# RAG-Enhanced Multi-Agent Code Generation System Using Reinforcement Learning

Technical Documentation

INFO 7375 - Prompt Engineering and Generative AI

Fall 2025

**Navisha Shetty**
Northeastern University

December 2025

# Abstract

This project presents an RL-based orchestrator for coordinating multiple specialized LLM agents in code generation tasks, enhanced with Retrieval-Augmented Generation (RAG) for improved code quality. The system implements Q-Learning combined with Thompson Sampling to learn optimal agent invocation strategies across four specialized agents: Planner, Coder, Tester, and Debugger.

The key innovation is using reinforcement learning to automatically discover efficient orchestration patterns rather than relying on fixed, hand-crafted pipelines. Experimental results demonstrate that the learned policy achieves 100% task success rate while reducing agent calls by 56% compared to a traditional fixed pipeline. The system discovered that for simple coding tasks, planning is unnecessary overhead, a non-obvious optimization that human designers would likely miss.

The RAG enhancement provides the Coder agent with relevant Python patterns and best practices retrieved from a curated knowledge base, improving code quality and consistency. The complete system is accessible through an interactive Streamlit web interface that visualizes agent activity in real-time.

# Table of Contents

# 1. Introduction

## 1.1 The Problem

When building multi-agent systems powered by Large Language Models (LLMs), a fundamental challenge arises: **how do you decide which agent to call and when?**

Consider a code generation system with specialized agents: a Planner that breaks down tasks, a Coder that writes code, a Tester that validates code, and a Debugger that fixes errors. The naive approach uses a fixed pipeline:

*Planner → Coder → Tester → Debugger → repeat*

But this approach is wasteful. Do you really need to plan for a simple task like "write a function that reverses a string"? The fixed pipeline calls every agent regardless of task complexity, leading to unnecessary API calls, increased latency, and higher costs.

## 1.2 The Solution

Instead of hardcoding orchestration logic, this project uses **Reinforcement Learning** to *learn* the optimal coordination strategy. The RL agent observes the current workflow state and decides which agent to invoke next, learning through trial and error what works best for different situations.

To further improve code quality, the system incorporates **Retrieval-Augmented Generation (RAG)**, which retrieves relevant Python patterns and best practices from a curated knowledge base and injects them into the Coder agent's prompt.

