A Algorithm

The reward engineering process with Chain-of-Thought (CoT) reasoning is detailed in Algorithm 1. This algorithm presents a systematic approach to iteratively improve reward functions through a combination of task-oriented initialization and reflection-guided refinement.

Algorithm 1 Reward Engineering with Chain-of-Thought

```
Initialize environment and best_reward = (code=”, fitness=−∞)
// Iteration 0: Task-oriented CoT
prompt = BuildInitialPrompt(task_description, env_code)
if has_initial_reward then
    prompt += InitialRewardSuggestion()
end if
reward_codes = GenerateWithCoT(prompt, n_samples)
best_reward = EvaluateRewards(reward_codes)
// Iterations 1+: Reflection-guided CoT
for iteration = 1 to max_iterations do
    components = AnalyzeComponents(best_reward)
    reflection = CreateReflection(components)
    prompt = BuildReflectivePrompt(reflection, components)
    // CoT-guided reward generation
    reward_codes = GenerateWithCoT(prompt, n_samples)
    for each code in reward_codes do
        fitness = TrainAndEvaluate(code)
        if fitness ¿ best_reward.fitness then
            best_reward = (code, fitness)
        end if
    end for
end for
return best_reward
// Chain of Thought generation process
analysis = ”1. Analysis of previous performance”
reasoning = ”2. Improvement strategy”
implementation = ”3. Reward function code”
return LLM.generate(analysis + reasoning + implementation)
```

B Experimental Details

B.1 Task Details

The BipedalWalker-v3 environment from Gymnasium presents a challenging control task where a bipedal robot must learn to walk efficiently across terrain. The agent receives observations including hull angle, angular velocity, horizontal speed, vertical speed, position of joints and joints angular speed, legs contact with ground, and 10 lidar rangefinder measurements. The action space consists of 4 continuous actions controlling the torques applied to the joints. The episode terminates if the robot reaches the goal, falls over, or exceeds the maximum number of steps.

B.2 Implementation Details

The implementation configuration details are summarized in Table 1, which shows the key parameters used across different components of the system.

Table 1: Implementation Configuration Parameters

Component	**Parameter**	**Value**
Main	Backend	ollama
	Model	llama3.1
	Temperature	0.7
	Iterations	8
	Samples	8
	Sleep Time	5 min
	Use Initial Reward	false
CoT	Enabled	true
	Performance Weight	0.7
	Thought Process Weight	0.3
	Max Tokens	2000
Environment	Name	bipedal_walker
	Class Name	BipedalWalker
	Max Episode Steps	1600
	Benchmark	BipedalWalker-v3
RL Algorithm	Type	PPO
	Policy	MlpPolicy
	Network Architecture	[64, 64]
	Learning Rate	0.0003
	Batch Size	64
	N Steps	2048
	N Epochs	10
	Gamma	0.999
	GAE Lambda	0.95
	Clip Range	0.2
	Max Grad Norm	0.5
Training	Total Timesteps	1,000,000
	Number of Envs	10
	Device	auto
	State Stack	1
	Eval Episodes	8
	Eval Seeds	[5, 10]
Logging	Log Directory	logs/bipedal_walker
	Log Level	INFO
	Save GIF	true

The configuration spans multiple components: the main system settings, Chain of Thought parameters, environment specifications, reinforcement learning algorithm parameters, and training settings. As shown in Table 1, we used the llama3.1 model with a temperature of 0.7 for generation, and implemented PPO as the base RL algorithm with carefully tuned hyperparameters. The CoT component was enabled with a balanced weighting between performance (0.7) and thought process evaluation (0.3).

C Prompt Details

The prompting system consists of three main stages for reward function generation. Each stage employs specific prompts to guide the generation process.

