Master of Science Thesis in Electrical Engineering
Department of Electrical Engineering, Linköping University, 2025

Autonomous Forklift Path Following Using Large Language Model Agents and Reinforcement Learning

Sean Moran and Arvid Önsten

Master of Science Thesis in Electrical Engineering

Autonomous Forklift Path Following Using Large Language Model Agents and Reinforcement Learning

Sean Moran and Arvid Önsten

LiTH-ISY-EX--25/5759--SE

Supervisor: **Sebastian Karlsson**
ISY, Linköping University
Erik Sellén
Toyota Material Handling MS

Examiner: **Daniel Axehill**
ISY, Linköping University

Division of Automatic Control
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Abstract

This thesis explores the viability of utilising Large Language Models (LLMs) to iteratively tune automatic control parameters for an autonomous forklift truck performing a path-following task. Additionally, an established control policy learning method, Reinforcement Learning (RL), is implemented to perform the path-following task. The main objective is to evaluate the performance of each method, allowing for a comparison between them.

Two systems are implemented to investigate the objective: a multi-agent LLM parameter tuning framework and a deep Q-network RL agent. The LLM framework focuses on tuning three control structures: Linear Quadratic (LQ), State Feedback, and Pure Pursuit, while examining the impact of different in-context learning prompt types, including zero-shot, zero-shot chain-of-thought, few-shot, and few-shot chain-of-thought prompting. The RL solution is trained in simulation to develop a policy that achieves desirable path-following performance.

The results demonstrate the promising capability of the LLM-based framework to tune control parameters effectively using simulation feedback, achieving strong performance. Although the RL approach achieves good results when trained with a simplified system model, it struggles with policy transfer to the more realistic simulation environment. Both LLM and RL methods require significant time and effort to implement. The extensive in-context prompt design needed for consistent LLM tuning results is comparable in complexity to the reward function design and training required by the RL solution.

Acknowledgments

First, we would like to sincerely thank our two supervisors: Erik Sellén at Toyota Material Handling and Sebastian Karlsson at Linköping University for their dedication throughout the thesis project. Their invaluable insight, thoughtful discussions, and continuous guidance have greatly helped shape the direction of this work. We would also like to thank our examiner, Daniel Axehill, for his guidance and support, offering thoughtful discussion which helped steer the thesis in the right direction.

We are also deeply grateful to Johan Lindell at Toyota Material Handling for providing us the opportunity to explore the objectives of this thesis. Toyota has been incredibly supportive, generously sharing time and resources, making us feel welcome and part of the team. Additionally, we would like to thank Hans Birkedal for taking time out of his busy schedule to assist us in any possible way. Our gratitude extends to all colleagues at Toyota Material Handling who have created a positive and motivating environment in which we have thrived and enjoyed our time at the company.

A special appreciation goes to our fellow master's thesis colleagues, Hanna Häger and David Albrekt, with whom we have shared valuable time, wisdom, and enjoyable moments throughout this journey.

Finally, we extend our heartfelt thanks to all family and friends who have supported us, not only throughout this thesis but throughout our entire educational careers.

Linköping, June 2025
Sean Moran and Arvid Önsten

Contents

Notation

Control Parameters

Notation	Meaning
Q	State weight matrix for a linear quadratic controller
R	Control weight matrix for a linear quadratic controller
L	Feedback gain matrix for state feedback and linear quadratic controller
l_d	Look-ahead distance of a pure pursuit controller

Abbreviations

Abbrevation	Meaning
LQ	Linear Quadratic (controller)
SF	State Feedback (controller)
PP	Pure Pursuit (controller)
ZS	Zero-shot (prompt)
ZS CoT	Zero-shot chain-of-thought (prompt)
FS	Few-shot (prompt)
FS CoT	Few-shot chain-of-thought (prompt)
LLM	Large Language Model
RL	Reinforcement Learning
DQN	Deep Q Network

1 Introduction

The main topic of this thesis is the investigation of the potential applications of Large Language Models (LLMs) within the field of automatic control. The research was conducted at Toyota Material Handling Manufacturing Sweden in Mjölby, Sweden. This introductory chapter provides an overview of the thesis, including the study's background, its objectives, key research questions, some limitations, a division of labor, and an outline of the thesis structure.

1.1 Background

Automatic control as an area of development is not new. The need for controlling systems, such as vehicles and industrial systems, has been of interest for a long time [1]. As technology has continued to evolve, automatic control systems have advanced accordingly. Designing effective automatic control systems involves several aspects, one of the most time-consuming being the fine-tuning of control parameters. Once the controller is designed, the tuning of these parameters is essential to ensure that performance requirements are met. Typically, a control system engineer performs this tuning through iterative trial-and-error testing, adjusting parameters until all requirements are satisfied. Given the necessity for domain-specific expertise combined with performance feedback, human reasoning remains the most effective method for tuning these control parameters.

Designing and tuning controllers for systems with unknown dynamics can introduce uncertainties, which further complicates the parameter tuning process. This is due to the fact that designs based on initial system knowledge may not be sufficient to create a controller that performs as intended. One existing solution to this challenge is Reinforcement Learning (RL). RL is a machine learning

method which through trial and error attempts to find an optimal control solution. This approach enables the handling of incomplete knowledge about a system's dynamics when searching for the solution. Although RL mitigates the need for prior knowledge of system dynamics, designing an effective reward function remains a time-consuming task.

In recent years, the availability and performance of Large Language Models have significantly increased. As a result, new potential applications for LLMs are continuously being explored, ranging from summarizing large volumes of information to assisting software developers in generating and debugging code. With increased availability of Large Language Models comes an increased interest in prompt engineering to effectively utilize them. By providing examples (Few-shot prompting) or instructions how to approach a problem, how to divide the task and in what order (Chain-of Thought prompting) more human-like problem solving is possible [2]. As a result, tasks traditionally suited to human capabilities could be performed by LLMs.

1.2 Objective

The objective of this thesis is to investigate whether the increasing performance of Large Language Models can be leveraged to estimate and fine-tune control parameters for forklifts, while also evaluating the model's ability to consider the performance characteristics of specific vehicles during this process. To assess the LLM's capability in performing this task, an existing adaptive computer-based solution utilising Reinforcement Learning will be implemented, allowing for a comparison between the two computational approaches.

The overarching objective can be divided into the following three specific aims:

- Implement an LLM-based solution to generate and fine-tune control parameters based on path-following scenarios in a simulation environment. Explore how the choice of controller type affects the LLM's ability to fine-tune these parameters.
- Implement a RL algorithm to iteratively design an optimal control solution by optimising a reward function to evaluate the LLM's overall performance in fine-tuning control parameters.
- Compare the two control solutions by assessing total error, along with the time taken and number of iterations required to generate sufficient control parameters and a sufficient control policy, respectively.

1.3 Research Questions

The research questions to be answered in this thesis are:

- How well can Large Language Models tune control parameters to find an

optimal control solution compared to a control solution created by a Reinforcement Learning algorithm?

- How do different controller types impact a Large Language Model's ability to generate and fine-tune control parameters?
- Can a Large Language Model use performance results from simulated forklift trucks to fine-tune control parameters?
- What impact does an initial parameter value guess have on the Large Language Models ability to fine-tune control parameters?
- Is it possible to utilise Large Language Models to reduce the time required for parameter tuning?

1.4 Limitations

Certain limitations have been placed upon the thesis project. The first limitation relates to the simulation environment. Due to the time constraint within the project, it was not possible to develop a simulation environment specifically tailored to the thesis's needs. Therefore, a pre-existing simulation environment was utilised. As a result, certain aspects such as forklift type and simulation-to-reality connection are not explored.

Furthermore, the simulation environment operates in real time, which limits the training speed of the Reinforcement Learning agent. If the simulator had supported accelerated execution, the RL agent could have trained exclusively within that environment. Due to this constraint, parts of the training were conducted in a simplified mathematical environment, which may impact the final policy performance in the real-time simulator.

Moreover, the choice and availability of Large Language Model has impacted the thesis project. Running an LLM on a local machine places a limit on the maximum size of any LLM leading to smaller number of parameters and reduced training data size. Additionally, when utilising an Application Programming Interface (API) to access a larger model, the cost is a factor limiting use. Therefore, not all testing and evaluation was performed on the largest state-of-the-art models.

1.5 Division of Labour

The work in this master's thesis project was collaboratively undertaken by the two authors, Arvid and Sean. Each author primarily focused on a distinct subsystem: Sean concentrated on the implementation of the LLM-based parameter tuning system, while Arvid focused on developing the reinforcement learning solution. Tasks that involved both subsystems, as well as general responsibilities such as documentation, integration and planning of testing and evaluation were performed jointly by both authors.

1.6 Thesis Outline

The thesis begins by presenting a few related works in Chapter 2 followed by Chapter 3 which provides detailed insights into the theory necessary for addressing the questions presented in earlier sections. Chapter 4 elaborates on the structure and functionality of the Large Language Model solution. Subsequently, the Reinforcement Learning control solution is explained in Chapter 5. The methodology chapters are concluded with Chapter 6 describing the testing and evaluation of the two systems. The results are presented and discussed in Chapter 7. Finally, a conclusion in Chapter 8 and future work in Chapter 9.

At the end of the thesis, the references will be provided, along with the LLM prompts in Appendix A.

2
Related Works

The following chapter focuses on related articles and works investigating similar or relevant areas to the objective of this thesis report.

2.1 Classic Control Design

Controller design usually considers a specific application and is approached accordingly. Different control strategies vary in their complexity and effort required for implementation. The desired use case also impacts the choice of control strategy and how to tune performance. Regardless of the method, achieving the desired system behavior relies heavily on the tuning process, which is crucial for the performance of any controller.

Traditionally, tuning relies on human expertise to manually select suitable controller parameters that fulfill specific design criteria. Our work aims to make the control parameter tuning process more efficient by using LLM agents.

2.2 Human level reward design by using LLM

A work closely related to both our LLM-based system and reinforcement learning approach is the method proposed by Ma et al. [3], in which a coding LLM is utilized to design the Reinforcement Learning reward function. Designing an effective reward function remains one of the most challenging aspects of RL development, as it directly influences the agent's learning and performance. In their approach, a Large Language Model generates an initial reward function, which is subsequently evaluated through simulation. Based on the simulated performance, a reward reflection mechanism analyzes the reward function's effective-

ness and generates feedback, which is then used by the coding LLM to produce an updated reward function in an iterative manner.

This novel framework, named EUREKA, represents a universal reward design algorithm powered by coding Large Language Models. Notably, the approach achieves human-level reward design for a diverse range of robot tasks without requiring task-specific prompt engineering or human intervention. The concept of iterative refinement through reflection and feedback has inspired our own workflow for tuning control parameters using LLMs, as well as demonstrated the potential of integrating language models with reinforcement learning for automated and efficient system design.

2.3 Iterative Use of LLM Agents

The utilization of Large Language Models extends beyond simple text generation, encompassing various innovative applications such as the deployment of LLM-based agents. By assigning these agents specific personas and clearly defined roles, their capabilities can be enhanced to perform more targeted and human-like problem-solving tasks. This approach is especially relevant in the domain of control system design, where iterative refinement is a fundamental aspect of achieving optimal performance.

Control design typically involves a cyclical process of testing, evaluation, and modification to improve system behavior. Recent work has explored the use of multiple interacting agents to collaboratively address complex problems. In particular, Guo et al. [4] introduced a ControlAgent framework that capitalizes on iterative interactions guided by explicit numerical feedback, such as pole placement criteria, steady-state error, and settling time. This feedback-driven loop enables the ControlAgent to incrementally modify and enhance the controller's performance, effectively emulating the iterative design strategies employed by experienced control engineers.

Their implementation focuses primarily on proportional-integral-derivative (PID) controllers within linear time-invariant (LTI) systems. In contrast, this work extends the ControlAgent to additional control structures and a nonlinear vehicular system. The ControlAgent demonstrated the ability to perfectly solve simple control tasks involving first- and second-order systems. By integrating domain-specific knowledge and tools within the LLM framework, the approach outperforms both conventional LLM applications and traditional control system toolboxes. This fusion of language model capabilities with expert control methods highlights the potential of LLM agents as highly reliable and efficient tools for automated control system design.

2.4 Reasoning Through Chain-of-Thought Prompting

Recent research has demonstrated that chain-of-thought (CoT) prompts, designed to guide Large Language Models through a sequence of intermediate reasoning steps, can significantly enhance their ability to perform complex reasoning tasks [5]. This prompt design strategy effectively elicits more deliberate and structured reasoning from sufficiently large LLMs by explicitly instructing the model on the reasoning steps required to arrive at the final answer.

