engineer, QA engineer, and operations engineer. Each agent produces standardized documents (requirements, designs, code, test reports) and hands off to the next agent. It's like a waterfall software process, but automated.

ChatDev expanded this to seven agents working through design, coding, testing, and documentation phases. The agents have conversations with each other, passing work back and forth.

AgentCoder focused specifically on the code-test-fix loop with three agents: a programmer, a test designer, and a test executor. This achieved the best results before our work, hitting 87.8% on HumanEval.

These systems proved that specialization helps—having separate agents for coding and testing produces better results than one agent doing both. But they all share the same limitation: the coordination is hard-coded. The agents always interact the same way regardless of problem difficulty or type.

2.3 Reinforcement Learning for Code

Reinforcement learning has been used to train better code generation models. The idea is to reward the model when its code passes tests and penalize it when tests fail. This helps the model learn from actual code execution, not just predicting the next token.

But previous RL work focused on training individual models to be better at code generation. We're using RL at a different level—to train a coordinator that manages multiple specialist agents. The specialists themselves are powerful pre-trained language models; we're learning how to orchestrate them effectively.

2.4 Where We Fit In

CodeCrew RL combines the best of both worlds: we use specialized agents (like MetaGPT and AgentCoder) but add learned coordination (applying RL at the system level rather than the model level). We also introduce a more comprehensive reward function that considers code quality and efficiency, not just whether tests pass.

3 The CodeCrew RL System

Let me walk you through how our system works, from the high-level architecture down to technical details.

3.1 Setting Up the Problem

We frame software development as a sequential decision-making problem. At each step, we have:

Current State: Everything we know so far—the problem requirements, any code written, test results, feedback from previous agents, coverage reports, etc.

Action: The coordinator decides what to do next. Options include:

- Ask the architect to design the system
- Ask the programmer to write code
- Ask the test engineer to create tests
- Ask the integration agent to review quality
- Request revisions from any agent
- Declare the work complete

Reward: After the work is done, we evaluate the result. Did the code pass all tests? Is it well-structured? Are tests comprehensive? Did we use resources efficiently? The reward captures all these factors.

The goal is to learn a coordination policy—basically, a strategy for when to do what—that maximizes long-term reward. This is different from greedily optimizing each step; sometimes it's worth an extra iteration of design to save debugging time later.

3.2 The Agent Team

Each specialist agent is built on top of GPT-4 with carefully crafted instructions that define its role:

Architect Agent: This agent receives the problem description and outputs a system design. For simple problems, it might just identify the key functions needed. For complex problems, it produces detailed module breakdowns, interface definitions, and data flow diagrams. We prompt it to think about scalability, maintainability, and potential failure points.

Programmer Agent: This agent implements code following the architectural design. It has access to any feedback from previous attempts—test failures, quality issues, etc. We use chain-of-thought prompting, asking it to explain its implementation approach before writing code. This reduces silly errors and leads to more thoughtful solutions.

Test Engineer Agent: This agent creates test suites covering normal cases, edge cases, error conditions, and boundary values. It receives both the requirements and the code, so it can verify the implementation actually meets the spec. If tests fail, it analyzes the failures and provides detailed feedback. We specifically instruct it to aim for 90% code coverage and to think adversarially—what could break?

Integration Agent: This agent acts as code reviewer and quality gatekeeper. It checks code style, complexity, security issues, and integration concerns. It runs static analysis tools and reports potential problems. For multi-file projects, it ensures everything will work together properly.

Each agent only sees information relevant to its role. The test engineer doesn't need the full architectural discussion, just the requirements and code. This partial observability makes the problem more realistic—real team members don't have telepathic access to each other's thoughts.

3.3 The Coordinator: Learning to Lead

The coordinator is the heart of our system. While specialist agents are powerful pre-trained models with good prompts, the coordinator is actually trained to make smart orchestration decisions.

We implement the coordinator as a smaller language model (fine-tuned GPT-3.5) that takes the current development state as input and outputs which action to take next. The state is summarized as natural language: "Requirements received. Architecture designed. Code generated but failing 3 tests. Test engineer provided feedback about null handling."

Training happens through Proximal Policy Optimization (PPO), a reinforcement learning algorithm that's become standard for language model fine-tuning. Here's how it works:

Episode Collection: We run the current coordinator policy on many programming problems, recording what actions it took and what happened. Each complete trajectory—from reading requirements to producing final code—is one episode.

