A Specialized LLM Chain-of-Thought Framework

Our reward engineering pipeline employs four expert LLMs that work in a structured chain-of-thought workflow: Think LLM ($\mathcal{M}_{\text{think}}$), Code LLM ($\mathcal{M}_{\text{code}}$), Analysis LLM ($\mathcal{M}_{\text{analysis}}$), and Repair LLM ($\mathcal{M}_{\text{repair}}$). This architecture ensures systematic reward function generation by decomposing the complex task into distinct, focused stages with clear role boundaries and information flow constraints.

The chain-of-thought approach prevents common issues in single-model reward generation, such as inconsistent reasoning, code-logic misalignment, and insufficient error handling. Each LLM operates within strict role constraints to maintain workflow integrity. $\mathcal{M}_{\text{think}}$ focuses purely on conceptual analysis without code implementation, $\mathcal{M}_{\text{code}}$ translates structured analysis into executable functions, $\mathcal{M}_{\text{analysis}}$ leverages performance data for systematic improvements, and $\mathcal{M}_{\text{repair}}$ handles error correction while preserving original design intent.

This modular design enables targeted model selection for each subtask, allowing deployment of reasoning-optimized models for analysis, code-specialized models for implementation, and debugging-focused models for error correction. The structured information flow ensures that each stage builds upon validated outputs from previous stages, creating a robust and interpretable reward generation process.

A.1 Prompt for Think LLM

$\mathcal{M}_{\text{think}}$ decomposes natural language task descriptions into structured reward design logic, focusing on objective identification, observable analysis, and component boundaries without code generation.

Listing 1: Prompt for $\mathcal{M}_{\text{think}}$ (Think LLM)

```
You are an expert in reward function design. Your task is to analyze the given reinforcement
    learning task and provide structured reasoning for reward component design. DO NOT WRITE
    ANY CODE.
Follow these analysis steps strictly:

1. **Task & Environment Analysis**
   - What is the main goal of this task?
   - What are the key state variables available?
   - What are the physical and behavioral constraints?

2. **Objective Identification & Prioritization**
   - Identify the primary objective (most critical for task success)
   - List 2-3 secondary objectives that support the main goal
   - Explain the priority ranking and trade-offs between objectives

3. **Reward Component Design Logic**
   - For each objective, specify which observables should be used
   - Describe the desired behavior for each component (encourage/discourage)
   - Suggest normalization approaches (e.g., exponential, clipping, scaling)
   - Consider balance between exploration and exploitation

4. **Edge Case Analysis**
   - Identify potential boundary conditions or failure modes
   - Describe scenarios where the agent might exploit the reward
   - Suggest safeguards against unintended behaviors

**Input Context:**
Task Description: {TASK_DESC}
Environment Code: {ENV_CODE}
Available Observables: {OBSERVABLES}

**Output Format:**
Provide detailed reasoning for each section above. Focus on the logic and rationale behind
    each design decision. This analysis will be used by a code implementation specialist to
    create the actual reward function.
```

A.2 Prompt for Code LLM

$\mathcal{M}_{\text{code}}$ translates structured analysis into executable reward functions, ensuring code validity and environment compatibility.

Listing 2: Prompt for $\mathcal{M}_{\text{code}}$ (Code LLM)

```
You are a reward function implementation specialist. Based on the provided analysis, implement
    a working compute_reward function. Output ONLY the Python code wrapped in ```python ```
    blocks.

**Implementation Requirements:**
1. Function signature: def compute_reward(self, pos, action, state, terminated):
2. Return: (total_reward: float, components: Dict[str, float])
3. Initialize custom attributes: if not hasattr(self, 'attr'): self.attr = init_value
4. Define constants at function start using uppercase names
5. Add null checks for inputs to prevent runtime errors
6. Normalize components using np.exp, np.tanh, or np.clip
7. Keep individual component values in reasonable ranges [-5, 5]

**Input Analysis from M_think:**
{ANALYSIS_CONTENT}

**Environment Interface:**
- pos: Agent position (e.g., pos[0] = x-coordinate)
- action: Action vector taken by the agent
- state: Full observation vector from environment
- terminated: Boolean indicating episode termination

**Code Constraints:**
- Use only provided variables (pos, action, state, terminated) and valid self attributes
- Implement each reward component identified in the analysis
- Apply appropriate normalization for each component
- Ensure the function handles edge cases gracefully

Output only the complete compute_reward function implementation.
```