The effectiveness of CoT prompting has been evaluated across various domains, including arithmetic problem solving, commonsense reasoning, and symbolic reasoning [5]. Remarkably, even without any fine-tuning or additional training, pre-trained language models showed substantial improvement in arithmetic reasoning tasks when using simple CoT prompts. Furthermore, notable gains were also observed in commonsense and symbolic reasoning performance.

The findings by Wei et al. [5] highlight that standard prompting approaches only tap into a lower bound of the true capabilities of Large Language Models. By decomposing problems into smaller stepwise reasoning steps, CoT prompting unlocks more sophisticated and accurate responses from LLMs, underscoring its potential in improving LLMs ability to solve complex tasks.

2.5 Trackmania Reinforcement Learning

Trackmania is a popular racing game where the player controls a race car and drives it through various racing tracks. The car can turn left or right, accelerate, or brake. Recent projects have successfully applied reinforcement learning to control the car and train it on a specific track to complete it as quickly as possible. By using state information as input and reward the taken actions, these methods have managed to surpass the human world record on the track [6].

There are notable similarities between Trackmania and the task of guiding a forklift along a predefined path. Inspired by this approach, one goal of this thesis is to develop a reinforcement learning agent capable of accurately following a given track using similar techniques. To the best of the authors' knowledge, RL agents specific to forklift truck path following have not been researched previously, even if RL is a large research area for autonomous vehicles.

3
Theory

To answer the questions stated in the Introduction in Chapter 1, and evaluate the two systems against each other, a theoretical knowledge base is required. The following chapter provides the theory necessary for the implementation and evaluation of the two systems.

3.1 Path Following

The control problem is to guide a forklift to track a predefined path, see Figure 3.1. To do this, the main objective is to minimize the distance error and heading error relative to the reference path over time. Distance error is the distance between the point on the reference path, orthogonal to the current vehicle heading, and the current position of the vehicle. The heading error is the difference between the heading of a path segment and the heading of the vehicle.

3.1.1 Single-Track Model

To construct a path following controller, it is important to model the motion of the forklift. A four-wheeled forklift truck is nonholonomic, meaning it has motion constraints that must be considered. To model the lateral and longitudinal motion of the forklift truck, a single-track model is used [7]. The single-track model simplifies the behavior of a four-wheeled vehicle by approximating it as a two-wheeled bicycle. See Figure 3.2.

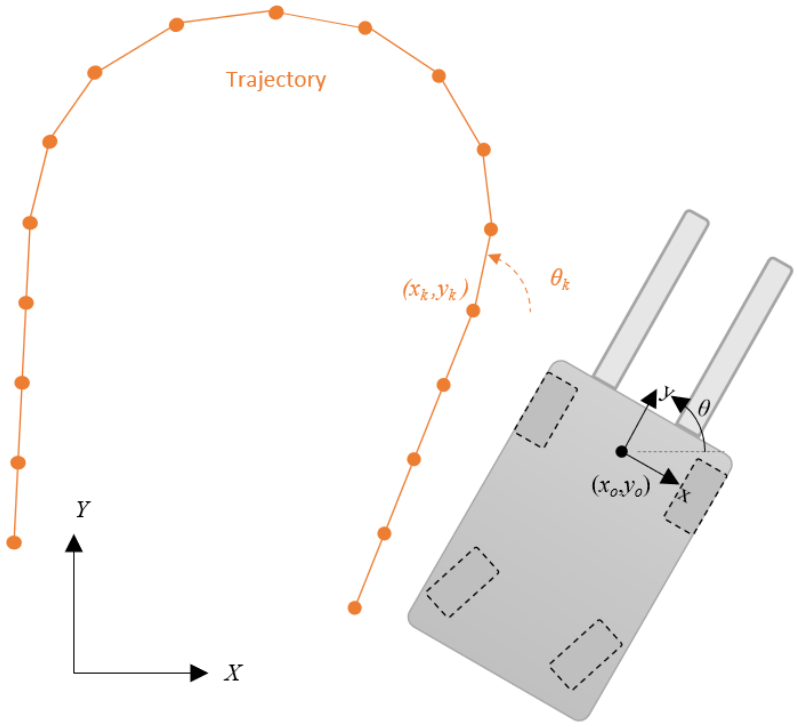


Figure 3.1: *Path-following visualisation example.*

This approximation simplifies the vehicles equations of motion to be:

$$\begin{aligned} \dot{x} &= v\cos(\theta) \\ \dot{y} &= v\sin(\theta) \\ \dot{\theta} &= \frac{v}{L_w}\tan(\delta) \end{aligned} \tag{3.1}$$

where v is velocity, θ is heading, and L_w is the vehicle's wheelbase. δ represents the steering angle [8].

The key assumptions for the model to be accurate are:

- Wheel slip is negligible, i.e., at low velocities.
- The two wheels on each axle are combined into a single wheel at the axle's center.
- The vehicle moves in a plane.

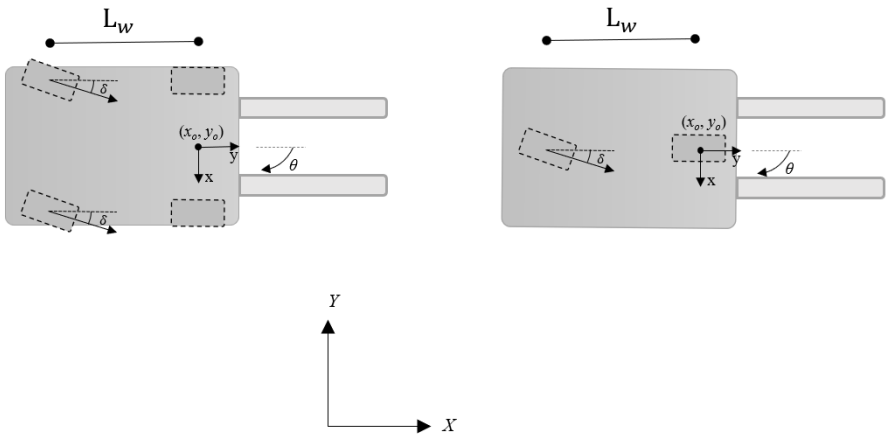


Figure 3.2: *Visualisation of single-track model of a forklift.*

3.1.2 Vehicle Local Error Equations

As written above, the main objective of the controller is to minimize the distance error and heading error to the path over time. If the dynamics of the errors are known, it is easier to minimize the errors. The errors are represented as:

- d_e - the distance to the closest path point.
- θ_e - the heading error compared to the closest path segment.

Where the dynamics of the vehicle local error equations are:

$$\begin{aligned} \dot{n} &= \frac{v}{1 - d_e c(n)} \cos(\theta_e) \\ \dot{d}_e &= v \sin(\theta_e) \\ \dot{\theta}_e &= \frac{v}{L_w} \tan(\delta) - \dot{n} c(n) \end{aligned} \tag{3.2}$$

where $n \epsilon [0, n_{end}]$ is the traveled distance along the path, $c(n)$ is the curvature of the path segment at path length n, and $p(n)$ is the global position of the path segment at path length n [7]. See Figure 3.3.

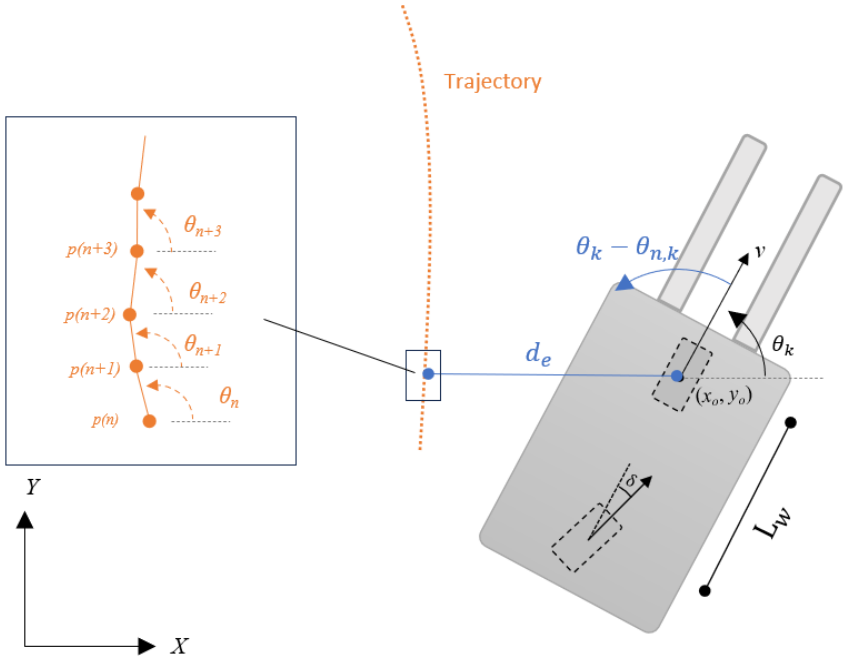


Figure 3.3: *Visualisation of path segments and errors.*

3.2 Controllers

There are several different approaches to path-following vehicle control. In the section below, the different control approaches conducted in this thesis are described.

3.2.1 Pure Pursuit Controller

The goal of the Pure Pursuit algorithm is to calculate a curvature from the given vehicle position to a desired point ahead on the reference path. An analogy which often is used is to compare the Pure Pursuit method with the way a human tends to drive, look at a point at some distance ahead and steer towards that point [9]. The distance to the point ahead is often called look-ahead distance (l_d) and by changing this, the properties of the controller change, tuning the controller behaviour.

Consider (x_l, y_l) to be the look-ahead point on the path in global coordinates, at a look-ahead distance from the position of the controlled vehicle. See Figure 3.4.

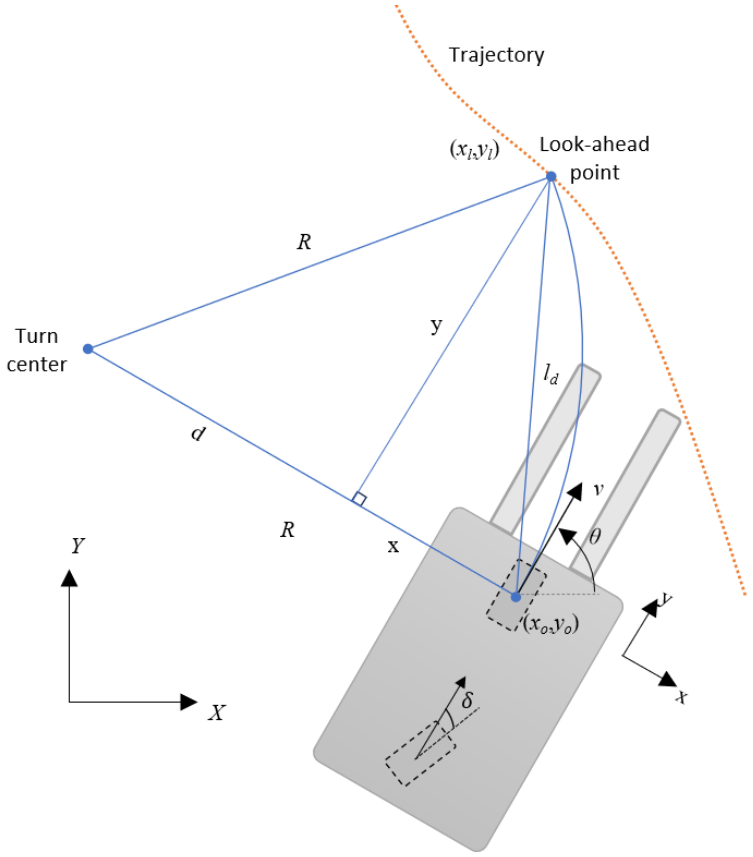


Figure 3.4: *Pure Pursuit geometry example.*

To get the x-coordinate of the look-ahead point in vehicle local coordinates, the vector $(x_l, y_l) - (x_0, y_0)$ is projected on a vector orthogonal to the vehicles heading of length 1.

Then the following equations hold.

$$x^2 + y^2 = l_d^2 \tag{3.3}$$

$$d^2 + y^2 = R^2 \tag{3.4}$$

$$R - d = x \tag{3.5}$$

Equation 3.3 describes the circle with the look-ahead as horizon. The goal point is on this circle. The next equation, Equation 3.4 describes the circle with radius R to both the vehicle and the goal point. The third equation, Equation 3.5 states the relationship between R and x, where x is the distance in the x direction to the goal point, described in vehicle coordinates.