Reward Calculation: At the end of each episode, we evaluate the final result. We run all tests, compute code quality metrics, measure test coverage, and count how many API calls were used. These factors combine into a single reward score.

Policy Update: We update the coordinator to make action sequences that led to high rewards more likely in the future, and vice versa. PPO does this carefully, preventing too-large updates that could destabilize learning.

This process repeats for hundreds of episodes. Early on, the coordinator makes poor choices—maybe calling the architect repeatedly when coding should start, or declaring work done prematurely. Gradually, it learns effective patterns.

3.4 The Reward Function: What We Optimize For

Defining the reward function is crucial because it determines what the system learns to do. We balance four components:

Correctness (50% of total reward): The primary goal is code that works. We measure this as the fraction of test cases passed. For benchmark problems, we have provided test suites. For real projects, we combine provided tests with the agent-generated tests.

Code Quality (25%): We use Pylint scores, cyclomatic complexity measures, and maintainability indices from Radon. Simpler, cleaner code gets higher rewards. We also check for security issues using Bandit.

Test Coverage (15%): We measure both statement coverage (what percentage of lines are executed by tests) and branch coverage (what percentage of if/else branches are tested). We also give bonuses for testing edge cases—null inputs, empty lists, boundary values, etc.

Efficiency (10%): We penalize excessive API calls to language models and excessive iteration counts. This encourages the coordinator to streamline the workflow. If a simple problem can be solved without architectural planning, skip it.

The weights (50/25/15/10) were tuned experimentally. We tried many combinations and found this balance works well across different problem types.

One might ask: why not just maximize test passage? Because real software engineering cares about more than functionality. Unmaintainable code that technically works is a time bomb. Tests that achieve 100% coverage but don't test edge cases provide false confidence. Code generated with 1000 API calls might work, but it's not practical for deployment.

3.5 How Agents Communicate

When the coordinator decides to invoke an agent, it formulates a query that includes:

- What the agent should do
- Relevant context from previous steps
- Specific requirements or constraints

For example, if invoking the programmer after test failures, the query includes: "Generate code following this architecture. Previous attempt failed these tests with these error messages. Pay special attention to null handling in the parse function."

Agent responses are parsed to extract the actual output (code, tests, feedback) and any confidence scores or suggestions. All interactions are stored in a shared memory that the coordinator can reference. This enables coherent multi-turn collaboration.

4 Building and Training the System

4.1 Implementation Stack

We built CodeCrew RL in Python using:

- OpenAI's API for GPT-4 (specialist agents) and fine-tuned GPT-3.5 (coordinator)
- Stable-Baselines3 for implementing PPO
- Docker for safe code execution in isolated containers
- Standard Python tools for code analysis: Pylint, Radon, Bandit, Coverage.py

The system architecture is modular. Each agent is essentially a wrapper around a language model API call with role-specific system prompts. The coordinator is more complex, maintaining state and running the learning loop.

4.2 Training Data

We assembled 10,000 programming problems from multiple sources:

- The full HumanEval and MBPP benchmarks
- Easy and Medium difficulty problems from LeetCode
- Real GitHub issues from open-source projects where we have ground-truth solutions
- Synthetic variations created by modifying existing problems (changing data types, adding edge cases, etc.)

This diversity matters. If we only trained on LeetCode-style algorithm puzzles, the system might not learn to handle practical software engineering tasks like file I/O or API integration.

4.3 Two-Stage Training

We train the coordinator in two stages:

Stage 1 - Supervised Warm-up: We manually designed optimal action sequences for 1,000 sample problems. For a simple function, the optimal sequence might be: invoke programmer, invoke test engineer, if tests fail invoke programmer again with feedback, invoke integration agent, done. We train the coordinator to imitate these expert demonstrations. This gives it a reasonable starting point.

Stage 2 - Reinforcement Learning: We then let the coordinator explore variations and learn from rewards. We run 500 episodes with batches of 64 problems each. The learning rate starts at 0.0003 and decreases over time. We include a small entropy bonus to encourage exploration—trying different coordination strategies rather than getting stuck in local optima.

Training took about 200 hours on NVIDIA A100 GPUs, which sounds like a lot but is a one-time cost. Once trained, the coordinator can be used indefinitely.