C.1 Stage 1: Base Prompts and Task Context

The first stage combines foundational system prompts with task-specific context. Table 2 and Table 3 present the initial system prompt and coding instructions that form the foundation of our approach. These base prompts establish the core requirements and constraints for reward function generation. Following the base prompts, we define the specific task requirements and environment context. For the BipedalWalker task, this includes the task description (Table 4), environment implementation details (Table 5), and an optional initial reward template (Table 6). These components provide crucial details for the reward function design and establish the baseline implementation context.

Table 2: Base System Prompts

Initial System Prompt

You are a reward engineer writing effective reward functions for reinforcement learning tasks.
Your goal is to create a reward function to help the agent learn the task described in text.
Use relevant environment variables as inputs. Example signature:
`def compute_reward(self, ...):`
`  return reward, {}`
Do not use type hints. Return type: float, Dict[str, float].

Table 3: Coding Instructions

Coding Instructions
The reward function output should include: 1. Total reward (float) 2. Dictionary of individual reward components Format as a Python code string. Tips: 1. Normalize rewards using transformations like np.exp 2. Ensure input types match expected types 3. Use only self attributes from environment class definition 4. Initialize self variables with null checks 5. No new input variables 6. Include proper code formatting 7. Pass self as first argument 8. Do not compute fitness_score components 9. Create only the reward function 10. Maintain smooth transitions

The implementation details of the environment are provided in Table 5, which shows the core step function of the BipedalWalker environment. This function handles the physics simulation and state updates, providing

Table 4: Task-Specific Components

Task Description (BipedalWalker)
The goal is to make a biped navigate a 2D environment:
- The biped starts standing at the left end of the terrain with the hull horizontal
- The biped has to reach the right end and avoid falling down
- Must keep moving forward, avoid getting stuck
- Optimize for speed and minimal episode steps
- Goal achievement prioritized over movement smoothness

the foundation for our reward engineering process.

Based on the environment implementation, we optionally develop an initial reward template as shown in Table 6. When used, this template serves as a starting point for our CoT-based reward function enhancement process, incorporating basic components such as progress tracking, energy efficiency, and stability measures.

The environment implementation details are shown in Table 5, which presents the core step function of the BipedalWalker environment.

Table 5: Step Function Implementation of BipedalWalker

Step Function Implementation

```
    def step(self, action: np.ndarray):
        for i in range(4):
            self.joints[i].motorSpeed = float(SPEED_HIP * np.sign(action[i]))
            self.joints[i].maxMotorTorque = float(
                MOTORS_TORQUE * np.clip(np.abs(action[i]), 0, 1)
            )

        self.world.Step(1.0 / FPS, 6 * 30, 2 * 30)

        pos = self.hull.position
        vel = self.hull.linearVelocity

        for i in range(10):
            self.lidar[i].fraction = 1.0
            self.lidar[i].p1 = pos
            self.lidar[i].p2 = (
                pos[0] + math.sin(1.5 * i / 10.0) * LIDAR_RANGE,
                pos[1] - math.cos(1.5 * i / 10.0) * LIDAR_RANGE,
            )
            self.world.RayCast(self.lidar[i], self.lidar[i].p1, self.lidar[i].p2)

        state = [
            self.hull.angle,
            2.0 * self.hull.angularVelocity / FPS,
            0.3 * vel.x * (VIEWPORT_W / SCALE) / FPS,
            0.3 * vel.y * (VIEWPORT_H / SCALE) / FPS,
        ]
        state += [l.fraction for l in self.lidar]

        terminated = self.game_over or pos[0] < 0 or \
                     pos[0] > (TERRAIN_LENGTH - TERRAIN_GRASS) * TERRAIN_STEP

        reward, individual_reward = self.compute_reward(pos, action, state, terminated)
        fitness_score = self.compute_fitness_score(pos, action, state, terminated)
        individual_reward.update({'fitness_score': fitness_score})

        return np.array(state, dtype=np.float32), reward, terminated, False,
            individual_reward
```

The reward function implementation generated by our approach is presented in Table 6, which can be used as a starting point for the reward function generation process.