With these equations, a steer angle δ can be derived.

$$(R - x)^2 + y^2 = R^2 \tag{3.6}$$

$$x^2 - 2Rx + R^2 + y^2 = R^2 \tag{3.7}$$

$$2Rx = l_d^2 \tag{3.8}$$

The curvature from the vehicle to the goal point on the path is

$$\frac{1}{R} = \frac{2x}{l_d^2} \tag{3.9}$$

The curvature of a single track model for a given steer angle is calculated as:

$$\text{curvature} = \frac{\tan \delta}{L_w} \tag{3.10}$$

The applied control action is calculated as:

$$\delta = \arctan\left(\frac{2L_w x}{l_d^2}\right) \tag{3.11}$$

3.2.2 Linear Quadratic Controller

The next control strategy is a Linear Quadratic Regulator (LQR). It is a method of computing an optimal feedback gain for a linear system represented in state space form. The goal is to find a feedback gain by minimizing a cost function which penalizes large system states and inputs. The cost function is given by

$$J = \sum_{t=0}^{\infty} \mathbf{x}_t^T \mathbf{Q} \mathbf{x}_t + u_t^T \mathbf{R} u_t \tag{3.12}$$

where $\mathbf{Q}$ is a diagonal matrix with values to determine how fast each state should converge to the reference value. $\mathbf{R}$ is a diagonal matrix with values to determine the penalty of a large control signal.

A linear state space system can be represented as:

$$\dot{\mathbf{x}}(t) = \mathbf{A}\mathbf{x}(t) + \mathbf{B}u(t) \tag{3.13}$$

$$\mathbf{y}(t) = \mathbf{C}\mathbf{x}(t) \tag{3.14}$$

As written above, the objective of the controller is to follow a path by minimizing the distance error and heading error. In Section 3.1.2, the dynamics of the error equations are determined. The idea is to use the errors as system states, then a nonlinear system

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} \dot{d}_e \\ \dot{\theta}_e \end{bmatrix} = f(\mathbf{x}(t), u(t)) \tag{3.15}$$

can be formed as Equation 3.2. The linear quadratic controller requires linear system dynamics, therefore the nonlinear system dynamics need to be linearized. By using the Jacobian [7] the error dynamics are given by:

$$\frac{\partial f}{\partial x} = v \begin{bmatrix} 0 & \cos(\theta_e) \\ \frac{-c(n)^2 \cos(\theta_e)}{(1-dc(n))^2} & \frac{\sin(\theta_e)}{1-dc(n)} \end{bmatrix} \tag{3.16}$$

$$\frac{\partial f}{\partial u} = v \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{3.17}$$

Linearizing with reference curvature as a straight path, $c(n) = 0$ and $\theta_e = 0, d_e = 0$, then the following linear dynamics are given in state space form:

$$\dot{\mathbf{x}}(t) = \begin{bmatrix} 0 & v \\ 0 & 0 \end{bmatrix} \mathbf{x}(t) + \begin{bmatrix} 0 \\ v \end{bmatrix} u(t). \tag{3.18}$$

The feedback control signal $u(t)$ minimizing the cost function is computed by

$$u(t) = -\mathbf{L}\mathbf{x}(t) + r(t) \tag{3.19}$$

where $r(t)$ is the curvature of the projected point on the reference path. The optimal feedback gain matrix $\mathbf{L}$ is computed as

$$\mathbf{L} = \mathbf{R}^{-1}\mathbf{B}^T\mathbf{P} \tag{3.20}$$

and $\mathbf{P}$ is the unique, positive semi-definite, symmetric solution to the algebraic Riccati equation

$$\mathbf{A}^T\mathbf{P} + \mathbf{P}\mathbf{A} + \mathbf{Q} - \mathbf{P}\mathbf{B}\mathbf{R}^{-1}\mathbf{B}^T\mathbf{P} = 0 \tag{3.21}$$

It follows that if the eigenvalues of the matrix $(\mathbf{A} - \mathbf{B}\mathbf{L})$ are placed in the left half-plane, the feedback is within the stability region [10].

When implementing an LQ controller on a computer, a time discrete implementation is required. This means that stable poles are shifted from the left half-plane to within the unit circle. The following equation is the discrete time feedback gain matrix

$$\mathbf{L} = (\mathbf{R} + \mathbf{B}^T\mathbf{P}\mathbf{B})^{-1}\mathbf{B}^T\mathbf{P}\mathbf{A} \tag{3.22}$$

and discrete time algebraic Riccati equation

$$\mathbf{P} = \mathbf{A}^T\mathbf{P}\mathbf{A} - \mathbf{A}^T\mathbf{P}\mathbf{B}(\mathbf{R} + \mathbf{B}^T\mathbf{P}\mathbf{B})^{-1}\mathbf{B}^T\mathbf{P}\mathbf{A} + \mathbf{Q} \tag{3.23}$$

3.2.3 State Feedback Controller

The next control strategy is a state feedback controller [1]. It is a feedback controller similar to the LQR controller above, but instead of computing the optimal

feedback gain **L** by minimizing a cost function, the gain is tuned by the user to match the desired behavior. By measuring the system states and feeding them back into the controller, a next control action can be determined to reach towards the goal states.

For a state feedback system with the feedback law

$$\mathbf{u}(t) = -\mathbf{L}\mathbf{x}(t) + r(t) \tag{3.24}$$

a state space system can be as:

$$\dot{\mathbf{x}}(t) = (\mathbf{A} - \mathbf{B}\mathbf{L})\mathbf{x}(t) + \mathbf{B}r(t) \tag{3.25}$$

$$\mathbf{y}(t) = \mathbf{C}\mathbf{x}(t). \tag{3.26}$$

where **L** is the feedback gain matrix and $r(t)$ is the reference path curvature.

The solution **x**(t) to the state space system is

$$\mathbf{x}(t) = e^{\mathbf{A}(t-t_0)}\mathbf{x}(t_0) + \int_0^t e^{\mathbf{A}(t-\tau)}\mathbf{B}u(\tau)\, d\tau \tag{3.27}$$

For the state feedback system the solution **x**(t) is given by

$$\mathbf{x}(t) = e^{(\mathbf{A}-\mathbf{B}\mathbf{L})(t-t_0)}\mathbf{x}(t_0) + \int_0^t e^{(\mathbf{A}-\mathbf{B}\mathbf{L})(t-\tau)}\mathbf{B}r(\tau)\, d\tau \tag{3.28}$$

The goal is to form a feedback gain **L** which stabilises the system, follows the path and gives a desirable driving behaviour. The characteristics of the controller, such as stability are determined by the eigenvalues of the matrix $(\mathbf{A} - \mathbf{B}\mathbf{L})$. The eigenvalues of the matrix represent the placement of system poles. Poles placed in the right half-plane result in an unstable controller.

3.3 Large Language Models

The following section will present an introduction to Large Language Models and the developmental methods required for the objectives of the thesis project.

LLMs are large deep learning models, a type of machine learning designed to understand and generate human language. At their core, LLMs utilize neural networks based on the Transformer architecture, consisting of an encoder and a decoder that work together to extract meaning from text and understand the relationships between words and sentences. These neural networks are immensely large, containing vast numbers of parameters that enable them to identify complex linguistic patterns, such as similar meanings and parts of speech, by considering multiple factors in context [11, 12].

As a significant breakthrough in natural language processing (NLP), a subfield

of artificial intelligence focused on enabling computers to recognize and understand human language, LLMs have advanced generative AI capabilities, which can be utilized in areas such as text summarization and code generation. NLP involves decomposing language into smaller units, such as words or subword tokens, a process known as tokenization. This decomposition allows models to analyze the structure and meaning at multiple levels, learning how individual words combine and interact to form coherent meaning. Advances in NLP research have enabled computers not only to recognize but also to generate natural language, improving the communication skills of LLMs and enabling more natural interactions [12, 13].

During training, LLMs are exposed to massive amounts of text data from a wide variety of domains. The models learn by predicting the next word in a sentence based on the context provided by preceding words, attributing probabilities to possible next words. This process allows the model to capture grammar, semantics, and conceptual relationships autonomously. The scale of their training and the immense size of their neural networks empower LLMs to grasp complex language patterns and produce coherent, contextually relevant natural language outputs, drawing upon their extensive learned knowledge [11, 12].

Numerous LLMs are available, each with different strengths and weaknesses in various aspects. OpenAI has its GPT-4.1 and GPT-4.1-mini models, where the latter is a smaller model containing fewer parameters. They also offer a model called o1, which functions as a reasoning model, thereby solving problems in a step-by-step manner [14]. Other available models include Meta's Llama 3.3 [15], which is comparable to GPT-4.1, and DeepSeek-V3 [16], also a reasoning model, making it comparable to the o1 model from OpenAI. Many of these models are available in different sizes or parameter counts, and the number of parameters can influence a model's capacity to handle complex problems and requests.

3.3.1 Agents

When an LLM is presented with a complex problem, the pre-trained model may not be capable of solving it. In this case, an agent can be helpful. In LLM programming, an agent can be viewed as a coordinator or a "brain". The agent is activated with a prompt specifying how to solve a task, detailing the order in which to address issues and identifying any available tools that may assist in task resolution. Additionally, it can keep track of previous answers or behaviors by using memory [17]. Given the LLM's comprehensive internal world knowledge, even a model that is not trained on domain-specific information can effectively tackle domain-specific tasks by employing agents with planning and tool-calling capabilities [18]. Figure 3.5 illustrates an example of an agent's structure.

Another method for instructing an agent on solving a task, while also improving its performance in a specific domain, is by assigning the agent a persona. For example, an agent could be designated the role of a philosopher when addressing a moral question, or as a software engineer when providing assistance with code generation [17, 19]. This persona is specified in the activation prompt.

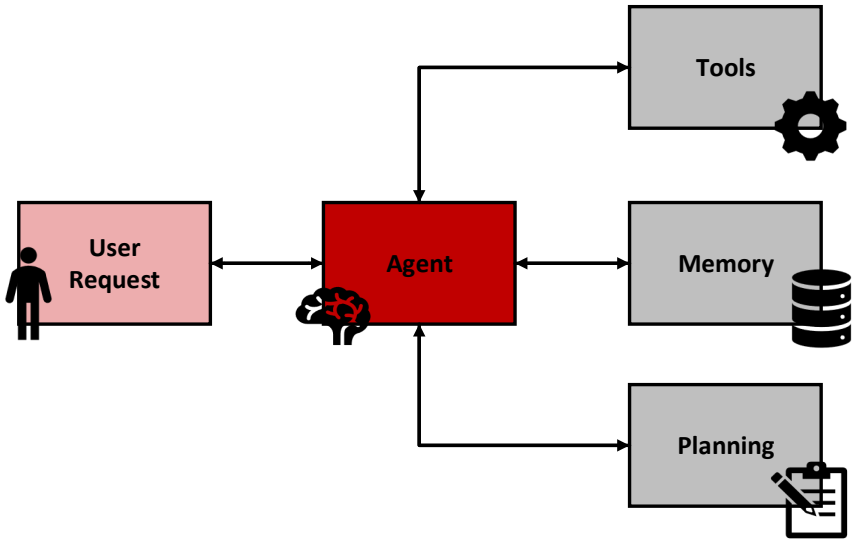


Figure 3.5: *LLM agent structure with tool calling, memory and planning.*

3.3.2 Multi-Agent Systems

Multi-agent programming with Large Language Models involves utilising multiple agents working together to solve a task, thereby allowing for increased specialisation among the agents. By creating a role-play scenario in which each agent assumes a specific role, realistic scenarios similar to human collaboration enable an LLM to address problems in an accurate and step-by-step manner. For example, a team of experts solving different aspects of a problem [19]. The use of multi-agent LLM solutions has become more prevalent, with one example highlighted by Xingang et al. (2024) [4]. In the article, a multi-agent structure is employed to tackle a control system problem. An initial agent takes user input and analyses the task before delegating to another agent, which is responsible for creating a solution to the control system problem. Based on the performance of the solution, feedback is provided to the second agent responsible for controller design, who updates the solution to enhance performance. By implementing a multi-agent structure, the authors demonstrate a method of solving a control system task in a manner similar to human collaboration, while leveraging LLMs [4].

3.4 Prompt Engineering

The Natural Language Processing capabilities of Large Language Models and the growing interest in their potential applications have made prompt engineering increasingly important, as the primary method of interaction with LLMs is through prompts. The design of the prompt determines the behaviour of the LLM, a process known as prompt programming [20]. Since the prompt acts as a guide to the LLM, the result is impacted by the prompt it receives. Consequently, better and more finely tuned prompts can produce superior results [21].

There are various aspects to consider when designing a prompt, including wording, context, and structure. According to the OpenAI guide on effectively prompting language models [22], a few best practices include:

- Writing clear instructions, including the complexity level of the answer and the desired output length, to improve performance. The model can only act upon the instructions provided to it.
- Splitting large tasks into smaller sub-tasks and creating a workflow where the solution from one task is used to address the next.
- Including examples of input, methods for solving the task, and expected output to promote desired behavior.
- Allowing the model time to think rather than expecting a quick answer, which it might assume to be correct.