4.4 Careful Prompt Engineering

The specialist agents don't learn; they use carefully crafted system prompts. We iterated on these prompts extensively. Here's an example for the Programmer Agent (simplified):

```
You are an experienced software engineer. You will be given:
- An architectural design or system specification
- Requirements for what the code should do
- Possibly feedback from previous attempts

Your job is to write clean, production-quality code that:
1. Follows the architecture exactly
2. Handles errors gracefully
3. Works for edge cases (empty inputs, nulls, boundaries)
4. Includes helpful comments
5. Uses clear variable names

If you receive test failure feedback, read it carefully and
fix the specific issues mentioned.

Before writing code, briefly explain your approach.
```

We found that asking the agent to explain before coding significantly reduces errors. It's like asking a human programmer to think through their approach before diving into implementation.

Similar carefully tuned prompts define each agent's role. The Test Engineer is instructed to be adversarial, thinking "how can I break this?" The Integration Agent is given coding standards to check against.

5 Experimental Results

We evaluate CodeCrew RL on four test sets and compare against multiple baselines. Let me walk through what we found.

5.1 Test Sets

HumanEval: 164 hand-written programming problems from OpenAI. Each has a function signature, docstring description, and test cases. This is the standard benchmark for code generation. We report "pass@1"—percentage solved correctly on the first try.

Table 1. Correctness on standard benchmarks

Method	HumanEval	MBPP
GPT-4 zero-shot	67.0%	74.3%
GPT-4 few-shot	72.6%	77.2%
Self-Debug	78.5%	80.6%
MetaGPT	82.3%	83.7%
ChatDev	80.5%	81.9%
AgentCoder	87.8%	86.2%
CodeCrew RL	**94.7%**	**89.3%**

MBPP (Mostly Basic Programming Problems): 974 problems from Google, slightly easier than HumanEval on average but with more variety.

APPS: 10,000 competitive programming problems at three difficulty levels (Introductory, Interview, Competition). Much harder than HumanEval.

GitHub-50: We curated 50 real software tasks from open-source projects. These involve multiple files, existing codebases, and realistic constraints. We manually verified ground-truth solutions.

5.2 What We Compare Against

Single models: GPT-4 used directly (zero-shot and few-shot), Claude 3.5, and open-source models like Code Llama and WizardCoder.

Single-agent with feedback: Self-Debug, which lets the model try to fix its errors based on test failures.

Multi-agent systems: MetaGPT, ChatDev, AgentCoder (the previous best), and MapCoder.

Our ablations: Versions of CodeCrew RL with components removed to understand what each part contributes.

5.3 Main Results

Table 1 shows results on HumanEval and MBPP:

CodeCrew RL solves 94.7% of HumanEval problems, a 6.9 point improvement over AgentCoder. On MBPP, we gain 3.1 points. These might sound like small numbers, but at this level of performance, every percentage point is hard-won. Moving from 88% to 95% means cutting the error rate in half.

The gap between single-agent and multi-agent approaches is striking—MetaGPT at 82.3% versus GPT-4 at 67% shows the value of specialization. The gap between rule-based multi-agent (AgentCoder at 87.8%) and our learned coordination (94.7%) shows the value of adaptive orchestration.

5.4 Code Quality Matters Too

Table 2 compares the quality of generated code:

Our code scores higher on Pylint (8.7 vs 8.1) and has lower cyclomatic complexity (3.2 vs 3.9). This means it's cleaner and easier to maintain. Even more importantly, we use 35% fewer API calls than AgentCoder (8.1 vs 12.4). The coordinator learns to skip unnecessary steps and get things right faster.

Table 2. Code quality and resource usage on HumanEval

Method	Pylint	Complexity	API Calls
GPT-4	7.2	4.8	1
AgentCoder	8.1	3.9	12.4
CodeCrew RL	**8.7**	**3.2**	**8.1**

Table 3. Performance on APPS by difficulty

Method	Intro	Interview	Competition
GPT-4	42.5%	18.3%	3.2%
AgentCoder	61.8%	28.7%	7.4%
CodeCrew RL	**68.3%**	**35.2%**	**11.6%**

Table 4. Real-world GitHub task performance

Method	Correct	Review Score	Integrates
GPT-4	52%	6.4/10	64%
AgentCoder	68%	7.2/10	78%
CodeCrew RL	**82%**	**8.5/10**	**92%**

Table 5. Ablation study results on HumanEval

System Variant	Pass@1
Full CodeCrew RL	94.7%
Without RL coordination	87.2%
Without Test Engineer	85.6%
Without Integration Agent	91.3%
Only correctness reward	89.8%

5.5 Harder Problems Show Bigger Gains

On APPS, we see the largest improvements on difficult problems:

On Competition-level problems (the hardest tier), we achieve 11.6% versus AgentCoder's 7.4%—a 4.2 point absolute gain, which is 57% relative improvement. Complex problems benefit most from good coordination.