A.3 Prompt for Analysis LLM

$\mathcal{M}_{\text{analysis}}$ improves reward functions using training performance data, identifying ineffective components and unintended behaviors.

Listing 3: Prompt for $\mathcal{M}_{\text{analysis}}$ (Analysis LLM)

```
You are a performance analysis specialist for reward function optimization. Your task is to
    enhance the existing reward function based on training results.

Structure your response in two parts:

**Part 1: Performance Analysis**
Analyze the training metrics and identify:
- Key performance trends (convergence, plateau, instability patterns)
- Ineffective reward components (low correlation with success metrics)
- Unintended agent behaviors (actions that game the reward system)
- Specific improvement opportunities based on the data

**Part 2: Improved Implementation**
Provide an updated compute_reward function that addresses the identified issues:
- Preserve effective components that correlate with task success
- Modify or remove problematic components
- Add new components only if clearly justified by performance gaps
- Maintain code structure and return format
```

```

**Input Performance Data:**
Training Metrics: {TRAINING_METRICS}
Agent Behavior Observations: {BEHAVIOR_NOTES}
Current Reward Function: {CURRENT_CODE}

**Optimization Guidelines:**
- Focus on fitness_score as the primary optimization target
- Balance component magnitudes to prevent any single component from dominating
- Address training instability through better normalization
- Ensure changes align with the original task objectives

Provide both the analysis reasoning and the improved code implementation.
```

A.4 Prompt for Repair LLM

$\mathcal{M}_{\text{repair}}$ fixes runtime errors in generated code while preserving the original reward logic and component structure.

Listing 4: Prompt for $\mathcal{M}_{\text{repair}}$ (Repair LLM)

```
You are a code debugging specialist for reward functions. Fix the runtime error in the
    provided code without changing the reward logic or adding new features.

**Error Information:**
Error Type: {ERROR_TYPE}
Error Message: {ERROR_MSG}
Error Context: {ERROR_CONTEXT}

**Faulty Code:**
{FAULTY_CODE}

**Fix Requirements:**
1. Preserve all original reward components and their intended behavior
2. Fix only the specific error causing the runtime failure
3. Maintain the function signature and return format
4. Use only variables available in the environment interface
5. Keep all constants and normalization approaches unchanged
6. Ensure the fixed code handles similar errors in the future

**Available Environment Variables:**
- pos: Position tuple (pos[0] = x, pos[1] = y)
- action: Action array from the agent
- state: Full state observation vector
- terminated: Episode termination flag
- self.* attributes: Only those defined in the environment class

**Common Fix Patterns:**
- Replace undefined variables with correct state indices
- Add missing hasattr checks for custom attributes
- Fix array indexing errors with proper bounds checking
- Correct mathematical operations that cause overflow/underflow

Output only the corrected compute_reward function wrapped in ```python ``` blocks.
```

B Dual Dynamic Optimization Mechanisms

This section presents the two adaptive mechanisms that enhance reward function generation through dynamic parameter adjustment and model selection strategies. The LLM selector (LLMSelector class) and temperature adjustment (EntropyDynamicTemperature class) constitute the dual dynamic adjustment mechanism, each implementing distinct optimization logic to achieve adaptive responses to task requirements and performance feedback.

B.1 Performance-Based Model Selection

The LLM selector dynamically allocates models for each role according to cumulative performance, task-specific requirements, and exploration needs. Specialized model pools are maintained for distinct stages, and multiple strategies are available to balance optimization and diversity.