In terms of providing the model with examples and allowing it time to think, three specific categories of prompts are of interest: zero-shot prompting, few-shot prompting, and chain-of-thought prompting. All three of these prompt types are a type of in-context learning. This means that by including additional information relating to the task in a prompt, and therefore including it within the context of the LLM, the LLM is able to temporarily learn from the information provided in the prompt. Due to this, a few-shot prompt containing a number of relevant examples can assist the LLM in solving a problem more efficiently or even solve a task perviously outside the domain specific expertise of the model [23].

Although these three categories are distinct prompt types, a combination may yield the best results [4].

3.4.1 Zero-Shot and Few-Shot Prompting

Zero-shot prompting refers to prompts written without any specific examples of how to approach the problem. For instance, asking the LLM to translate a certain word or sentence from one language to another [24]. By not providing any examples, the Large Language Model is free to approach the task in any way it sees fit. However, this freedom may result in the model not behaving in the desired manner. One way to improve zero-shot performance is by tuning the model based on relevant datasets. If tuning is not an option, providing examples

or demonstrations in the prompt, a technique known as few-shot prompting [25], can improve performance.

3.4.2 Chain-of-Thought Prompting

Chain-of-thought (CoT) prompting represents a further step towards demonstrating to the LLM how to think. This involves instructing the model to take its time and providing concrete examples of how to approach certain tasks and in what order to execute them. Wei et al. (2022) [5] provides examples of differences between standard prompts and chain-of-thought prompts. An example of how the two different prompts could look is illustrated in Figure 3.6. Another method of triggering CoT-reasoning is by adding a sentence to a prompt, such as "Let's think step-by-step", to encourage reasoning and step-by-step problem solving [26]. Even without providing concrete examples, this can improve the CoT ability of the model.

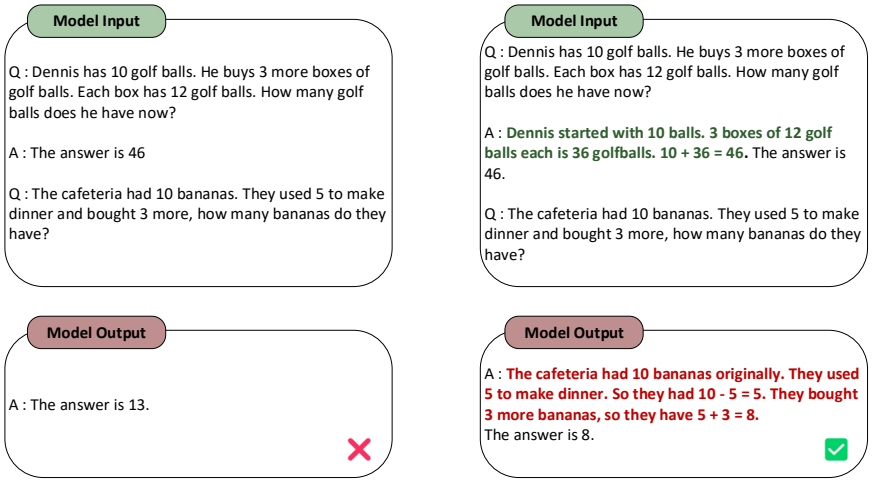


Figure 3.6: *Example of different model inputs and outputs with and without chain-of-though prompting. Different coloured text indicates additions when writing CoT.*

Although the size and performance of LLMs have significantly improved in recent years, this advancement is not sufficient for all levels of problem solving, particularly for arithmetic and common sense problems, which continue to pose challenges. Therefore, enhancing performance through chain-of-thought prompting is a relevant strategy [5].

3.5 Reinforcement Learning

As the computational power available increases, more computationally heavy methods for control grow in interest. Reinforcement Learning is a branch of Machine Learning where system dynamics is learned in a trial and error manner, by interacting with an environment over many sequential episodes. The goal of reinforcement learning is to let an agent learn good policies for sequential decision problems, by optimizing cumulative future rewards. To do this the agent interacts with a simulated environment and receives both a reward and a next state based on the taken action.

In Figure 3.7, the interaction between an agent and a simulation environment is illustrated over discrete time steps $t = 0, 1, 2, 3, \ldots, n$. At every time step (t), the agent receives information about environment state $s_t \epsilon \mathcal{S}$ from the simulation. Based on this state, the agent selects an action $a_t \epsilon \mathcal{A}(s_t)$ from the available action space. At next time step the agent receives a numerical reward $r_{t+1} \epsilon \mathcal{R}$ and an updated state s_{t+1} from the environment. This is done over multiple time steps, forming an episode.

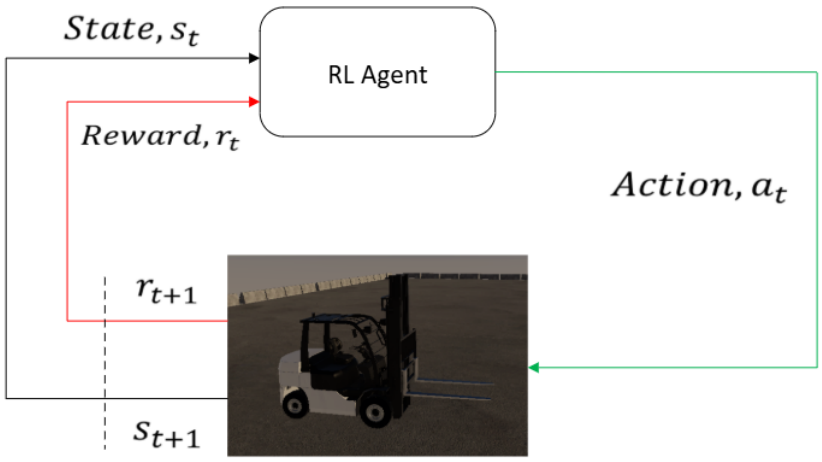


Figure 3.7: *A schematic figure of how an agent interacts with a simulated environment.*

During each time step, the agent maintains a mapping from states to actions, aiming to learn which action maximizes cumulative reward at different states. After a large number of episodes the mapping should have a high probability of choosing the action a_t which returns the highest reward r_{t+1} at state s_t. This mapping is called the agent's policy π_t.

There are different Reinforcement Learning methods and they differ in how the policy is updated based on the agent's experiences.

3.5.1 Exploration/Exploitation

Since the agent does not know the environment or the dynamics, it does not know the optimal actions initially. Reinforcement Learning can be seen as a trial-and-error approach, where the agent plays around with different actions to learn which state and action combination returns the highest reward. As the number of performed trials increases and the agent gains more experience, it can make improved decisions. Thus, in deep reinforcement learning there is a tradeoff between choosing an action based on trained policy or a random action sampled from action space. Initially during the learning process the amount of random actions is set to be high, the exploration phase. The probability of choosing an action based on policy increases with the number of trained episodes. In the end the amount of random actions should be low, the exploitation phase.

One exploration/exploitation method is called epsilon greedy where the probability to choose random action ε_p decays exponentially as the number of episodes increases:

$$\varepsilon_p = \varepsilon_{\text{end}} + (\varepsilon_{\text{start}} - \varepsilon_{\text{end}}) \cdot e^{-\text{k}/\varepsilon_{\text{d}}} \tag{3.29}$$

where $\varepsilon_{\text{start}}, \varepsilon_{\text{end}}$ is the start and end probabilities, k is the current episode and ε_d is the decay speed of the probability. With this method the agent explores the environment in the start and as the number of executed episodes increases it maximizes the rewards by following the learned policy in the exploitation phase.

3.5.2 Markov Decision Process

Markov Decision Process (MDP) is a model for sequential decision making, also called stochastic dynamic program. The problem above can be modeled as a simplified MDP, which in general is denoted as a tuple [A, S, R, P], where

- $\mathcal{S}$ is the set of possible environment states,
- $\mathcal{A}$ is the set of actions,
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$ is the reward function, and
- $P : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the state transition probability function.

In reinforcement learning the function $\pi : \mathcal{S} \to \mathcal{A}$ defines as a stochastic stationary policy of MDP. The policy π maps states s_t to actions a_t in the action space, denoted as $a_t = \pi(s_t)$.

The sequential reinforcement learning problem, seen in Figure 3.7 can be modeled as an MDP. At each time step t, the agent observes the current state s_t and selects the action a_t according to the policy $\pi(s_t)$, receives a reward r_t, and transitions to the next state s_{t+1}, governed by the transition probability function P and reward function $\mathcal{R}(s_t, a_t, s_{t+1})$.

At every time step, a set of state, action and reward are collected. Every episode forms a sequence of the these transition sets, a Markov Decision Process. The MDP relies on the Markov assumption, the probability of the next state s_{t+1} depends only on the current state s_t and action a_t, not on proceeding states or actions [27].

3.5.3 Value Function and Quality Function

For every episode, according to the Markov Decision Process, the total reward can be calculated as

$$R = r_1 + r_2 + ... + r_n. \tag{3.30}$$

Furthermore, the total future rewards from time t is calculated as

$$R_t = r_t + r_{t+1} + ... + r_{t+n}. \tag{3.31}$$

If the problem is an infinite horizon problem, the future reward will grow unboundedly. If a discount factor $\gamma \epsilon [0, 1]$ is added to the future rewards,

$$R_t = r_t + \gamma r_{t+1} \gamma^2 r_{t+2} + ... + \gamma^{t+n} r_{t+n} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k}. \tag{3.32}$$

the reward will not diverge and focus on either short term reward or long term rewards according to the value γ.

The best policy of the agent is to choose an action that maximizes the discounted future rewards. The goal is to estimate an optimal policy π^* which satisfies

$$J_{\pi^*} = \max_{\pi} \mathbb{E}_{\pi} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \tag{3.33}$$

where J_{π^*} is the expected total reward of the state trajectory given by policy π^*. Next, the optimal state value function can be formulated as

$$V_{\pi^*}(s) = \mathbb{E}_{\pi^*} \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s \right] \tag{3.34}$$

which is the expected reward of following policy from state s. Furthermore the optimal state action function can be formulated

$$Q_{\pi^*}(s, a) = \mathbb{E}_{\pi^*} \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s,\ a_0 = a \right] \tag{3.35}$$

which defines the expected value of an action at state s.

The Bellman optimality function defines recursive relationships for the value function and state action functions [28]. So according to the Bellman optimal-

ity function, the optimal value function satisfies

$$V^*(s) = \mathbb{E}\left[R_{t+1} + \gamma \max V^*(s_{t+1}) \mid S_t = s\right] \tag{3.36}$$

and the optimal state-value function satisfies

$$Q^*(s,a) = \mathbb{E}\left[R_{t+1} + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}) \mid S_t = s, A_t = a\right]. \tag{3.37}$$

The optimal state-action function is

$$Q^*(s,a) = \max_{\pi} Q_\pi(s,a) \tag{3.38}$$

and the optimal policy is

$$\pi^*(s) = \arg\max_{a} Q^*(s,a) \tag{3.39}$$

3.5.4 Temporal Difference and Q-Learning

Without any knowledge about the environment, one central reinforcement learning approach is temporal difference learning (TD). The method tries to learn optimal policies without knowledge of the environment and its dynamics. The policy is learned from experience and a temporal difference error update rule. In Q-learning this method is used with the update rule for state-action function. The update rule is

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha \Delta TD \tag{3.40}$$

where α is the learning rate and

$$\Delta TD = \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right]. \tag{3.41}$$

Q-learning is called an off-policy temporal difference method. The state action function $Q(s,a)$ is used to estimate future reward of an action and to learn the optimal policy. $Q(s,a)$ is often seen as a lookup table where a specific state and action generate a future reward value. This is useful if the state and action spaces of the problem is small and discrete. But in many real world problems, the state and action spaces are large and continuous. One approach could be to discretize the state and action spaces. This would work fine for small problems, but as the state and action spaces grow the Q-table gets huge, often with more values in the table than there are atoms in the known universe [27]. Thus, another approach is needed for bigger problems. If the Q-table is approximated to a functional, $f(s,a,\theta)$, the $Q(s,a)$ value can be approximated without the need to know the exact value of each state and action pair in the spaces. The approximated functional $f(s,a,\theta)$ is often a neural network where parameters θ are network weights. The method is called deep Q learning.

3.5.5 Deep Q Learning

When the Q-table is approximated with a neural network, the method is called Deep Q Learning. The neural network finds features in the Q-table and can then

approximate it with a functional.

Experience batch and training

As the agent explores the environment, the experience set of state, action, reward and next state is saved to an experience memory buffer in each time step. This is a widely used trick in deep learning to enhance convergence of the algorithm. When training the Deep Q network, a batch of the experiences, $\{s_t, a_t, r_t, s_{t+1}\}$, are collected from the memory and replace the most recent transition. This will make the training more stable since subsequent similar samples are randomly permutated, reducing the risk that the network parameters converge to a local minimum.