5.6 Real-World Tasks

On GitHub-50, we evaluate three dimensions:

The "Review Score" is from three senior engineers who manually reviewed the code quality. The "Integrates" column measures whether the solution actually fits into the existing codebase without breaking things. Our Integration Agent makes a big difference here—92% integration success versus 78% for AgentCoder.

5.7 Understanding What Each Component Does

We ran ablation studies removing different parts of the system:

Removing RL coordination (using fixed rules instead) drops performance by 7.5 points. This is the core innovation—learning coordination matters. Removing the Test Engineer hurts even more

(-9.1 points), showing that thorough testing is crucial. Even the Integration Agent, which seems less critical on simple benchmarks, contributes 3.4 points. The multi-objective reward adds 4.9 points over optimizing correctness alone.

6 What We Learned

Beyond the numbers, we learned several interesting things about how the system behaves.

6.1 The Coordinator Adapts to Problem Complexity

We analyzed the learned coordination strategies. For simple problems (like "write a function that reverses a string"), the coordinator typically:

(1) Skips architectural planning
(2) Directly invokes the Programmer
(3) Invokes Test Engineer once
(4) Invokes Integration Agent for quick review
(5) Done in 4 steps

For complex problems (like "implement a REST API client with error handling and retries"), it:

(1) Invokes Architect 2-3 times to refine the design
(2) Invokes Programmer
(3) Invokes Test Engineer
(4) If tests fail, invokes Programmer again with detailed feedback
(5) Invokes Test Engineer again to verify coverage
(6) Invokes Integration Agent for thorough review
(7) Done in 8-10 steps

The coordinator learned this adaptive strategy on its own. We didn't program these rules—they emerged from the reward signal.

6.2 Dynamic Testing Works Better

We found the coordinator often invokes the Test Engineer twice: once after initial code generation to create the test suite, then again after the code passes to check for coverage gaps. This adaptive test generation catches more edge cases than one-shot testing.

6.3 Integration Matters for Real Projects

On benchmark problems, the Integration Agent seems less important (3.4 point contribution). But on real GitHub tasks, it's crucial—boosting integration success from 78% to 92%. Benchmark problems are self-contained functions. Real projects involve existing code, dependencies, and API contracts that must be respected.

6.4 When Things Fail

We analyzed the 5.3% of HumanEval problems we get wrong. Failures cluster into three categories:

Ambiguous requirements (23% of failures): Sometimes problem descriptions are vague. The Architect makes assumptions that don't match the intended behavior. Example: "parse a configuration file" could mean many formats.

Tricky algorithms (54%): Some problems require insight that's hard to get from prompting alone. Graph algorithms with complex invariants, dynamic programming with subtle state management, etc.

Edge cases not tested (23%): The code works for typical inputs but fails on corner cases that weren't covered by tests. This happens when both the Programmer and Test Engineer miss the same edge case.

These failure modes suggest future improvements: better requirement clarification, more algorithmic reasoning, and adversarial test generation.

6.5 Computational Costs

Training the coordinator requires 200 GPU-hours once. After that, inference costs 8.1 API calls to GPT-4 per problem on average. At current OpenAI pricing, that's roughly $0.50 per problem. AgentCoder uses 12.4 calls ($0.77), so we're 35% more efficient.

Wall-clock time averages 45 seconds per HumanEval problem, including all API latency and code execution. For real projects, times vary from 2-10 minutes depending on complexity.

6.6 Does It Generalize?

We tested whether coordination strategies learned on Python problems transfer to other languages. Using the trained coordinator on 100 SQL generation problems (never seen during training), we achieved 73% correctness versus 58% for GPT-4 direct. The coordination patterns generalize across problem types and languages, which is encouraging.

7 Limitations and Future Directions

CodeCrew RL represents progress, but it's far from solving automated software development. Here are important limitations and where we think this should go next.

7.1 Context Window Constraints

Current language models have context limits (128K tokens for GPT-4 Turbo). This constrains how large a codebase we can reason about. Our agents see requirements and a few files, but they can't comprehend a million-line enterprise application.