B.1.1 Stage-Specific Model Pools

Different task stages employ dedicated pools of candidate models, though certain models may appear in multiple pools due to overlapping capabilities (e.g., DeepSeek-Coder for both reasoning and debugging). Table 1 summarizes the mapping.

Table 1: Stage-specific model pools for dynamic selection

Stage	**Candidate Models**
Environment Understanding	DeepSeek-R1, DeepSeek-Coder
Reward Design	Qwen2.5-Coder, CodeGemma
Performance Analysis	Gemma-3, Llama-3.1
Error Debugging	DeepSeek-Coder, Qwen2.5

B.1.2 Multi-Strategy Selection Logic

The system implements three complementary selection strategies:

- **Round-robin:** Ensures fair usage by cycling through models within the stage pool.
- **Stage-based:** Combines 40% random exploration with 60% preference-based selection from curated lists.
- **Performance-oriented (default):** Dynamically evaluates and scores models based on effectiveness and reliability.

Algorithm 1 outlines the decision process, where exploration probability adapts to usage history and enforced switching prevents over-dependence on a single model.

Algorithm 1 Dynamic Model Selection Algorithm

```
1: Input: Current stage $s$, iteration $i$, reward history $H$, strategy $\sigma$
2: Output: Selected model $m$
3: if $\sigma = \text{round\_robin}$ then
4:     $m \leftarrow \text{models}[s][i \bmod |\text{models}[s]|]$
5: else if $\sigma = \text{stage\_based}$ then
6:     if random() $< 0.4$ then
7:         $m \leftarrow \text{random\_choice}(\text{models}[s])$
8:     else
9:         $m \leftarrow \text{preferred\_models}[s][i \bmod |\text{preferred\_models}[s]|]$
10:    end if
11: else                                                    ▷ Performance-oriented
12:    for each model $m_j$ in models[$s$] do
13:        Calculate reward-based performance and stability
14:        Assign fused score to scores[$m_j$]
15:    end for
16:    $m \leftarrow \text{argmax}_{m_j}$ scores[$m_j$]   with exploration adjustment
17: end if
18: return $m$
```

This procedure integrates both exploitation and exploration. The exploration rate begins at 20% and can rise to 50% with repeated model usage, while enforced switching every two iterations further promotes diversity.

B.1.3 Performance Evaluation Framework

The performance-oriented strategy relies on a multi-component scoring system that balances peak reward, consistency, stability, and historical robustness. The fused score is expressed compactly as:

$$\text{FinalScore} = (1-\gamma) \cdot \underbrace{\Big(\underbrace{0.7 \cdot \text{MaxReward} + 0.3 \cdot \text{MeanReward}}_{\text{BasePerformance}} \times (0.8 + 0.2e^{-\text{StdDev}}) \Big)}_{\text{CurrentScore}} + \gamma \cdot \text{HistoricalScore}$$

where $\gamma = 0.5$ balances recent and historical performance. This formulation prioritizes maximum reward (70%) while smoothing results through mean reward (30%). The exponential stability factor penalizes high variance, and historical integration prevents overfitting to short-term fluctuations.

B.2 Entropy-Driven Temperature Adjustment

As shown in Algorithm 2, temperature adjustment modulates LLM sampling randomness based on reward component diversity, performance trends, and output confidence to balance exploration and exploitation throughout the optimization process.

B.2.1 Multi-Dimensional Adjustment Criteria

Temperature adjustment operates on three key indicators that capture different aspects of generation quality and optimization progress.

Entropy analysis measures reward distribution diversity through kernel density estimation. When entropy falls below 0.3, indicating concentrated distributions that may lead to agent stagnation, temperature increases by 5% to promote exploration. Conversely, when entropy exceeds 0.5, suggesting dispersed distributions that may cause training instability, temperature decreases by 5% to enhance consistency.