Table 6: Reward Function Implementation for BipedalWalker

Reward Function Implementation

```python
def compute_reward(self, pos, action, state, terminated):
    # Distance-based reward: reduced scale and increased precision
    distance_reward = np.exp(pos.x / 2000.0) - 1

    # Stability reward:
    angle_penalty = np.tanh(np.abs(state[0]) / 0.01) * 0.05
    angular_velocity_penalty = np.tanh(np.abs(state[1]) / 0.01) * 0.05

    # Smoothness reward:
    action_penalty = np.tanh(-np.sum(np.abs(action)) / len(action) / 10.0) * -0.2

    # Obstacle avoidance penalty: reduced magnitude
    obstacle_penalty = -np.min([l.fraction for l in self.lidar]) * 1

    # Velocity penalty: increased magnitude
    velocity_penalty = np.tanh(np.abs(state[2]) / 0.5) * -1.5

    # Jump penalty: reduced magnitude
    jump_penalty = np.tanh(-state[3] / 2.0) * 0.5

    # Proximity to goal reward:
    proximity_to_goal_reward = -np.exp(-(pos.x - TERRAIN_LENGTH) ** 2 / 50.0) * 1.5

    # Total reward: adjusted weights and scales for improved balance
    reward = (
        distance_reward
        + 0.3 * angle_penalty
        + 0.4 * angular_velocity_penalty
        - 0.6 * action_penalty
        - 0.1 * obstacle_penalty
        - 0.8 * velocity_penalty
        - 0.25 * jump_penalty
        + 2.0 * proximity_to_goal_reward
    )

    # Termination conditions remain the same

    individual_reward = {
        "distance_reward": distance_reward,
        "angle_penalty": angle_penalty,
        "angular_velocity_penalty": angular_velocity_penalty,
        "action_penalty": action_penalty,
        "obstacle_penalty": obstacle_penalty,
        "velocity_penalty": velocity_penalty,
        "jump_penalty": jump_penalty,
        "proximity_to_goal_reward": proximity_to_goal_reward,
    }

    return reward, individual_reward
```

C.2 Stage 2: Initial CoT Generation (Iteration 0)

The second stage focuses on the initial Chain-of-Thought (CoT) generation process. Table 7 presents the detailed prompt template used in the first iteration (Iteration 0) of our reward engineering process. This systematic prompt guides the language model through a careful analysis of the task requirements and constraints before implementing the actual reward function.

Table 7: Initial CoT Generation Prompt (Iteration 0)

```
Before implementing the reward function, please follow these steps:

1. First, analyze the task description and environment code carefully:
   - What is the main goal?
   - What are the key state variables?
   - What are the constraints?

2. Identify the key objectives and constraints:
   - Primary objectives
   - Secondary objectives
   - Physical constraints
   - Action constraints

3. Break down how different actions should be rewarded:
   - What behaviors should be encouraged?
   - What behaviors should be discouraged?
   - How to balance exploration vs exploitation?

4. Consider potential edge cases:
   - Boundary conditions
   - Extreme scenarios
   - Potential exploitation of the reward

5. Design your reward components:
   - Base reward components
   - Shaping rewards
   - Normalization approach

Format your response as follows:
1. Start with "Reasoning:" followed by your detailed analysis of each step above
2. Then provide your code implementation in a Python code block (‘‘‘python ... ‘‘‘)
```

The initial Chain-of-Thought (CoT) generation process follows a structured approach to reward function design. Table 7 presents the detailed prompt template used in the first iteration (Iteration 0) of our reward engineering process. This systematic prompt guides the language model through a careful analysis of the task requirements and constraints before implementing the actual reward function.

Stage 3: Iterative Refinement (Iterations 1 to n)

The final stage implements an iterative refinement process using three key components shown in Tables 8, 9, and 10:

1. Performance Analysis Prompt (Table 8): Evaluates the previous iteration's performance through training metrics, fitness scores, and behavioral patterns. This prompt helps identify strengths and weaknesses

in the current reward function.