Deep Q Network

The main idea of Q-learning is that given the Q-function $Q(s, a)$ which gives the reward for an action, an optimal policy π^* can be created. Since this function is not available and the knowledge of the environment is unknown, it needs to be approximated. Neural networks are universal function approximators and can act as the Q-function when trained. The constructed neural network is a feed forward network which take in a state and generates an action.

One important part of the DQN algorithm is the use of a target network. The target network $f(s, a, \theta^-)$ is a neural network with the same dimension as the policy network. But the parameters θ^- is softly updated every step from the policy network, according to the update rate τ.

$$\theta^- = \tau\theta + (1 - \tau)\theta^- \tag{3.42}$$

Thus, the target net is not updated as fast as the policy net, and can give a more stable estimation of the expected Q values.

3.5.6 Training Update/Optimization Step

During training, the neural network parameters are updated through an optimization step. In this step, the predicted Q-values from the policy network are compared to the target Q-values computed from the Bellman equation. This difference, called temporal difference in Section 3.5.4, is minimized using a Huber loss function. The Huber loss acts like the mean squared error when the error is small, but like the mean absolute error when the error is large. This makes it more robust to outliers when the estimates of $Q(s, a)$ is noisy [29].

The Huber loss is calculated over a batch B of transitions:

$$\mathcal{L} = \frac{1}{|B|} \sum_{(s_t, a_t, s_{t+1}, r_t) \epsilon B} \mathcal{L}(\Delta TD) \tag{3.43}$$

where

$$\mathcal{L}(\Delta TD) = \begin{cases} \frac{1}{2}(\Delta TD)^2 & \text{for } |\Delta TD| \leq 1 \\ |\Delta TD| - \frac{1}{2} & \text{otherwise} \end{cases} \tag{3.44}$$

Based on the loss $\mathcal{L}$ the policy network parameters can be updated with respect to learning rate. A method for this is the Adam optimizer [30].

Transfer Learning

A big limitation in training reinforcement learning agents is gathering real world data, simulation environments are widely used to train agents [31]. This introduces a problem since there is always a gap between the simulated environment and the real world where the trained agent should operate. This gap will negatively impact the performance of pre-trained policies once transferred to real world agents. By closing the gap between simulation and real world, the performance will increase.

One approach to decrease the gap, is to train the simple dynamics in simulation, and then continue to train the agent in the real world. With this approach, the agent can use the knowledge learned in the simulation and then train the real world details in the final stages of training. When transferring to a new environment, to learn the best actions the exploration rate is increased to let the agent explore actions again. But it is set lower than initial training to balance learning new dynamics with using knowledge from the previous training session.

4

LLM Parameter Tuning

To investigate and evaluate the Large Language Models ability to tune automatic control parameters, developing a framework capable of utilising the LLM together with a simulation environment is required. The following chapter describes the development steps and methodology used to build the LLM parameter tuning framework.

4.1 System Overview

The overall system framework contains multiple different sub-systems, all interacting to give the desired functionality. In Figure 4.1, an overview of the framework can be seen with visualization of the LLM multi-agent workflow, simulator based parameter evaluation loop and the result formatting.

4.1.1 Docker Containers

One method used to simplify software development is utilising Docker containers. By creating multiple specific closed working environments with necessary packages installed and dependencies solved, development and running of a system can take place on any computer as long as Docker is installed. Due to different requirements between the simulation/ROS environment needing Ubuntu 20.04 and the LLM interaction environment needing Ubuntu 24.04, splitting the system into two different containers was necessary. Communicating between the two, a Flask RESTFul API was used [32] allowing for requests to be sent between the two containers.

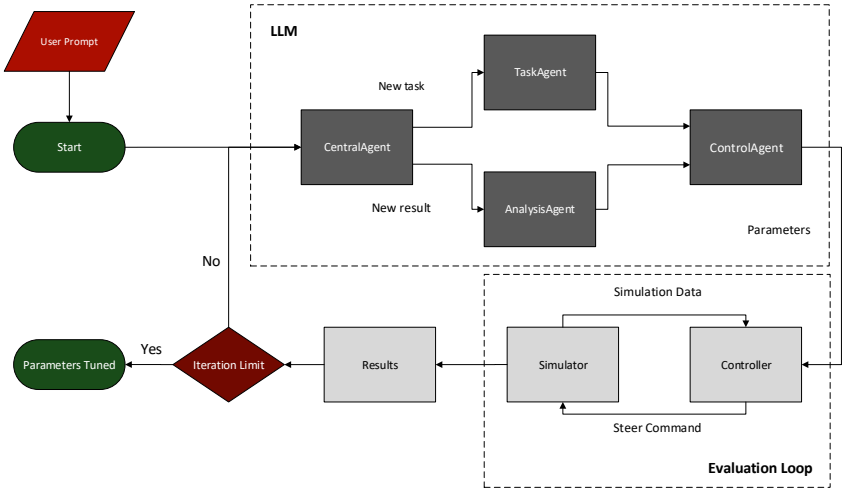


Figure 4.1: *System overview of LLM parameter tuning framework.*

4.2 Simulation Environment

To allow for testing and evaluation of control parameters, a simulation environment is needed. This simulation is a Unity-based simulator containing a forklift and two different environments: an indoor environment resembling a warehouse, and an outdoor open world. Communication with the simulator and the forklift within is done using ROS (Robot Operating System). ROS uses topics and services, on which you either publish and subscribe to a topic or request a service. Using topics, steering commands can be sent to the simulated forklift, and positional information can be extracted from it.

The steering commands sent on the topic `\vehicle_command` are split into two parts: steering angle and velocity. Positional data from the simulation is formatted as global positional coordinates and a heading angle, published on the topic `\localization\ground_truth`. This format allows automatic control algorithms to access positional information and generate vehicle commands to control the forklift truck. The methods of communication with the simulation are visualised in Figure 4.2.

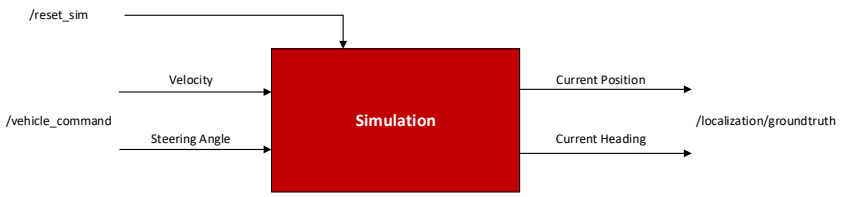


Figure 4.2: *ROS communication with simulation environment including topics and reset service.*

4.3 Controller Types

Below are the tunable controller parameters presented together with equations for calculating the steering angle δ in the different cases.

4.3.1 Pure Pursuit

The LLM will tune the look-ahead horizon for the pure pursuit controller. The steering angle will be calculated as described in Section 3.2.1.

4.3.2 Linear-Quadratic Regulator

The LLM will tune parameters for the LQ controller. The parameters are the Q and R matrices. Since the controller minimizes two states θ_e and d_e, the Q-matrix will be 2x2 dimensional. The steering angle δ is used to control the states, thus the R matrix will be 1x1 dimensional:

$$\begin{aligned} \mathbf{Q} &= \begin{bmatrix} q1 & 0 \\ 0 & q2 \end{bmatrix} \\ \mathbf{R} &= \begin{bmatrix} r1 \end{bmatrix} \end{aligned} \tag{4.1}$$

The feedback gain is calculated as described in Section 3.2.2, where the feedback gain is presented as:

$$\mathbf{L} = \begin{bmatrix} k1 \\ k2 \end{bmatrix} \tag{4.2}$$

To ensure that the controller works, even when the distance from the path is large [7], a nonlinear feedback control signal is constructed as:

$$u = u_0 - k_1 \frac{\sin \theta_e}{\theta_e} d_e - k_2 \theta_e \tag{4.3}$$

where

$$u_0 = c(n+1) \tag{4.4}$$

is the curvature of a path segment ahead. From the calculated value of u the steering angle is computed as:

$$\delta = \arctan(-uL_w) \tag{4.5}$$

since the motion model is a single track model, described in Section 3.1.1.

4.3.3 State-Feedback

The LLM will tune feedback gains for state feedback controller. The parameters are

$$\mathbf{L} = \begin{bmatrix} k1 \\ k2 \end{bmatrix}. \tag{4.6}$$

Since the controller minimizes the states θ_e and d_e, there will be 2 feedback gains to tune. The control signal u and steering angle will be calculated in the same way as for the LQ controller above.

4.4 Large Language Model

The Large Language Models used within this thesis project are all pre-trained. Therefore, domain expertise in automatic control depends on the domain knowledge included in the data on which the models were originally trained. These pre-trained models differ in size. One model is accessed from OpenAI through the OpenAI API; GPT-4.1 is the model trialled. Another model tested for its parameter tuning ability is an open-source model run locally, utilising an RTX 4090 graphical processing unit (GPU). The model run on the local GPU is Llama 3.2 and is smaller due to the limitations on a local machine. There are methods for further training Large Language Models on additional domain-specific information to improve performance, although this thesis will not include this method of LLM utilisation due to the cost of tuning a model [14] and the need for a large amount of domain relevant text data. All performance is based on agentic programming, agentic structure, and in-context learning through prompt design.

4.4.1 Agentic Programming

Agentic programming is the use of agents when interacting with and utilising Large Language Models. LLM agents allow for expanded usage of LLMs by combining traditional prompting to generate a response with additional functionality, such as memory and tools. Additionally, agents can assume a persona, improving the LLMs' ability to act as a human being would in the same role. The creation of these agents is done through a system prompt, describing who the agent is, what its task is, and how it should go about solving any task it is given. By allowing agents to specialise in certain aspects of a larger task, the outcome can be improved. Therefore, four different agents are created to assist in the implementation of the thesis project's LLM parameter tuning framework. The individual agents are CentralAgent, AnalysisAgent, TaskAgent and ControlAgent, and are described later in the thesis report.

Each agent being able to perform a certain aspect of a given task is an important part of utilising LLMs in an optimal way. To build a complete framework, the interactions between agents is also important. In this case, creating a multi-agent system or workflow allows for these agent-to-agent interactions.

4.4.2 Multi-agent Workflow

A multi-agent workflow is a collection of LLM agents interacting and collaborating to solve a problem. When designing a workflow, a few different factors are taken into consideration: what order agents are called upon, what relevant information needs to be available for each agent and how many agents are needed. Taking this into account, two different workflow structures are implemented.

Single Workflow

A complete single workflow containing four LLM agents provides a modular design in which a CentralAgent, acting as a supervisor, directs the flow of information and the order in which tasks are solved. Including all agents in a single workflow allows for more straightforward memory management and information passing. Figure 4.3 provides a visualisation of the complete workflow. The CentralAgent takes in a user prompt and delegates the task to the relevant agent depending on the contents of the prompt. Finally, the ControlAgent receives information from the previously selected agent, performs its task, and provides an output containing control parameters.

Split Workflow

An alternative strategy is splitting the workflow into multiple parts. By splitting the workflow and placing each agent on their own, agent interaction becomes dependent on the code structure, and information passing must be done manually. The agents communicate only according to the code written. Agent-to-agent interactions and collaboration require intelligent decision making; incorrect information passing or failure to prompt the next agent in the workflow halts the en-

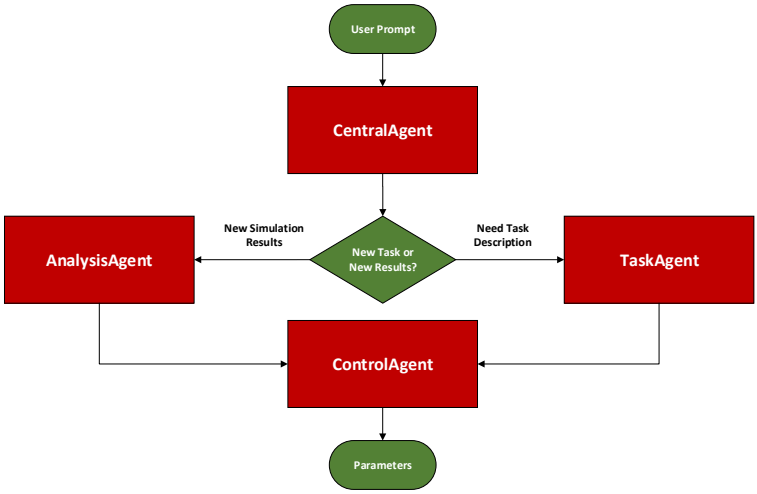


Figure 4.3: *Complete combined LLM multi-agent workflow for parameter tuning. Agents and interaction between them are shown in the figure.*

tire planned function of the agent workflow. For the smaller open-source model, where the required collaborative behaviour is less consistent from the model, this simplified structure will be used. An overview of this split workflow structure is presented in Figure 4.4.