Algorithm 2 Dynamic Temperature Adjustment

```
1: **Input:** Current rewards $R$, historical rewards $H$, current temperature $T_{prev}$
2: **Output:** Updated temperature $T_{new}$
3: Calculate entropy: entropy $\leftarrow$ KDE_entropy($R$)
4: Calculate confidence: confidence $\leftarrow$ 1/coefficient_of_variation($R$)
5: temp_factor $\leftarrow$ 1.0
                                                          ▷ Entropy-based adjustment
6: if entropy $< 0.3$ then
7:     temp_factor $\leftarrow$ temp_factor $\times 1.1$          ▷ Increase exploration
8: else if entropy $> 0.5$ then
9:     temp_factor $\leftarrow$ temp_factor $\times 0.9$          ▷ Decrease exploration
10: end if
                                                          ▷ Confidence-based adjustment
11: if confidence $> 0.8$ then
12:     temp_factor $\leftarrow$ temp_factor $\times 0.95$        ▷ High confidence, stabilize
13: else if confidence $< 0.4$ then
14:     temp_factor $\leftarrow$ temp_factor $\times 1.05$        ▷ Low confidence, explore
15: end if
                                                          ▷ Performance change adjustment
16: current_best $\leftarrow$ max($R$)
17: historical_best $\leftarrow$ max($H$)
18: if current_best $< 0.8 \times$ historical_best then
19:     temp_factor $\leftarrow$ temp_factor $\times 2.0$         ▷ Performance drop, explore more
20: else if current_best $\geq$ historical_best then
21:     temp_factor $\leftarrow$ temp_factor $\times 0.5$         ▷ Performance improved, exploit
22: end if
23: $T_{target} \leftarrow T_{prev} \times$ temp_factor
24: $T_{new} \leftarrow 0.9 \times T_{prev} + 0.1 \times T_{target}$          ▷ Momentum smoothing
25: $T_{new} \leftarrow \text{clip}(T_{new}, 0.1, 1.0)$                      ▷ Bound temperature
26: return $T_{new}$
```

Confidence measurement evaluates output consistency based on the inverse coefficient of variation. High confidence situations ($CV^{-1} > 0.8$) trigger temperature reduction ($\times 0.95$) to stabilize successful patterns. Low confidence scenarios ($CV^{-1} < 0.4$) increase temperature ($\times 1.05$) to expand the search space and discover better solutions.

Performance tracking compares current results with historical best achievements, triggering significant exploration when performance declines, and favoring exploitation when performance improves.

B.2.2 Smoothing and Coordination Mechanisms

Temperature updates employ momentum smoothing to prevent rapid oscillations that could destabilize the generation process. The new temperature combines 90% of the previous temperature with 10% of the target adjustment, ensuring gradual transitions. Temperature values are constrained within [0.1, 1.0] to avoid extreme sampling behaviors that could compromise generation quality.

The dual dynamic mechanisms coordinate through shared performance metrics and exploration states. Model selection determines which model to use while temperature adjustment controls how the model outputs, creating synergistic adaptive responses. When model selection favors exploration of new models, temperature adjustment may simultaneously increase randomness to enhance discovery. Conversely, when focusing on high-performing models, temperature typically decreases to stabilize outputs and reinforce exploitation of proven strategies.

C Experimental Configuration Details

This section provides comprehensive implementation specifications for reproducing the experimental setup and ensuring consistent evaluation across different reinforcement learning environments.

C.1 Environment Setup and Dependencies

The experimental framework supports multiple RL environments through standardized interfaces and carefully configured dependencies. Table 2 summarizes the key environment specifications.

Table 2: Environment Configuration Specifications

Environment	**Action Space**	**Observation Space**
BipedalWalker-v3	Continuous (4D)	24D (joints, velocities, LiDAR)
CartPole-v1	Discrete (2D)	4D (cart pos/vel, pole angle/vel)
Ant-v5	Continuous (8D)	111D (joint pos/vel, torso state, contacts)
SpaceMining-custom	Continuous+Discrete (3D)	53D (agent state, asteroids, mothership)

BipedalWalker-v3 utilizes `gymnasium[box2d]` v0.29.1 with continuous 4-dimensional joint torque control and 24-dimensional observations including joint angles, velocities, and LiDAR sensor data. Episodes terminate after 1600 steps or when the agent falls, with rewards ranging from -100 to over 300 based on forward progress and stability maintenance.