2. CoT Reflection Prompt (Table 9): Builds upon the performance analysis to guide structured thinking about potential improvements. This prompt maintains consistency with the initial CoT approach while incorporating learned insights.

3. Reflection Instructions Prompt (Table 10): Provides specific guidelines for analyzing reward components and their effectiveness. This prompt focuses on practical aspects such as scaling, component relevance, and fitness score optimization.

This iterative process continues until the desired performance is achieved or the maximum number of iterations is reached. Each iteration builds upon the insights gained from previous rounds, incorporating feedback and performance metrics to guide the optimization process.

Table 8: Performance Analysis Prompt

```
Previous Iteration Performance Analysis:

1. Training Metrics:
{training_metrics}

2. Current Best Reward Function:
{best_reward_code}

3. Performance Statistics:
- Maximum achieved fitness: {max_fitness}
- Average episode length: {avg_episode_length}
- Learning stability: {learning_stability}
- Success rate: {success_rate}

4. Key Observations:
- Learning curve characteristics
- Agent behavior patterns
- Critical events or milestones
- Potential issues or limitations

Please analyze this information using the Chain of Thought process to design
an improved reward function.
```

This iterative process continues until the desired performance is achieved or the maximum number of iterations is reached. Each iteration builds upon the insights gained from previous rounds, incorporating feedback and performance metrics to guide the optimization process.

D Examples of CoT-based Reward Generation and Reflection

This section presents concrete examples of how the Chain-of-Thought (CoT) approach is applied in practice for reward function engineering. We showcase two key aspects: the initial reward generation process and subsequent reflection-based refinement.

Table 9: CoT Reflection Prompt

```
Before implementing the reward function, please follow these steps:

1. First, analyze the task description and environment code carefully:
    - What is the main goal?
    - What are the key state variables?
    - What are the constraints?

2. Identify the key objectives and constraints:
    - Primary objectives
    - Secondary objectives
    - Physical constraints
    - Action constraints

3. Break down how different actions should be rewarded:
    - What behaviors should be encouraged?
    - What behaviors should be discouraged?
    - How to balance exploration vs exploitation?

4. Consider potential edge cases:
    - Boundary conditions
    - Extreme scenarios
    - Potential exploitation of the reward

5. Design your reward components:
    - Base reward components
    - Shaping rewards
    - Normalization approach

Format your response as follows:
1. Start with "Reasoning:" followed by your detailed analysis of each step above
2. Then provide your code implementation in a Python code block (‘‘‘python ... ‘‘‘)
```

Table 10: Reflection Instructions Prompt

```
Please analyze the policy feedback and provide an improved reward function
to better solve the task.
Tips for analyzing feedback:
1. If a reward component's values are nearly identical, or it increases when
it should decrease (minimize) and vice versa, consider the following options:
    a. Change its scale or temperature parameter.
    b. Re-write the component.
    c. Discard the component.
2. If a component's magnitude is significantly larger, re-scale it.
3. You want to maximize the fitness score as it is the ground truth evaluator.
4. You want to maximize positive reward components values during training and
minimize negative reward components values.
5. If the fitness score is not improving during training, try to change the reward
function.
Analyze each existing reward component first, then write the new reward function
code proposal.
```

The first example demonstrates how the CoT approach guides the initial generation of a reward function. It shows both the analytical reasoning process and the resulting implementation for iteration 0 of the reward engineering process.

The second example illustrates how the CoT approach facilitates reflection on reward function performance. It includes detailed performance metrics, analysis of reward components, and subsequent refinements based on observed behavior.

