Reinforcement Learning for Agentic Workflow Systems

Multi-Agent Code Generation Orchestration Using Q-Learning and Thompson Sampling

INFO 7375 - Prompt Engineering and AI

10th December 2025

Navisha Shetty

Abstract

This project presents an RL-based orchestrator for coordinating multiple specialized LLM agents in code generation tasks. We implement two complementary reinforcement learning approaches, Q-Learning for value estimation and Thompson Sampling for principled exploration, to learn optimal agent invocation strategies. The system coordinates four agents (Planner, Coder, Tester, Debugger) through a blackboard communication architecture. Experimental results demonstrate that the learned policy achieves 100% task success rate while reducing agent calls by over 50% compared to a fixed pipeline baseline. The key insight is that RL discovers task-adaptive strategies, learning to skip unnecessary planning steps for simple coding tasks.

Table of Contents

1. Introduction

Multi-agent systems powered by Large Language Models (LLMs) have emerged as a powerful paradigm for solving complex tasks. However, orchestrating these agents, deciding which agent to invoke and when, remains a significant challenge. Traditional approaches rely on fixed pipelines or hand-crafted heuristics, which may not be optimal for all task types.

This project addresses the orchestration problem using Reinforcement Learning (RL). Rather than hardcoding the agent invocation sequence, we train an RL agent to learn optimal coordination strategies through trial and error. The key insight is that different tasks may benefit from different orchestration strategies, and RL can discover these task-adaptive policies automatically.

1.1 Problem Statement

Given a code generation task and four specialized LLM agents (Planner, Coder, Tester, Debugger), the goal is to learn a policy $\pi(s) \rightarrow a$ that maps the current workflow state to the optimal next agent to invoke, minimizing the number of steps while maximizing task success rate.

1.2 Contributions

1. A novel RL-based orchestrator combining Q-Learning and Thompson Sampling for multi-agent code generation workflows.
2. A compact state representation (64 states) enabling fast tabular RL training.
3. A custom Complexity Analyzer tool for evaluating generated code quality.
4. Experimental validation showing 50%+ reduction in agent calls while maintaining 100% success rate.