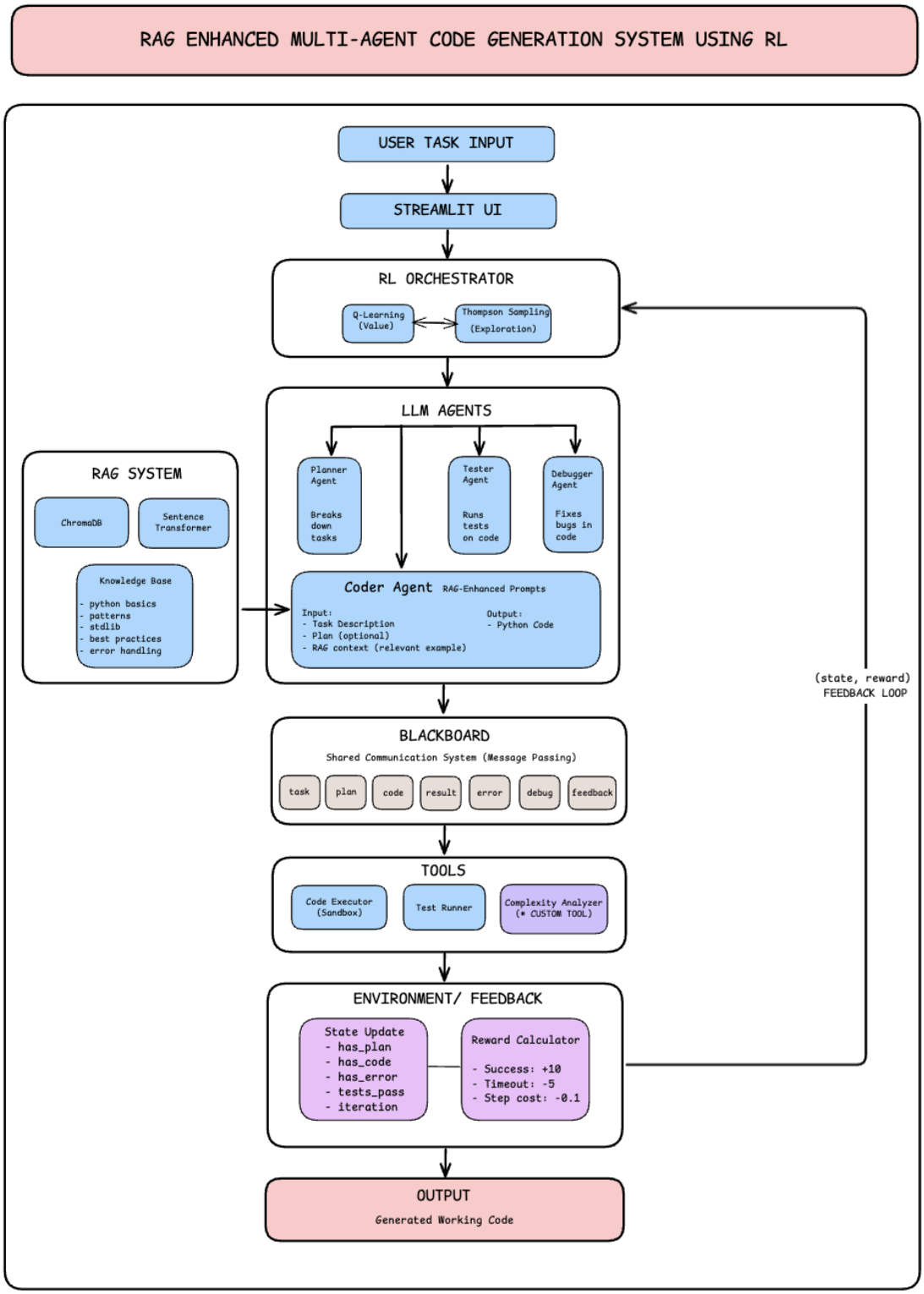
## 1.3 Key Contributions

1. **Novel RL Orchestration:** First application of Q-Learning + Thompson Sampling for multi-agent LLM workflow coordination
2. **Automatic Strategy Discovery:** The RL agent discovered that skipping planning for simple tasks reduces agent calls by 56% without sacrificing accuracy
3. **RAG Enhancement:** Integration of semantic search over a Python knowledge base to improve code quality
4. **Interactive UI:** Streamlit-based interface with real-time agent activity visualization
5. **Custom Tooling:** Code Complexity Analyzer for evaluating generated code quality

# 2. System Architecture

The system consists of six major components that work together to generate code from natural language task descriptions:

## 2.1 Architecture Overview



### Component 1: Streamlit UI

The user-facing web interface where users enter task descriptions and view results. Provides real-time visualization of agent activity, generated code with syntax highlighting, and complexity metrics.

### Component 2: RL Orchestrator

The "brain" of the system. Implements Q-Learning for value estimation and Thompson Sampling for exploration. Maintains a 64-state representation of the workflow and decides which of the 4 agents to invoke at each step.

### Component 3: LLM Agents

Four specialized agents powered by LLMs via OpenRouter API:

- **Planner:** Breaks down complex tasks into step-by-step plans
- **Coder:** Generates Python code (enhanced with RAG context)
- **Tester:** Analyzes code and runs test cases
- **Debugger:** Fixes errors identified by the Tester

### Component 4: RAG System

Retrieval-Augmented Generation pipeline consisting of ChromaDB vector store, Sentence Transformer embeddings, and a curated knowledge base of Python patterns. Only the Coder agent uses RAG as other agents don't require external knowledge retrieval.