The iterative process is similar to the single workflow, except for the absence of a CentralAgent. Instead, the actions of the CentralAgent are incorporated into the code structure. The TaskAgent creates a task description, the ControlAgent generates an initial set of control parameters, the AnalysisAgent provides performance feedback based on simulation results, and the ControlAgent tunes the control parameters. The AnalysisAgent and ControlAgent are then called at every iteration to tune the control parameters. When transitioning from one agent to the next, information is saved from the previous agent and re-initialized in the next using a dictionary data type.

Although two different structures of agent workflows are implemented, both utilise the same framework, LlamaIndex [33], to build a multi-agent workflow with memory, function calling, and inter-agent collaboration.

4.4.3 CentralAgent

CentralAgent is used in the single workflow strategy, which serves as the supervisor within the multi-agent system. CentralAgent is responsible for the initial processing of user inputs provided as user prompts. Its primary role is to assess

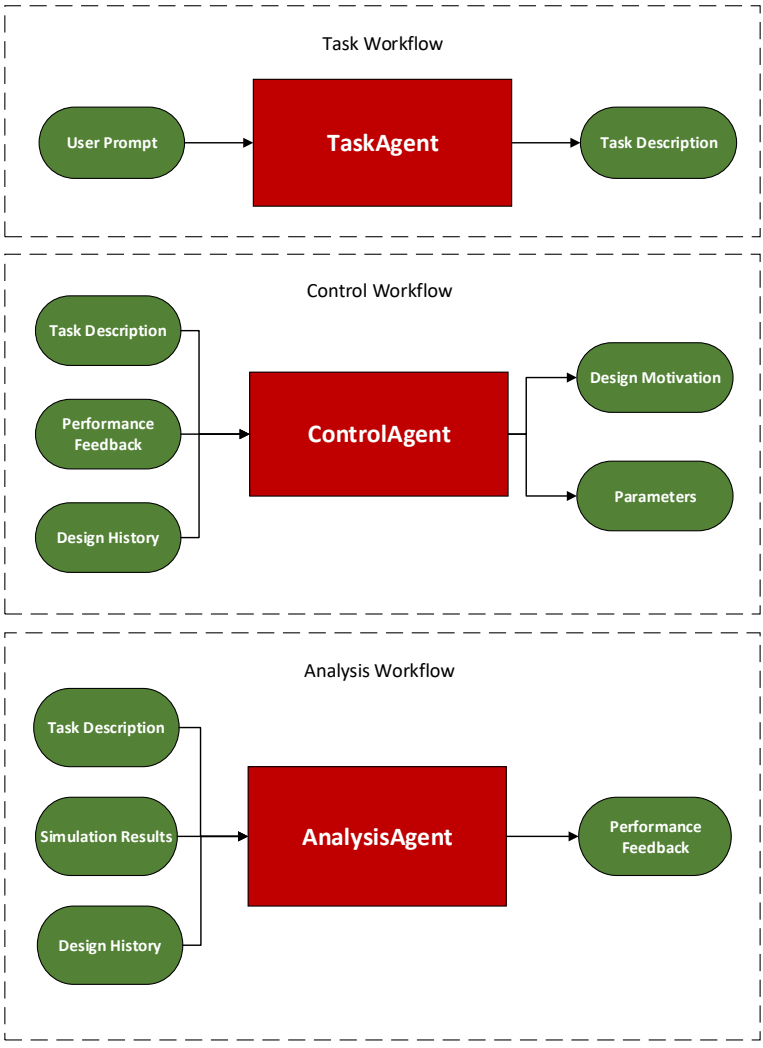


Figure 4.4: *Alternate structure of LLM multi-agent workflow for parameter tuning using split workflows. Agents and interaction between them are described in the figure.*

the content and context of these prompts to determine the appropriate course of action: whether to engage a AnalysisAgent or a TaskAgent. If the user input relates to new results that require evaluation or analysis, CentralAgent hands the

query off to the AnalysisAgent, which is tasked with analysing the current results and providing constructive feedback based on predefined criteria. On the other hand, if the input indicates a need for a task description to be generated, such as outlining objectives or specifying operational parameters, CentralAgent delegates this responsibility to the TaskAgent. This decision-making process ensures the system remains efficient by avoiding unnecessary work when the user input does not require it. A hierarchical structure allows for smooth flow of communication within the multi-agent workflow.

4.4.4 TaskAgent

The TaskAgent is designed to generate a comprehensive task description following the hand-off from the CentralAgent. Upon receiving the user prompt, TaskAgent utilises its natural language processing capabilities to extract relevant details and formulate a coherent task description. This description includes essential elements such as task objectives, performance requirements, and potential constraints placed upon the system. Once the task description is complete, TaskAgent hands it off to the ControlAgent. The ControlAgent is then responsible for translating this task description into control parameters. By splitting task description and parameter generation, the domain-specific knowledge of each agent can be increased, and the risk of misinterpretation and hallucinations decreased.

4.4.5 ControlAgent

The ControlAgent plays a pivotal role in the multi-agent system by generating and fine-tuning control parameters based on the task description received from the TaskAgent. Upon receiving the task outline, ControlAgent first identifies the required control type specified within the description. It then utilises this information to create an initial set of control parameters tailored to the task's objectives and constraints. To refine these parameters further, ControlAgent actively considers feedback provided by the AnalysisAgent, incorporating valuable insights gained from previous executions and evaluations. Additionally, the ControlAgent reviews and compares the new parameter set against previous iterations to ensure parameters are not evaluated multiple times. This iterative process of parameter tuning enhances the effectiveness of the procedure and creates human-like behaviour by taking multiple factors into consideration when tuning parameters. The generated parameters are then evaluated in the simulation environment.

4.4.6 AnalysisAgent

The AnalysisAgent is called upon by the CentralAgent when new results from the simulation are available for analysis. Its primary function is to evaluate these results, focusing on pre defined aspects of performance relevant to the path-following scenario, described in more detail under Performance Feedback in Section 4.4.7. The AnalysisAgent examines various aspects of the agents' actions and

their effectiveness in achieving the outlined objectives. Based on this analysis, it generates targeted feedback that identifies both successful aspects and areas needing improvement. This feedback is then communicated to the ControlAgent, providing essential information that can be used to adjust and fine-tune control parameters accordingly. By systematically assessing performance outcomes and producing actionable insights, the AnalysisAgent ensures that the ControlAgent has the necessary data to optimise the parameters and enhance the overall operation of the path-following task.

4.4.7 Agent Inputs and Outputs

Task Description

The TaskAgent receives the user prompt, seen in Figure 4.9, and aims to generate a task description. To make it as clear as possible for the ControlAgent to use the task description, it is formatted as a structured output with instructions to divides the description into:

- Objective: Clearly define the main task or goal derived from the user input.
- System Description: Identify the system components involved (e.g., type of controller, target application).
- Control Parameters: List the parameters to be tuned or generated, including any specific matrices or variables like Q and R in a Linear Quadratic controller.
- Performance Requirements: Outline the key performance metrics that need to be evaluated, such as stability, total error, and response optimization.
- Constraints and Conditions: Include any specific conditions or constraints provided by the user that must be adhered to during the task.

This task description is then kept available to the LLM agents to ensure the goals of the task are always known.

Control Parameters

The control parameter output is generated by the ControlAgent, consisting of two parts. One is the control parameters, the other is a short design motivation meant to improve the design history. By providing a reasoning behind a choice of parameters, future tuning iterations can utilise previous motivations when motivating new parameter choices.

Simulation Results

Results from the simulation are formatted in a certain format to ensure comparable information to the AnalysisAgent every time the agent is called and to limit the amount of data to analyze. Context size is a relevant factor which impacts the

amount of information a LLM can analyse all at once. Therefore, the following results are presented to the LLM:

- distance error: The distance error between the reference path and the actual path traveled.
- heading error: The heading error between the reference path heading and the actual path traveled.
- accumulated distance error: The total accumulated distance error for the path following simulation.
- accumulated heading error: The total accumulated heading error for the path following simulation.
- termination reason: Reason why the path-following process terminated.
- performance reward: Numerical reward indicating how well the control parameters performed.

Based on these inputs, the AnalysisAgent generates feedback on the control parameter performance.

Performance Feedback

Performance Feedback is generated by the AnalysisAgent based on the task description, the design history, described below, and the simulation results. This feedback is meant to be clear and act as a summarised version of how the previous set of parameters performed. To ensure clear and structured feedback, the AnalysisAgent is given an instruction how to format the performance feedback:

- A brief statement on overall performance highlighting strengths or key issues.
- Identify specific errors or behaviors that require improvement.
- Provide actionable recommendations for tuning control parameters (e.g., which gains to adjust).
- Conclude with the expected effects of your recommendations on future system performance.

These four aspects ensure that the simulation results are captured in a short summary, providing the ControlAgent with the best chance to improve the control parameter performance.

Design History

Keeping track of design history is important to ensure that the LLM agents are aware of which parameters have been previously evaluated and how well they performed. The design history consists of three data points: design motivation, control parameters and performance reward. Figure 4.5 provides an example of

the design history. The first two data points are taken from the control parameter output from the ControlAgent, and the reward is added after evaluation is performed in the simulation environment.

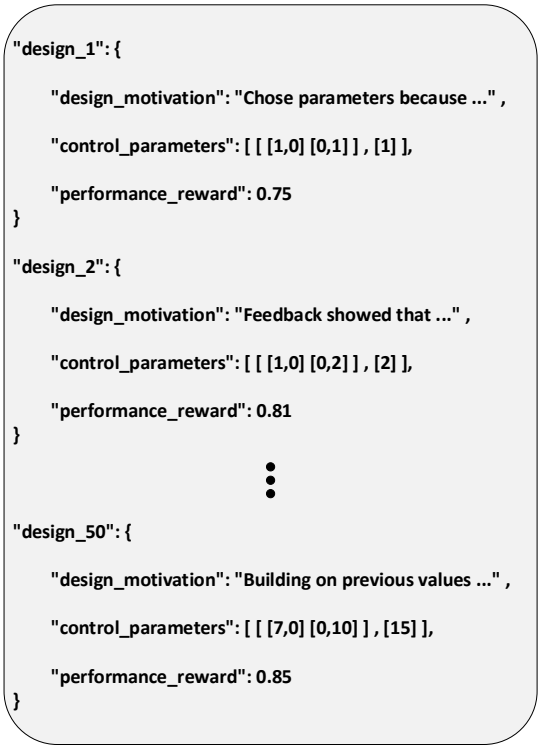


Figure 4.5: *Example of design history for the LLM agents.*

Agent I/O Overview

CentralAgent:

- **Inputs:** User prompt.
- **Outputs:** None, hands off to TaskAgent or AnalysisAgent within the single workflow.

TaskAgent:

- **Inputs:** User prompt with additional task information.
- **Outputs:** Task description

ControlAgent:

- **Inputs:** Task description, performance feedback and design history

- **Outputs:** Control parameters, including design motivation.

AnalysisAgent:

- **Inputs:** Task description, simulation results and design history.
- **Outputs:** Performance feedback.

4.5 Prompt Design

Designing prompts for the thesis project requires two main types of prompts: system prompt and user prompt. The system prompt acts as an agent initializer, assigning a persona and providing instructions. The user prompt acts as the input from the user to the system and the Large Language Model. It therefore contains the specific task the user would like the LLM to solve, along with any requirements placed upon the system or solution. More clear explanations of the respective prompt type are described later in this section.

The Large Language Models used in evaluation are general pre-trained models. Consequently, the pre-trained domain-specific expertise of a model depends on the textual data used to train it. Since the objective requires certain domain-specific knowledge, the amount of in-context learning required for successful parameter tuning is to be evaluated.

To evaluate which prompt type is most effective, a combination approach will be implemented. Four different prompt types will be used: zero-shot (ZS), zero-shot chain-of-thought (ZS-CoT), few-shot (FS), and few-shot chain-of-thought (FS-CoT). The specific design factors of each prompt type will be presented in the following sub-sections.

A set of prompts used to evaluate LLM parameter tuning viability can be found in Appendix A.

4.5.1 System Prompts

The system prompts form the basis for each agent and guide how they approach a problem. These prompts are tailored to each agent and their specific task. The agent-specific tasks can be divided into two groups. The first includes the supervisory role of the CentralAgent and the task description role of the TaskAgent. Both tasks are commonly performed by LLM agents: making a simple decision based on input for the first agent, and summarising and structuring information for the second. Therefore, these tasks are assumed to be within the capabilities of the pre-trained model. As a result, the system prompts for these two agents will only be zero-shot.

The second group consists of the AnalysisAgent and the ControlAgent, which handle tasks that require more specialised knowledge. Both analysing results from the simulation and tuning control parameters based on feedback require

domain-specific knowledge, meaning additional in-context learning may be necessary to achieve satisfactory results. To evaluate different levels of in-context learning, system prompts of four types (ZS, FS, ZS-CoT, and FS-CoT) have been created. These are based on the example structures shown in Figures 4.6 and 4.7. In Figure 4.8, a zero-shot system prompt for the ControlAgent is presented to provide a concrete example of how a system prompt looks.