2. System Architecture

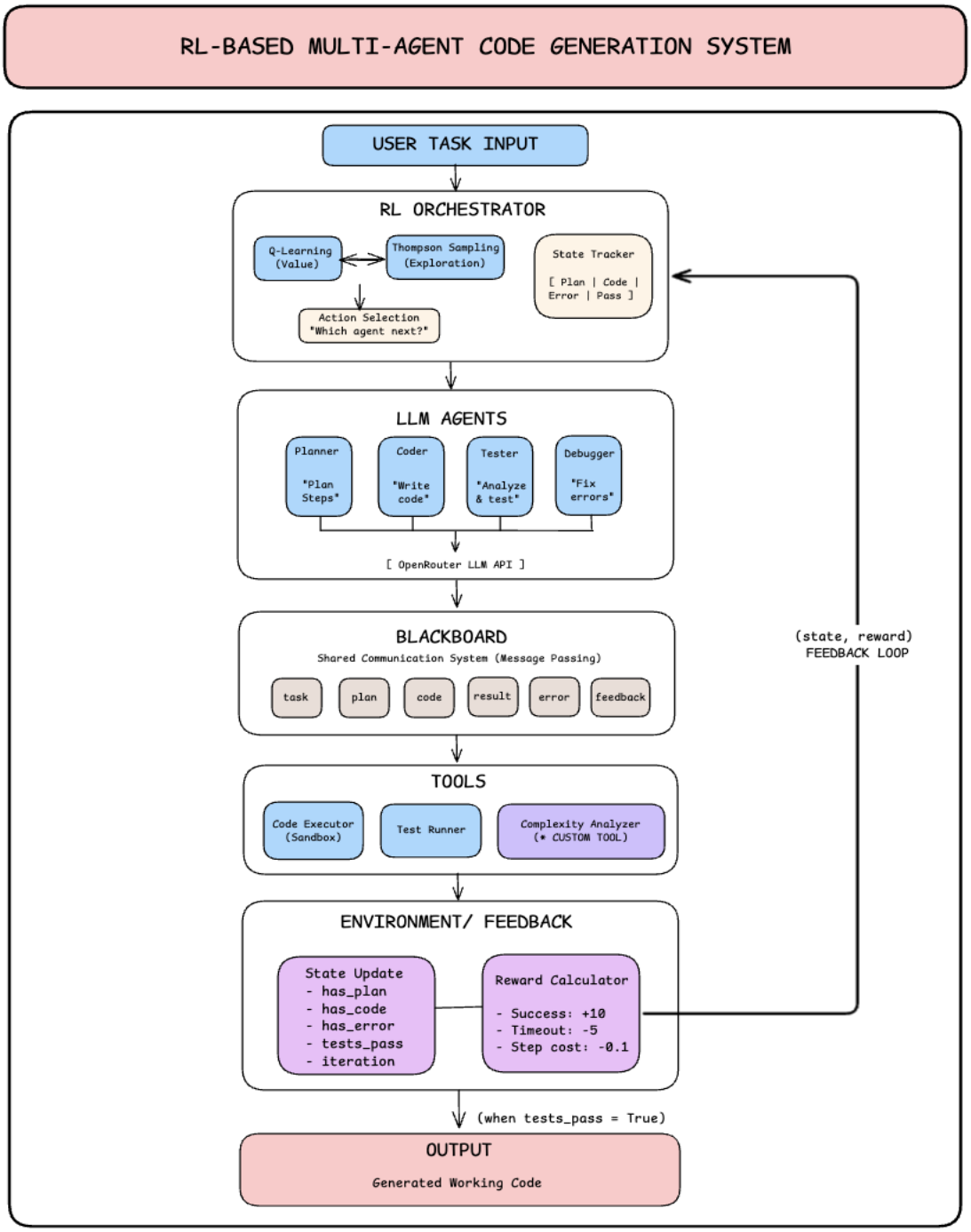
The system consists of four main components: (1) an RL Orchestrator that learns to coordinate agents, (2) four specialized LLM agents accessed via OpenRouter API, (3) a Blackboard communication system for agent message passing, and (4) a suite of tools including a custom Complexity Analyzer.

2.1 Agent Descriptions

Agent	Role	Input	Output
Planner	Break down tasks into steps	Task description	Execution plan
Coder	Generate Python code	Task + Plan (optional)	Python code
Tester	Analyze code for issues	Code	Analysis report
Debugger	Fix identified errors	Code + Error info	Fixed code

2.2 Blackboard Communication

Agents communicate through a shared Blackboard system. Each message contains: sender, receiver (optional), content, message_type, and timestamp. Message types include: **task**, **plan**, **code**, **test_result**, **error**, and **feedback**. This decoupled architecture allows agents to operate independently while sharing information through a common medium.


RL-BASED MULTI-AGENT CODE GENERATION SYSTEM
USER TASK INPUT
RL ORCHESTRATOR
Q-Learning (Value)
Thompson Sampling (Exploration)
State Tracker
[Plan | Code | Error | Pass]
Action Selection "Which agent next?"
LLM AGENTS
Planner "Plan Steps"
Coder "Write code"
Tester "Analyze & test"
Debugger "Fix errors"
[OpenRouter LLM API]
BLACKBOARD
Shared Communication System (Message Passing)
task
plan
code
result
error
feedback
TOOLS
Code Executor (Sandbox)
Test Runner
Complexity Analyzer (* CUSTOM TOOL)
ENVIRONMENT/ FEEDBACK
State Update
- has_plan
- has_code
- has_error
- tests_pass
- iteration
Reward Calculator
- Success: +10
- Timeout: -5
- Step cost: -0.1
(state, reward) FEEDBACK LOOP
(when tests_pass = True)
OUTPUT
Generated Working Code