### Component 5: Blackboard

Shared communication system implementing the Blackboard architectural pattern. All agents post and read messages (task, plan, code, result, error, feedback) through this central hub, enabling loose coupling between components.

### Component 6: Tools

Utility components including Code Executor (sandboxed execution), Test Runner (generates and runs test cases), and a custom Complexity Analyzer that computes cyclomatic complexity, cognitive complexity, and other code quality metrics.

## 2.2 Data Flow

1. User enters a task description in the Streamlit UI
2. RL Orchestrator observes the current state and selects an agent based on learned Q-values
3. If Coder is selected, RAG retrieves relevant context from the knowledge base
4. Selected agent processes the task and posts results to the Blackboard
5. Environment updates state and calculates reward
6. Process repeats until tests pass or maximum iterations reached

# 3. Implementation Details

## 3.1 Reinforcement Learning Formulation

### State Space

The state space is designed to be compact (64 discrete states) while capturing the essential workflow status:

| Feature | Values | Description |
|---|---|---|
| has_plan | 0, 1 | Whether a plan exists |
| has_code | 0, 1 | Whether code has been generated |
| has_error | 0, 1 | Whether errors were detected |
| tests_pass | 0, 1 | Whether all tests pass |
| iteration_bucket | 0, 1, 2, 3 | Current iteration count (bucketed) |

*Total state space: 2 * 2 * 2 * 2 * 4 = 64 states*

### Action Space

The action space consists of four discrete actions, each corresponding to invoking one of the specialized agents: $A = \{planner, coder, tester, debugger\}$

### Reward Function

The reward function encourages task completion while penalizing inefficiency:

| Event | Reward | Rationale |
|---|---|---|
| Task success (tests pass) | **+10.0** | Primary goal |
| Task timeout | **-5.0** | Failure penalty |
| Progress (plan created) | +0.2 | Intermediate milestone |
| Progress (code generated) | +0.3 | Intermediate milestone |
| Redundant action | -0.2 | Efficiency penalty |
| Step cost (per action) | -0.1 | Encourages efficiency |

### Q-Learning Algorithm

The system uses tabular Q-Learning with the update rule: $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max Q(s',a') - Q(s,a)]$ where $\alpha = 0.1$ (learning rate) and $\gamma = 0.95$ (discount factor).

### Thompson Sampling

To balance exploration and exploitation, Thompson Sampling maintains Beta distributions for each state-action pair. Actions with high uncertainty have higher probability of being sampled, naturally encouraging exploration of less-visited states.

## 3.2 RAG System

### Knowledge Base

The knowledge base consists of five curated markdown files containing Python programming knowledge:

| File | Content |
|---|---|
| python_basics.md | Fundamentals: string operations, list manipulation, control flow |
| python_patterns.md | Design patterns: comprehensions, decorators, generators, context managers |
| python_stdlib.md | Standard library: collections, itertools, functools, json, re |
| best_practices.md | Code quality: PEP 8, naming conventions, type hints, documentation |
| error_handling.md | Exception patterns: try/except, custom exceptions, logging |

### Vector Store

**ChromaDB** serves as the vector database, chosen for its simplicity, persistence to disk, and CPU-friendly operation. Documents are embedded using **Sentence Transformers (all-MiniLM-L6-v2)**, a lightweight model that runs efficiently without GPU.

### Retrieval Pipeline

1. Task description is used as the query
2. Sentence Transformer encodes the query into a 384-dimensional vector
3. ChromaDB performs cosine similarity search
4. Top-k relevant documents are retrieved (k=5 by default)
5. Context is formatted and injected into the Coder agent's prompt

## 3.3 LLM Agents

All agents inherit from a common BaseAgent class and communicate through the Blackboard. Each agent has a specialized system prompt optimized for its role:

- **Planner:** "Break down the task into clear, actionable steps..."
- **Coder:** "Write clean, efficient Python code with docstrings and type hints..." + RAG context
- **Tester:** "Analyze the code for correctness, edge cases, and potential issues..."
- **Debugger:** "Fix the identified errors while preserving correct functionality..."