In the design of the few-shot system prompts for each control structure and each agent, AnalysisAgent and ControlAgent, the examples included are meant to describe and capture commonly occurring situations and to describe situations the LLM may otherwise miss or interpret incorrectly. For the AnalysisAgent, oscillatory driving behaviour is specifically described to improve the agents ability to identify instability. Overshoots when turning are also described to ensure the AnalysisAgent does not misinterpret those situations.

For the ControlAgent, other aspects are important to describe: Specific actions based on feedback, how to dampen oscillations or minimize heading and distance errors. The examples of these situations are example inputs and outputs containing numerical values and changes to the parameter values, see Appendix A. From this, the ControlAgent can gain a reasonable understanding for both initial guesses and what magnitude of changes are reasonable to make if the feedback and reward are not as good as desired. As a result, both initial values and tuning efficiency can be improved.

The difference between zero-shot and few-shot prompts lies in the replacement of specific instructions with explanatory examples. When adding chain-of-thought to these two prompt types, this is done by first including the line "Let's think step-by-step". The instructions are also altered accordingly: the ZS-CoT prompt provides a more detailed definition of the steps to take, while the examples in the FS-CoT prompt include a step-by-step solution rather than just an answer.

4.5.2 User Prompts

The second prompt type is the user prompt, which acts as the user's input to the LLM, containing the task the user wants the LLM to solve. Specific instructions such as requirements, limitations, or any system-related details are included in the user prompt. If the system prompt acts as an activation prompt for each agent, then the user prompt serves as the starting trigger, instructing the LLM on the details of the task.

For the single workflow, two user prompts are required: one containing the task description, as seen for an LQ controller in Figure 4.9, and a second issued when the simulation is completed to reinitialize the LLM system in order to analyze the results and generate a new set of control parameters. In the split workflow, since the agents do not directly interact with each other, a user prompt is used for each agent, instructing them on the current stage of the tuning process and how to proceed. The initial user prompt in Figure 4.9, from which the TaskAgent generated a task description, remains the same for both workflows.

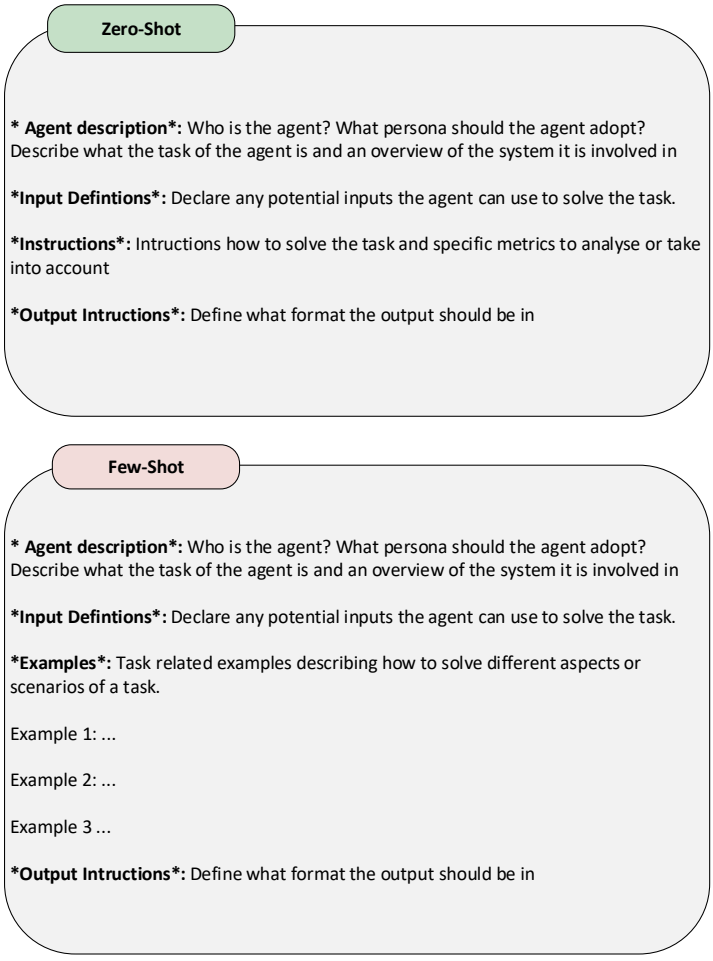


Figure 4.6: *Prompt examples for ZS and FS prompts.*

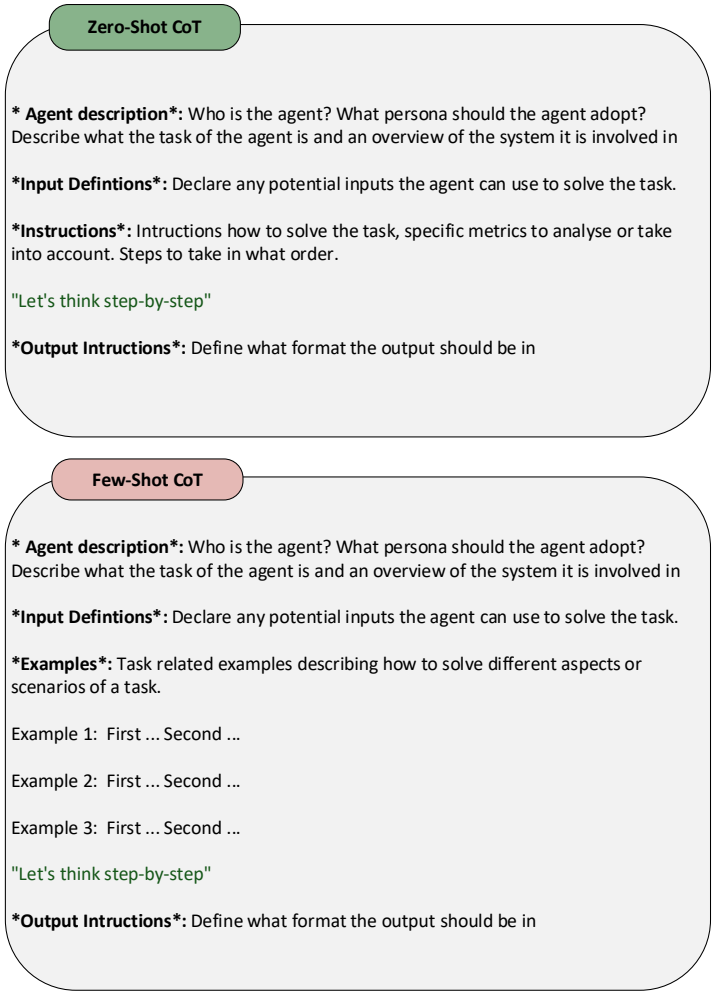


Figure 4.7: *Prompt examples for ZS-CoT and FS-CoT prompts.*

As shown in Figure 4.8, which displays a system prompt, the agents are provided with a detailed description of both their role and instructions from the system prompt. The user prompt, instead, provides specific task details to the agents which, if included in the system prompt, would lock the agent to one specific task. This allows the same system prompt, for example, in the TaskAgent, to be used across all three controller types.

System Prompt (ZS)

```
You are an expert control engineer tasked with determining the state weighting matrix
(Q) and the control weighting matrix (R) for a Linear Quadratic Controller (LQ Controller).
Based on the task description, performance feedback from previous parameters, and design history,
suggest values for Q and R.

### Inputs ###
- **task_description**: Description of the task objective, system description, parameters to tune,
                        requirements, and constraints.
- **performance_feedback**: Feedback on simulation results from the most recent control
                        parameters. Includes performance overview, strengths, weaknesses, and
                        suggested improvements.
- **design_history**: A history of previously evaluated control parameters, their design motivations,
                        and associated performance reward indicating level of success.

### Task Instructions ###
You are to generate and iteratively fine tune the control parameters for an LQ controller on a forklift
truck. These parameters will be evaluated,and feedback will be provided to allow for performance
improvements. Use the task description, performance feedback, and the performance reward from
the design history to propose and tune new better parameters aiming to find the parameters with the
largest performance_reward.
The design history should prevent repeating identical parameters and highlight high-performing sets.
The Q matrix should be formatted as Q = [[q1, 0], [0, q2]], where q1 penalises distance_error and q2
penalises heading error.
The R matrix should be formatted as R = [[r1]], where r1 reflects the cost of applying the control input.

### Output Instructions ###
Respond strictly in the following JSON format with three keys: 'design', 'Q', and 'R'.
 - 'design': A concise, three-sentence motivation explaining the parameter choices.
 - 'Q': The numerical 2x2 matrix as a nested list (e.g., [[q1, 0], [0, q2]]).
 - 'R': The numerical 1x1 matrix as a nested list (e.g., [[r1]]).
Do not include any text other than the JSON object.

### Example Output ###
{
 "design": "design_motivation",
   "Q": [[q1, 0], [0, q2]],
   "R": [[r1]]
}
```

Figure 4.8: *Zero-shot system prompt for the ControlAgent.*

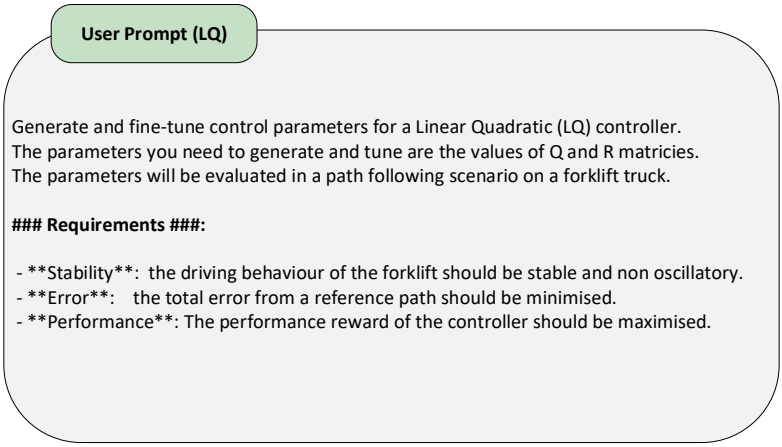


Figure 4.9: *User prompt for the tuning the control parameters of an LQ controller.*

5
Reinforcement Learning

The following chapter describes the development steps and methodology used to train the RL policy.

5.1 Simulation Environment

To train the reinforcement learning (RL) agent, an environment is required where the agent can act, observe state transitions, and evalutate rewards. For training with Deep Q-Networks (DQN), two different simulation environments were utilized. Initially, a lightweight mathematical simulator was used to rapidly train basic control behaviors. In the later training sessions, a Unity-based simulator was used to enable learning of more realistic forklift-specific behaviors with more physical constraints.

5.1.1 Unity Simulation

The simulation environment used to train the Reinforcement Learning policy is the same as the one used for the LLM solution. See Section 4.2 for details regarding the Unity simulation environment.

5.1.2 Mathematical Simulation

Since the Unity simulator operates in real-time, it becomes inefficient for training a large number of episodes. One of the main advantages of using simulation instead of real-world training is the potential to speed up the learning process. To enable this benefit, a lightweight mathematical simulator was developed for the initial training phase.

This simulator is based on the single-track vehicle model, as described in Section 3.1.1. Given the current state of the vehicle (position, heading, and velocity) and a steering input, the next state can be computed using simple kinematic equations. Although this model does not capture the full dynamics of the forklift, it helps the agent to quickly learn basic motion control strategies in a simplified and fast environment.

5.1.3 Transfer Learning

After training in the mathematical simulator, the agent is transferred to the more realistic Unity simulator. Here, the dynamics include additional physical constraints and environmental interactions. To adapt to this new setting, transfer learning is used: the agent is initialized with the parameters learned in the initial phase and continues training. This helps it to quickly adjust and leverage previously learned behaviors to perform well in the Unity-based environment.

5.2 Deep Q Network

The Q-function approximator used in the DQN agent, described in Section 3.5.5, is implemented as a neural network.

5.2.1 Action Space & State Space

In reinforcement learning, the agent aims to learn the optimal action to take in a given state to maximize cumulative reward. The action $a \epsilon \mathcal{A}$ is selected from a discrete action space, and the current situation of the agent is described by the state $s \epsilon \mathcal{S}$.