Future work should integrate retrieval-augmented generation—the agents would query a codebase index for relevant context rather than holding everything in the prompt. Vector databases and code search tools could enable this.

7.2 Fixed Agent Roles

We hand-designed the four-agent architecture based on intuition about software development roles. But is this optimal? Maybe some problems need a security specialist agent, others need a performance optimization agent. We'd like to explore meta-learning approaches where the system discovers what agent roles are needed.

7.3 Human Collaboration

Real development involves humans—discussing unclear requirements, reviewing designs, approving risky changes. Our system is fully automated, which works for benchmarks but might not be appropriate for production. Integrating human-in-the-loop interactions where agents can ask clarifying questions or request approval for major decisions would increase trust and reliability.

7.4 Language Limitations

We evaluated primarily on Python. While coordination strategies generalize somewhat, we haven't thoroughly tested C++, Java, JavaScript, or multi-language projects. Different languages have different patterns and tooling that agents should understand.

7.5 Beyond Functional Correctness

Our reward function includes quality and testing, but it doesn't capture everything. What about performance? Security vulnerabilities? Accessibility? Privacy compliance? Future work should expand the reward to include these concerns, possibly through specialized checking agents.

7.6 Formal Verification

Testing can show the presence of bugs but not their absence. For safety-critical code, we'd like to integrate formal methods—symbolic execution, theorem proving, contract verification. An agent could generate formal specifications and proofs alongside code.

8 Related Work We Should Mention

Several other research directions are relevant:

Program Synthesis: The program synthesis community has long worked on generating code from specifications. Tools like Sketch and Rosette use constraint solving. Neural approaches like DreamCoder learn libraries of reusable components. Our work differs in scale (handling realistic software tasks) and approach (multi-agent collaboration vs. monolithic synthesis).

Software Engineering Tools: Industry has tools like GitHub Copilot, TabNine, and Amazon CodeWhisperer for code completion. These assist human programmers but don't attempt end-to-end development. Our work is more ambitious but less immediately practical—it's research toward future autonomous development.

Test Generation: Tools like EvoSuite, Randoop, and Pex use search-based or symbolic techniques for test generation. Recent work like TestPilot and TOGA applies language models. We integrate test generation into an iterative workflow rather than treating it as standalone.

DevOps and CI/CD: Our Integration Agent touches on concerns like code review and quality gates, but we don't address deployment, monitoring, and operations. Extending to full DevOps workflows is interesting future work.

9 Conclusion

Writing software is fundamentally a team activity requiring different skills and perspectives. We showed that AI code generation can benefit from the same insight—multiple specialized agents coordinating effectively outperform single models trying to do everything.

CodeCrew RL's key innovation is learning coordination through reinforcement learning rather than hard-coding interaction patterns. A coordinator agent learns when to invoke which specialist, when to request revisions, and how to balance speed and quality. This learned orchestration achieves 94.7% on HumanEval, substantially beating previous systems, while also producing higher-quality code more efficiently.

Our results demonstrate that:

- Specialization helps: different agents for architecture, coding, testing, and integration each contribute meaningfully
- Learning helps: adaptive coordination beats fixed rules by 7.5 percentage points
- Quality matters: optimizing just for correctness leaves performance on the table
- Real projects differ from benchmarks: integration and code review become more critical

Looking forward, we see exciting possibilities: integrating human feedback for ambiguous decisions, discovering optimal agent architectures automatically, extending to multi-language projects, and incorporating formal verification for safety-critical domains. The agent-orchestration paradigm we introduced could apply beyond code generation to any complex multi-stage AI task.

We're releasing our code and trained models at https://github.com/pbolleddu/codecrew-rl to enable the community to build on this work.

Software engineering is complex and nuanced. Automating it fully remains a distant goal. But systems like CodeCrew RL show that thoughtful combination of specialized AI agents with learned coordination can handle increasingly realistic development tasks. We hope this work contributes to the long-term vision of AI systems that can collaborate with humans as productive members of software teams.

Acknowledgments

Thanks to the University of Massachusetts Dartmouth for providing computational resources. Thanks to my advisors for guidance and feedback on this research. This work was supported by NSF grant [to be added] and [other funding sources].