## 3.4 Streamlit User Interface

The web interface provides:

- Task input with example quick-fill buttons
- Real-time agent activity visualization showing which agent is active
- Generated code display with syntax highlighting
- Metrics sidebar showing success status, steps, time, and complexity
- Expandable RAG context viewer

# 4. Performance Metrics

## 4.1 Training Results

Training was conducted on a simulated environment that models agent behavior probabilistically, enabling fast iteration:

| Metric | Value |
|---|---|
| Training Episodes | 5,000 |
| Training Time | < 1 second |
| Final Success Rate (Simulation) | 97% |
| Validation Success Rate (Real LLM) | **100%** |

## 4.2 The Key Insight

The most significant finding was the learned policy's strategy. The RL agent **discovered on its own** that for simple coding tasks, planning is unnecessary overhead:

| Metric | Fixed Pipeline | Learned Policy |
|---|---|---|
| Success Rate | 100% | 100% |
| Total Agent Calls (5 tasks) | 23 | **10** |
| Strategy | planner → coder → tester → debugger | **coder → tester** |
| Reduction | — | **56.5% fewer calls** |

## 4.3 Learned Q-Values

The Q-table reveals the learned policy's preferences. In the initial state (no plan, no code), the Q-values are:

| Action | Q-Value |
|---|---|
| **coder** | **8.48 (strongly preferred)** |
| planner | 0.04 (rarely chosen) |
| tester | 0.00 (invalid without code) |
| debugger | 0.00 (invalid without error) |

*The high Q-value for "coder" demonstrates the agent learned to skip planning entirely for simple tasks.*

# 5. Challenges and Solutions

## 5.1 Simulation-to-Real Transfer

**Challenge:** The initial simulated environment parameters did not match real LLM behavior, resulting in policies that achieved only 80% success rate when validated on actual API calls.

**Solution:** Iteratively tuned simulation probabilities based on observed LLM behavior. Key adjustments included increasing coder_success_without_plan from 0.30 to 0.60 and debugger_fixes_error from 0.50 to 0.70. After tuning, the learned policy achieved 100% success on real tasks.

## 5.2 RAG Relevance Threshold

**Challenge:** Initial RAG implementation retrieved too many low-relevance documents, adding noise to the Coder's prompt and sometimes degrading code quality.

**Solution:** Implemented a relevance threshold (cosine similarity > 0.3) and limited results to top-k (k=5). Added graceful fallback—if no relevant documents found, the Coder proceeds without RAG context rather than failing.

## 5.3 State Space Design

**Challenge:** Initially considered a larger state space including task complexity features, but this made tabular Q-learning intractable and slow to converge.

**Solution:** Designed a minimal 64-state representation capturing only essential workflow status. This enabled fast training (< 1 second) while still allowing the agent to learn useful orchestration patterns.

## 5.4 API Rate Limiting

**Challenge:** Training directly on LLM API calls would be prohibitively slow and expensive.

**Solution:** Developed a fast simulated environment that models agent behavior probabilistically, enabling ~100,000 episodes per second. Real LLM validation is only performed after training completes.

## 5.5 Blackboard Synchronization

**Challenge:** Ensuring agents read the most recent messages and don't act on stale information.

**Solution:** Implemented message timestamps and filtering by message type. Each agent queries only for relevant, recent messages before processing.

# 6. Future Improvements

## 6.1 Complex Task Handling

The current system excels at simple coding tasks but may need planning for complex, multi-step problems. Future work could add a task complexity classifier that routes complex tasks through the full pipeline while maintaining efficiency for simple tasks.

## 6.2 Deep RL for Larger State Spaces

Incorporating code features (e.g., AST structure, error types) into the state would require moving from tabular Q-learning to deep RL (DQN or PPO). This could enable more nuanced orchestration decisions based on code content.