Table 11: Analysis from llm with CoT in Iteration 0

```
        Analysis:

Breaking down the task description and environment code reveals that the primary
objective is to make the biped navigate from the left end of the terrain to the
right end while avoiding obstacles, falling down, or getting stuck.

Key state variables include the biped's position (x, y), velocity (vx, vy), and
motor speeds for each joint. The constraints are:

* Physical: the biped must not fall down, get stuck, or exceed the terrain boundaries.
* Action: the agent can control the motor speeds of each joint.

Primary objectives:

* Reach the right end of the terrain
* Avoid falling down, getting stuck, or exceeding
the terrain boundaries

Secondary objectives:

* Move forward as much as possible
* Go as fast as possible (minimize episode steps)

Reward components:

1. **Reach reward**: encourage reaching the right end of the terrain.
2. **Avoidance reward**: penalize falling down, getting stuck, or exceeding
the terrain boundaries.
3. **Forward movement reward**: encourage moving forward.
4. **Speed reward**: incentivize fast movement.

Normalization approach:

* Use np.exp to normalize rewards and ensure they are always positive.
```

Table 12: Code from llm with CoT in Iteration 0

```
def compute_reward(self, pos, action, state, terminated):
    reach_reward = 0
    avoidance_reward = 0
    forward_movement_reward = 0
    speed_reward = 0

    # Reward reaching the right end of the terrain
    if pos[0] > (TERRAIN_LENGTH - TERRAIN_GRASS) * TERRAIN_STEP:
        reach_reward = 10.0

    # Penalize falling down, getting stuck, or exceeding the terrain boundaries
    if terminated:
        avoidance_reward = -5.0

    # Encourage moving forward
    forward_movement_reward = state[2]  # vx component of velocity

    # Incentivize fast movement
    speed_reward = np.exp(state[3])  # vy component of velocity
    # (positive reward for high speed)

    individual_reward = {
        'reach': reach_reward,
        'avoidance': avoidance_reward,
        'forward_movement': forward_movement_reward,
        'speed': speed_reward
    }

    reward = reach_reward + avoidance_reward + forward_movement_reward + speed_reward

    return reward, individual_reward
```

Table 13: Reflection of Components in Iteration 1

```
Performance Analysis:
   reach: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0].
    Max: 0.0 - Mean: 0.0 - Min: 0.0
   forward_movement: [38.14766542628862, 100.18565675155529, 82.92425245350518,
   225.32142357140503, 728.2149681850315, 61.379876609731, 69.62986257168465,
   97.01515441786847, 62.27421109898911, 886.024284332291].
   Max: 886.024284332291 - Mean: 235.11173554183497 - Min: 38.14766542628862
   speed: [nan, nan, 74.75, 345.75, 800.0, 44.9375, 50.125, 63.0625, 47.875, 477.75].
   Max: nan - Mean: nan - Min: nan
   avoidance: [0.0, -0.1875, -0.5, -0.3125, 0.0, -0.5, -0.5, -0.5, -0.5, -0.375].
   Max: 0.0 - Mean: -0.3375 - Min: -0.5
   feature: [52.5, 25.0, 17.5, 123.75, 727.5, 40.0, 40.0, 40.0, 71.25, 655.0].
   Max: 727.5 - Mean: 179.25 - Min: 17.5
   fitness_score: [-9.752663325726997, -44.71357389366377, -89.69247023539725,
   -45.75511213654681, 100.08154316943032, -94.92986408164555, -95.32390355962065,
   -91.24470863744043, -97.31017249609376, 77.49146458473439].
   Max: 100.08154316943032 - Mean: -39.114946061197045 - Min: -97.31017249609376
   reward: [nan, nan, 174.67425264045596, 694.5089239552617, 2255.714967980981,
   145.81737657636404, 159.2548627257347, 199.57765482366085, 180.89921092242002,
   2018.3992846086621]. Max: nan - Mean: nan - Min: nan
   episode_length: [1600.0, 1100.25, 149.5, 691.5, 1600.0, 89.875, 100.25, 126.125,
   95.75, 955.5].
    Max: 1600.0 - Mean: 650.875 - Min: 89.875

Best Performing Reward Function:
    # Generated code
    def compute_reward(self, pos, action, state, terminated):
        reach_reward = 10 if np.linalg.norm(pos) < 5 else 0
        forward_movement_reward = 3 * np.abs(state[2])
        speed_reward = 0.5 * (np.exp(state[3]) - 1) / (np.max(np.exp(state[3])) - 1)
        exploration_reward = 1 if state[4] > 20 else 0
        avoidance_reward = -0.5 if terminated else 0
        feature_reward = 10 if state[5] == 1 else 0
        point_reward = 5 if state[6] == 2 else 0
        individual_reward = {
            'reach': reach_reward,
            'forward_movement': forward_movement_reward,
            'speed': speed_reward,
            'exploration': exploration_reward,
            'avoidance': avoidance_reward,
            'feature': feature_reward,
            'point': point_reward
        }
        reward = reach_reward + forward_movement_reward + speed_reward + exploration_reward
        + avoidance_reward + feature_reward + point_reward
        return reward, individual_reward
```