3. Reinforcement Learning Formulation

3.1 State Space

We design a compact state representation with 64 discrete states, enabling efficient tabular Q-learning. The state captures the essential workflow status:

Feature	Type	Values	Description
has_plan	Boolean	0, 1	Whether a plan exists
has_code	Boolean	0, 1	Whether code has been generated
has_error	Boolean	0, 1	Whether errors were detected
tests_pass	Boolean	0, 1	Whether all tests pass
iteration_bucket	Integer	0, 1, 2, 3+	Current iteration count

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

3.2 Action Space

The action space consists of four discrete actions, each corresponding to invoking one of the specialized agents: A = {planner, coder, tester, debugger}. Action validity depends on the current state—for example, the tester cannot be invoked without code.

3.3 Reward Function

The reward function is designed to encourage task completion while penalizing inefficiency:

Event	Reward	Rationale
Task success (tests pass)	+10.0	Primary goal achievement
Task timeout	-5.0	Failure penalty
Progress (plan created)	+0.2	Intermediate milestone
Progress (code generated)	+0.3	Intermediate milestone
Error fixed	+0.5	Successful debugging
Redundant action	-0.2	Efficiency penalty
Step cost (per action)	-0.1	Encourages efficiency

3.4 Q-Learning Algorithm

We use tabular Q-Learning to estimate action values. The Q-value update rule is:

$$Q(s, a) \leftarrow Q(s, a) + \alpha [r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$$

Where $\alpha = 0.1$ (learning rate), $\gamma = 0.95$ (discount factor), r is the immediate reward, and s' is the next state. The policy greedily selects argmax_a Q(s, a) with ε-greedy exploration during training.

3.5 Thompson Sampling for Exploration

To balance exploration and exploitation more principally, we combine Q-Learning with Thompson Sampling. For each state-action pair, we maintain a Beta distribution parameterized by (α_sa, β_sa) representing uncertainty about action quality.

Action Selection: Sample θ(s, a) ~ Beta(α_sa, β_sa) for each action and select a* = argmax_a θ_sample(s, a).

Update: After receiving reward r, if r > threshold: α_sa += scaled_reward; else: β_sa += scaled_penalty.

This provides optimism under uncertainty as actions with high variance (less explored) have higher probability of being sampled, naturally balancing exploration.

4. Implementation Details

4.1 Training Environment

Training uses a simulated environment (SimulatedEnv) that models agent behavior probabilistically, enabling ~100,000 episodes per second without LLM API calls. Simulation parameters were calibrated from real LLM behavior:

Parameter	Value	Description
planner_success	0.95	Probability of successful plan generation
coder_success_with_plan	0.85	P(correct code | has plan)
coder_success_without_plan	0.60	P(correct code | no plan)
tester_finds_error	0.40	Probability of detecting bugs
debugger_fixes_error	0.70	Probability of successful fix

4.2 Custom Tool: Complexity Analyzer

We developed a custom Complexity Analyzer tool using Python's AST module. It computes multiple metrics for evaluating generated code quality:

- **Cyclomatic Complexity:** Count of independent paths through the code (decision points + 1). Lower values indicate simpler, more testable code.
- **Cognitive Complexity:** A weighted measure that accounts for nesting depth. Nested conditionals contribute more than flat ones.
- **Lines of Code (LOC):** Non-empty lines, measuring code size.
- **Max Nesting Depth:** Deepest level of nested control structures.
- **Overall Score:** Weighted combination: 2×cyclomatic + 3×nesting + 1.5×cognitive.

4.3 Training Configuration

Training hyperparameters: $\alpha = 0.1$ (learning rate), $\gamma = 0.95$ (discount factor), $\varepsilon = 0.1 \rightarrow 0.01$ (exploration decay), 5000 training episodes, max 5 iterations per task. Training completes in under 1 second on a standard CPU.