CartPole-v1 employs `gymnasium[classic_control]` v1.3.0 featuring discrete 2-action control (left/right force) and 4-dimensional observations capturing cart position, velocity, pole angle, and angular velocity. Episodes terminate after 500 steps, when the pole angle exceeds 12 degrees, or when the cart position exceeds ± 2.4 units. The reward corresponds to the duration of pole balancing, with a maximum of 500.

Ant-v5 is implemented with `gymnasium[mujoco]` v5.3.0, featuring an 8-dimensional continuous action space controlling joint torques. Observations include a 111-dimensional state vector encompassing joint positions/velocities, torso orientation, and contact sensor readings. Episodes terminate if the torso falls or becomes unstable. The reward function combines forward velocity, survival bonus, and control cost, typically ranging from 0 to 6000.

SpaceMining-v1 is a custom-designed environment adhering to the `gymnasium` interface. The observation space is 53-dimensional, integrating agent state (position, velocity, energy, inventory), asteroid features (up to 15 visible asteroids with relative positions and resource amounts), and mothership position. The action space consists of two continuous thrust controls and a binary mining action. Episodes terminate after 2000 steps or on collision. The reward function provides positive feedback proportional to mined resources and penalties for collisions and energy depletion.

Evaluation consistency is maintained through deterministic seeding strategies. Training employs randomized seeds per iteration for robustness, while evaluation uses fixed seeds [5, 10, 15, 20, 25] across all experiments. The software stack requires Python 3.8+, PyTorch 1.12+, and CUDA 11.6+ for GPU acceleration, with CUDA-compatible hardware featuring at least 8GB VRAM recommended for optimal performance.

C.2 Training Pipeline Configuration

The reinforcement learning training pipeline employs Proximal Policy Optimization (PPO) from Stable-Baselines3 v2.0.0 as the base algorithm. Table 3 details the standardized hyperparameters used across all environments.

Table 3: PPO Training Configuration Parameters

Parameter	**Value**
Policy Architecture	MlpPolicy [64, 64]
Learning Rate	3e-4 (linear decay)
Batch Size	64 samples
Discount Factor (γ)	0.999
GAE Lambda (λ)	0.95
PPO Clip Range	0.2
Optimization Epochs	10 per update
Value Function Loss	MSE (coef: 0.5)
Entropy Coefficient	0.01
Training Duration	1M timesteps/iteration
Evaluation Frequency	Every 10K timesteps
Evaluation Episodes	10 per checkpoint

The policy architecture uses multilayer perceptron networks with two hidden layers of 64 units each. Learning rate starts at 3e-4 with linear decay scheduling throughout training. Each policy update processes batches of 64 samples with 10 optimization epochs, using a discount factor of 0.999 for long-term reward consideration and GAE lambda of 0.95 for advantage estimation.

Training progresses for 1,000,000 timesteps per iteration with evaluation checkpoints every 1,000 timesteps. Each evaluation uses 10 episodes to assess performance stability. Early stopping triggers when performance plateaus for 5 consecutive evaluations without improvement.

Resource management includes NVML-based GPU utilization monitoring with dynamic device selection based on memory availability and computational load. Memory management features automatic batch size adjustment according to available VRAM. The framework supports multi-environment parallel training with shared policy updates and distributed computing through Ray framework integration for multi-node scaling.

Data storage employs compressed trajectory formats for efficient analysis and replay capabilities. TensorBoard logging captures comprehensive training metrics, reward component tracking, and agent behavior statistics. Model checkpoints save at peak performance points and iteration boundaries, enabling detailed analysis of learning progression and reward function effectiveness throughout the optimization process.