Table 14: Analysis from llm with CoT in Iteration 1

```
Analysis:

**Performance Analysis**

The learning curve trend is positive, indicating that the agent is improving over time.
However, the average fitness score of -39.114946061197045 suggests that the agent is not
consistently achieving good results. The maximum fitness score of 100.08154316943032
indicates that there are instances where the agent performs well, but it’s unclear what
triggers these successful behaviors.

**Reward Function Analysis**

* ‘reach_reward‘: Encourages the agent to reach a certain position within a threshold
(5 units). This might be effective in guiding the agent towards the goal.
* ‘forward_movement_reward‘, ‘speed_reward‘, and ‘exploration_reward‘: These rewards
are designed to encourage the agent to move forward, increase its speed, or explore
the environment. However, these components might not provide enough motivation for
the agent to perform complex behaviors.
* ‘avoidance_reward‘: This reward discourages termination of the episode, but it’s
unclear how effective it is in preventing premature termination.

The reward scales seem to be relatively high, which could lead to unstable learning
or overfitting. The ‘point_reward‘ and ‘feature_reward‘ components have a binary value
(0 or 1), which might not provide enough nuance for the agent to learn complex behaviors.

**Behavioral Analysis**

Based on the provided information, it’s unclear what strategies the agent has learned.
However, the presence of multiple reward components suggests that the agent is learning
to navigate the environment and interact with different features and points.
The learning trend indicates that the agent is improving over time, but there are instances
where it performs poorly (average fitness score). This might suggest that the agent is not
consistently learning effective strategies or is prone to overfitting.

**Improvement Planning**

1. **Reduce reward scales**: Decrease the magnitude of the rewards to avoid unstable
learning and overfitting.
2. **Make reward components more nuanced**: Replace binary rewards with continuous
values to encourage more complex behaviors.
3. **Improve exploration**: Increase the ‘exploration_reward‘ or introduce new
components that promote exploration, such as a penalty for stagnation or a bonus
for discovering new features.
4. **Tune the avoidance reward**: Adjust the ‘avoidance_reward‘ to prevent premature
termination while still encouraging the agent to explore and interact with the environment.
```

Table 15: Code from llm with CoT in Iteration 1

```
def compute_reward(self, pos, action, state, terminated):
    reach_reward = 0.5 if np.linalg.norm(pos) < 2 else 0
    forward_movement_reward = 1 * np.abs(state[2])
    speed_reward = 0.25 * (np.exp(state[3]) - 1) / (np.max(np.exp(state[3])) - 1)

    exploration_reward = 0.5 if state[4] > 10 else 0
    avoidance_reward = -0.25 if terminated else 0

    feature_reward = 2 if state[5] == 1 else 0
    point_reward = 1 if state[6] == 2 else 0

    individual_reward = {
        'reach': reach_reward,
        'forward_movement': forward_movement_reward,
        'speed': speed_reward,
        'exploration': exploration_reward,
        'avoidance': avoidance_reward,
        'feature': feature_reward,
        'point': point_reward
    }

    reward = reach_reward + forward_movement_reward + speed_reward
    + exploration_reward + avoidance_reward + feature_reward + point_reward

    return reward, individual_reward
```