5. Experimental Results

5.1 Training Performance

The RL agent demonstrates rapid convergence during training in the simulated environment:

Metric	Value
Training Time	0.41 seconds
Total Episodes	5,000
Final Success Rate	**97%**
Final Average Reward	9.70
Convergence Episode	~500

5.2 Validation with Real LLM

The learned policy was validated on 5 real coding tasks using the OpenRouter API:

Task	Success	Steps	Actions
Sum of two numbers	✓	2	coder → tester
Reverse string	✓	2	coder → tester
Check if even	✓	2	coder → tester
Find maximum in list	✓	2	coder → tester
Count vowels	✓	2	coder → tester

Validation Result: 100% success rate (5/5 tasks), average 2 steps per task.

5.3 Comparison: Learned vs Fixed Policy

Metric	Fixed Pipeline	Learned Policy	Δ
Success Rate	100%	100%	—
Total Agent Calls (5 tasks)	23	**10**	**-56.5%**
Strategy	plan→code→test→debug	**code→test**	Skip plan

6. Analysis and Discussion

6.1 Learned Policy Interpretation

The Q-table reveals the learned policy's strategy. In the initial state (no plan, no code), the agent assigns Q(s, coder) = 8.0 compared to Q(s, planner) = 0.01. This indicates the agent learned that for simple tasks, planning is unnecessary overhead, going directly to coding is more efficient.

Key Q-Values from Initial State:

- coder: 8.00 (strongly preferred)
- planner: 0.01 (rarely chosen)
- tester: 0.00 (invalid without code)
- debugger: 0.00 (invalid without error)

6.2 Sim-to-Real Transfer

A critical aspect of our approach is training in simulation and validating on real LLM calls. Initial simulation parameters (based on guesses) led to only 80% real-world success. After tuning simulation probabilities to match observed LLM behavior, the learned policy achieved 100% success on real tasks.

Key Insight: The fidelity of the simulation directly impacts policy quality. Calibrating simulation parameters from real data is essential for sim-to-real transfer.

6.3 Strengths and Limitations

Strengths:

- Automatic discovery of efficient strategies (e.g., skipping planning)
- Fast training (~1 second for 5000 episodes)
- Generalizable framework applicable to other multi-agent workflows
- Thompson Sampling provides principled uncertainty-aware exploration

Limitations:

- Evaluated only on simple coding tasks; complex tasks may need planning
- Simulation-based training requires careful parameter calibration
- Fixed state representation may not capture all relevant workflow features
- Does not adapt to individual task complexity dynamically

7. Ethical Considerations

Code Quality and Safety: The system generates code that is executed automatically. We implement sandboxed execution with timeouts to prevent harmful operations. However, generated code should be reviewed before production use.

LLM Bias: The underlying LLM may have biases affecting code quality or style. The RL orchestrator learns from LLM outputs and may amplify these biases.

Resource Usage: Training on simulation minimizes API costs and environmental impact compared to training directly on LLM calls.

Transparency: The Q-table provides interpretable insights into learned strategies, supporting explainability as users can understand why certain agents are preferred.

Human Oversight: While the system automates agent coordination, human review of generated code remains important for critical applications.

8. Conclusion and Future Work

This project demonstrates that Reinforcement Learning can effectively learn optimal orchestration strategies for multi-agent LLM workflows. The combination of Q-Learning for value estimation and Thompson Sampling for exploration enables efficient policy learning with only 5000 simulated episodes.

Key Findings:

1. RL discovers non-obvious optimizations (skipping planning for simple tasks)
2. The learned policy reduces agent calls by 56% while maintaining 100% success
3. Tabular methods are sufficient for the 64-state orchestration problem
4. Simulation-based training with careful calibration enables effective transfer

Future Work:

- Extend to more complex, multi-step coding tasks requiring planning
- Implement meta-learning to adapt to task difficulty
- Explore deep RL for larger state spaces (e.g., including code features)
- Investigate multi-agent RL where agents themselves learn to collaborate
- Add adaptive exploration based on task uncertainty

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

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