References

[1] Mark Chen, Jerry Tworek, et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv:2107.03374*
[2] Raymond Li, Loubna Ben Allal, et al. 2023. StarCoder: may the source be with you! *arXiv:2305.06161*
[3] Baptiste Rozière, Jonas Gehring, et al. 2023. Code Llama: Open Foundation Models for Code. *arXiv:2308.12950*
[4] Ziyang Luo, Can Xu, et al. 2023. WizardCoder: Empowering Code Large Language Models with Evol-Instruct. *arXiv:2306.08568*
[5] Xinyun Chen, Maxwell Lin, et al. 2023. Teaching Large Language Models to Self-Debug. *arXiv:2304.05128*
[6] Sirui Hong, Xiawu Zheng, et al. 2023. MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework. *arXiv:2308.00352*
[7] Chen Qian, Xin Cong, et al. 2023. Communicative Agents for Software Development. *arXiv:2307.07924*
[8] Dong Huang, Qingwen Bu, et al. 2024. AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation. *arXiv:2312.13010*
[9] Md Ashraful Islam, Mohammed Eunus Ali, et al. 2024. MapCoder: Multi-Agent Code Generation for Competitive Problem Solving. *arXiv:2405.11403*
[10] John Schulman, Filip Wolski, et al. 2017. Proximal Policy Optimization Algorithms. *arXiv:1707.06347*
[11] Mark Chen et al. 2021. Evaluating Large Language Models Trained on Code. *arXiv:2107.03374*
[12] Jacob Austin, Augustus Odena, et al. 2021. Program Synthesis with Large Language Models. *arXiv:2108.07732*
[13] Dan Hendrycks, Steven Basart, et al. 2021. Measuring Coding Challenge Competence With APPS. *NeurIPS Datasets and Benchmarks 2021*
[14] Soneya Binta Hossain, Nadia Nahar, et al. 2023. Automated Test Generation using LLMs. *arXiv:2310.13294*
[15] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. *ESEC/FSE 2011*

A Detailed Prompt Examples

A.1 Architect Agent Full Prompt

```
You are a software architect. Given a problem description,
design a well-structured solution.

Analyze:
1. What are the main components needed?
2. How should they interact?
3. What data structures are appropriate?
4. What are potential edge cases or challenges?

For simple problems, a brief design is fine. For complex
problems, provide detailed module specifications.

Output your design as structured text with clear sections:
- Overview
- Components
- Data Flow
- Key Decisions

Keep it concise but complete.
```

A.2 Test Engineer Agent Full Prompt

```
You are a test engineer. Create comprehensive tests for the
provided code.

Your test suite should include:
1. Normal cases: typical expected usage
2. Edge cases: empty inputs, single elements, maximum values
3. Error cases: invalid inputs, null values
4. Boundary conditions: min/max values, off-by-one

Write clear test names that describe what's being tested.
Include assertion messages explaining what should be true.

Aim for at least 90% code coverage. Think adversarially -
how might this code break?

Use Python pytest format with descriptive test names.
```

B Reward Function Implementation

B.1 Correctness Calculation

Correctness is simply the fraction of tests passed:

$$R_{\text{correct}} = \frac{n_{\text{passed}}}{n_{\text{total}}}$$

We run tests in an isolated Docker container with 10-second timeout and 512MB memory limit.

B.2 Quality Metrics

We use Pylint scoring (0-10 scale) directly. For cyclomatic complexity, we average over all functions and normalize:

$$R_{\text{complexity}} = \max(0, 1 - \frac{\text{avg\_complexity}}{10})$$

Simple code (complexity 1-2) gets high reward. Very complex code (10+) gets zero.

B.3 Test Coverage

We use Coverage.py to measure statement and branch coverage:

$$R_{\text{coverage}} = 0.6 \cdot \text{stmt\_cov} + 0.4 \cdot \text{branch\_cov}$$

Both are in [0,1]. We weight statement coverage slightly more because branch coverage can be undefined for simple code without conditionals.

C Extended Results

C.1 Per-Category APPS Results

Detailed breakdown of APPS performance:

Table 6. APPS results by problem category

Category	AgentCoder	CodeCrew RL
Algorithms	64.1%	72.3%
Data Structures	61.2%	68.7%
Mathematics	55.8%	64.5%
String Processing	69.4%	76.1%

Improvements are consistent across categories. Largest gains in mathematics problems which require careful reasoning about numerical properties.

Received 1 October 2024; revised 15 December 2024; accepted 20 January 2025