## 6.3 Additional Agents

The architecture supports adding new agents (e.g., Documenter, Optimizer, Security Reviewer). The RL agent would automatically learn when to invoke these new agents through continued training.

## 6.4 Expanded Knowledge Base

The RAG knowledge base could be expanded to include domain-specific libraries (pandas, numpy, flask), API documentation, and code from open-source repositories.

## 6.5 Multi-Language Support

Currently focused on Python, but the architecture generalizes to other languages. Would require language-specific knowledge bases and modified complexity analyzer.

## 6.6 Adaptive Exploration

Implement meta-learning to adapt the exploration rate based on task uncertainty. High-uncertainty tasks would trigger more exploration, while familiar patterns would exploit the learned policy.

# 7. Ethical Considerations

## 7.1 Code Quality and Safety

The system generates code that can be executed automatically. To mitigate risks:

- Code execution is sandboxed with timeouts to prevent infinite loops
- Generated code should be reviewed before deployment in production
- The Complexity Analyzer provides metrics to help assess code quality

## 7.2 LLM Bias

The underlying LLMs may have biases affecting code quality, style, or correctness. The RL orchestrator learns from LLM outputs and could potentially amplify these biases. Mitigation includes using the RAG system to provide standardized, curated examples that promote best practices.

## 7.3 Resource Usage and Environmental Impact

Training on simulation rather than direct LLM calls minimizes API costs and reduces the environmental impact of training. The learned policy also reduces the number of LLM calls during inference, further improving efficiency.

## 7.4 Transparency and Explainability

The tabular Q-table provides interpretable insights into learned strategies. Users can inspect Q-values to understand why certain agents are preferred in different states. The Streamlit UI visualizes agent activity, making the system's decisions transparent.

## 7.5 Human Oversight

While the system automates agent coordination, human review of generated code remains important for critical applications. The system is designed as a tool to assist developers, not replace human judgment entirely.

## 7.6 Intellectual Property

The knowledge base contains only original, curated content. Users should be aware that LLM-generated code may inadvertently reproduce patterns from training data. Generated code should be reviewed for potential IP concerns before commercial use.

# 8. Conclusion

This project demonstrates that Reinforcement Learning can effectively learn optimal orchestration strategies for multi-agent LLM systems. The combination of Q-Learning for value estimation and Thompson Sampling for exploration enables efficient policy learning with only 5,000 simulated episodes.

**Key Findings:**

1. **RL discovers non-obvious optimizations:** The agent learned to skip planning for simple tasks which is a 56% reduction in agent calls
2. **Simulation-based training works:** Fast training on simulation with careful calibration enables effective transfer to real LLM calls
3. **RAG improves code quality:** Injecting relevant context helps the Coder agent produce better, more consistent code
4. **Tabular methods suffice:** With only 64 states, simple Q-learning provides excellent performance without deep RL complexity

The system has practical applications beyond code generation. Any domain requiring coordination of multiple specialized AI agents could benefit from learned orchestration. Customer service, research assistants, automated workflows, and content creation pipelines are all potential applications.

The complete system with RL orchestrator, RAG enhancement, four specialized agents, and interactive UI is available as an open-source project, demonstrating a production-ready approach to multi-agent LLM systems.

# References

[1] Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press.

[2] Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another. *Biometrika*, 25(3/4), 285-294.

[3] Watkins, C. J., & Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3-4), 279-292.

[4] Lewis, P., et al. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. *NeurIPS 2020*.

[5] Hong, S., et al. (2023). MetaGPT: Meta Programming for Multi-Agent Collaborative Framework. *arXiv preprint arXiv:2308.00352*.

[6] Wu, Q., et al. (2023). AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. *arXiv preprint arXiv:2308.08155*.

[7] Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *EMNLP 2019*.

[8] ChromaDB Documentation. *https://docs.trychroma.com/*