The dimension and structure of both the action space $\mathcal{A}$ and the state space $\mathcal{S}$ directly affect the complexity of the learning process and the amount of training required. In this thesis, the DQN algorithm is applied using a discrete action space, with steering actions distributed within the interval $[-\pi/3, \pi/3]$.

$$\mathcal{A} = \left\{ \pm\frac{\pi}{3}, \pm\frac{\pi}{4}, \pm\frac{\pi}{5}, \pm\frac{\pi}{6}, \pm\frac{\pi}{7}, \pm\frac{\pi}{10}, \pm\frac{\pi}{12}, \pm\frac{\pi}{15}, \pm\frac{\pi}{20}, \pm\frac{\pi}{30}, 0 \right\} \tag{5.1}$$

Designing an effective state space is critical for enabling efficient learning. While a natural choice for the state might include the absolute position of the vehicle, velocity, and heading. Such a representation results in a large and highly specific state space which fails to exploit the local structure of the vehicle and the path. This would require the agent to visit a high number of unique states to learn effective behavior across the entire environment.

To address this issue, the state space was instead designed around relative quantities. Specifically, the state includes:

- The lateral distance error to the nearest path segment.

- The heading error relative to the orientation of the closest path segment.
- The current steering angle of the vehicle.
- The curvature of an upcoming path segment, enabling a degree of look-ahead.

This relative representation significantly reduces the size of the state space. Since similar errors can occur repeatedly along different parts of the path, the agent can generalize more effectively and requires fewer unique experiences to learn a robust policy.

5.2.2 Network Structure

The neural network architecture used for the path-following problem consists of four fully connected layers, seen in Figure 5.1. First an input layer, containing the state described above. Then two hidden layers are applied, each containing 32 neurons. The output layer is the $Q(s, a)$-value of every action in the action space, given the input state. The chosen steering angle is the action with highest $Q(s, a)$-value. Between each layer, a Rectified Linear Unit (ReLU) activation function is applied.

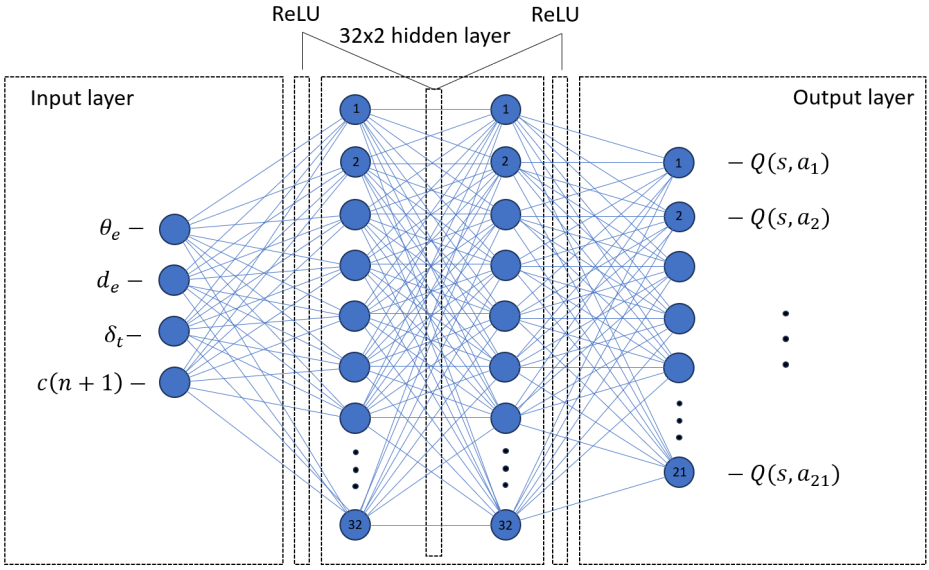


Figure 5.1: *A schematic figure of the neural network structure.*

5.2.3 Reward Function

To guide the learning process, a reward function is designed to evaluate the quality of each action taken by the agent. By shaping the reward function appropri-

ately, specific behaviors can be encouraged, such as smooth motion and accurate path following.

In the path-following problem, the agent receives a positive reward for the following conditions:

- The distance error to the reference path, d_e, is reduced compared to the previous time step.
- The heading error, θ_e, decreases relative to the previous heading error.
- The agent makes progress along the reference path.
- Small steer changes from previous steer angle, promoting smoother control.

The reward each time step t is computed as:

$$\mathcal{R}_t = 30\Delta d_e + 20\Delta\theta_e + 10\Delta n - 1.5\Delta\delta \tag{5.2}$$

where

$$\begin{aligned} \Delta d_e &= \mid d_e^t \mid - \mid d_e^{t-1} \mid \\ \Delta\theta_e &= \mid \theta_e^t \mid - \mid \theta_e^{t-1} \mid \\ \Delta n &= \mid n^t \mid - \mid n^{t-1} \mid \\ \Delta\delta &= \mid \delta^t \mid - \mid \delta^{t-1} \mid \end{aligned} \tag{5.3}$$

After initial training, once the agent has learned actions to move closer to the path between time steps, the reward is modified to penalize total distance error at each step:

- A penalty is applied based on the magnitude of the distance $\mid d \mid$ to the path.

The new reward function is:

$$\mathcal{R}_t = 30\Delta d_e + 20\Delta\theta_e + 10\Delta n - 1.5\Delta\delta - 4 \mid d \mid \tag{5.4}$$

This pushes the agent to be closer to the path.

One approach to alter path-following behaviour, such as potential oscillations, is to modify the reward function and retrain the agent. By reducing the penalty associated with one error, other features are allowed to have a larger impact on the reward resulting in a potential different behaviour.

5.2.4 Hyperparameters

Several hyperparameters are configured to control the training behavior of the Deep Q-Network. These include parameters for learning rate, exploration/exploitation level, and network updates:

- **Batch size:** 64 - The number of transitions sampled from memory for each training update.

- **Discount factor** ($\gamma = 0.95, 0.99$) - Determines the importance of future rewards. 0.95 in the first training session and 0.99 for the second and third training sessions.
- **Exploration start probability** ($\varepsilon_{\text{start}} = 0.9, 0.2$) - Initial probability of selecting a random action. 0.9 in the first two training sessions and 0.2 in the last training session.
- **Exploration end probability** ($\varepsilon_{\text{end}} = 0.05$) - Final probability of a random action by the end of training.
- **Exploration decay:** ($\varepsilon_d = 900$) - Controls how fast the exploration probability decays over time.
- **Target network update rate** ($\tau = 0.01$) - Determines how quickly the target network parameters are updated towards the policy network.
- **Learning rate:** 0.005 - The step size used for updating network weights.

6

Testing and Evaluation

Two different systems are created in the thesis, a Reinforcement Learning solution and a Large Language Model parameter tuning framework. The following chapter describes the tests for each system, what parameters and results are collected from the testing and how the results are evaluated to compare the two systems and answer the research questions presented in Chapter 1.

6.1 Performance Metrics

When comparing two systems that solve the same problem using different methods, it is essential to select performance metrics that are relevant to both systems. This ensures a fair evaluation between the two systems. The chosen metrics can be categorized into two main groups: controller performance and system performance. Each category encompasses several sub-metrics, which are detailed below.

Controller Performance

Three different control structures are evaluated in the LLM parameter tuning system. To compare performance between these three control structures and compare performance against the reinforcement learning system, two different performance criteria are used. These are:

- Total Error - The cumulative error for the full path-following task. The sum of the total heading error and total distance error to the reference path normalized by the number of error values collected. Used in reward calculation.

- Control Effort - The cumulative change in control signal throughout path-following task indicating how much the control signal varied throughout.

Both controller evaluation metrics are described further in Section 6.2.

System Performance

Measuring the overall performance of each system is required to effectively evaluate the two solutions against each other. The criteria used to measure the performance of each system is:

- Iterations - How many simulation iterations need to be performed for the controller performance to achieve good performance.
- Time Consumption - The amount of time the full system takes to solve the task and run the iterations required.
- Effort - How much human effort is needed to both run the system and make required changes when wanting to solve different control structure problems.

6.2 Evaluation

Each controller uses a different objective function to minimize path-following error. To evaluate the path-following performance of the controllers, a custom reward function is constructed. This function rewards small distance and heading errors relative to the path. While not entirely optimal, the reward will act as a general performance metric for comparing the controllers. The evaluation is performed after a complete path following episode on the path seen in Figure 6.1. By evaluating the simulation results, an episode reward can be computed. The reward function is computed as:

$$\mathcal{R}_{\text{eval}} = 1 - \sum_{k=0}^{N} \frac{\theta_e^k + d_e^k}{N} \tag{6.1}$$

where N is the number of evaluated points on the path, θ_e^k is the heading error at step k, and d_e^k is the distance error at step k.

As a complement to this evaluation, the control effort is also considered. By measuring the change in steering angle between consecutive time steps, we gain insight into how much actuation the controller requires to maintain path following. The control effort is computed as:

$$\delta_{tot} = \sum_{k=0}^{N} \frac{\delta_k - \delta_{k-1}}{N} \tag{6.2}$$

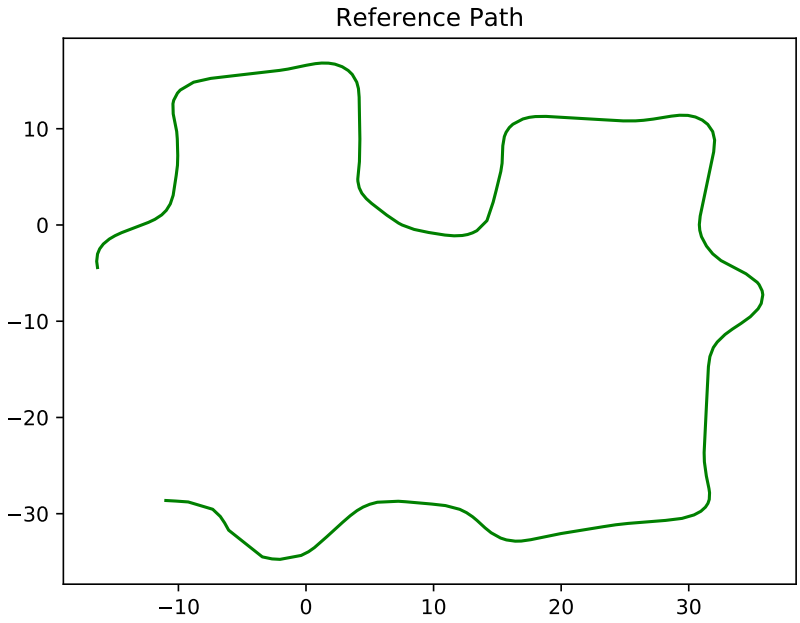


Figure 6.1: *The figure shows the reference path the forklift should follow.*

The total reward will be compared between the RL and LLM solutions to evaluate them against each other.

6.3 LLM Evaluation

Testing of the LLM parameter tuning framework is automated once initiated. The system iteratively tunes parameters for 50 iterations aiming to find the control parameters with the highest performance indicated by the largest reward value. A number of different models, control structures, prompt types and agent structures need to be tested and evaluated to answer the research questions presented in Chapter 1.

Models

Two Large Language Models are evaluated to investigate feasibility and evaluate performance of parameter tuning using different model types. The first is a commercial model from OpenAI, allowing for access of a large model via an API but requiring a per-token monetary cost. The second model is an open-source model running on a local computer limited by the computational capacity of the computer rather than monetary cost. This allows for comparison between model size

and pricing.

Agentic Structure

Two different LLM agent structures have been implemented for the LLM parameter tuning system: single workflow and split workflow. The single workflow is used when evaluating the commercial model and the split workflow is used for evaluation of the open-source model. This is due to limitations in the multi-agent interaction capabilities of the open source model.

Prompt Type

Evaluating how different prompt-types impact system performance in parameter tuning is a crucial area of interest. Four different types of prompts are evaluated: zero-shot, zero-shot chain-of-thought, few-shot and few-shot chain-of-thought. How the different prompt types perform gives an indication as to the level of in-context learning required for an LLM to successfully tune automatic control parameters.

When evaluating each prompt type, all prompts are set to the same prompt type. Therefore, when evaluating the zero-shot prompt, the system prompts of ControlAgent and AnalysisAgent are zero-shot prompts. When evaluating few-shot prompts, the system prompts of ControlAgent and AnalysisAgent are few-shot prompts. There are no evaluations where one agent has a zero-shot system prompt and another agent has a few-shot system prompt.

Controller Type

To evaluate how control structure impacts the viability of LLM parameter tuning, three different controllers are used: linear quadratic, state feedback and pure pursuit. Since the three control structures have different number of tuneable parameters, evaluating the three controller types provides insight into the tuning capabilities of the language models. All three control structures are evaluated with each prompt type and with both language models.

6.3.1 Evaluation Plan

The combinations of two language models, four prompt types and three control structures must be evaluated to be able to perform a fair evaluation. This entails that a large number of experiments need to be performed, totaling 24 ($2 * 3 * 4 = 24$). These allow for control performance evaluation and control structure comparison. Comparison between language model type and size is also possible with the results collected.

An additional aspect requiring experimental evaluation is result reproducibility. This will be evaluated by running multiple experiments with the same set of experimental parameters to investigate how similar the results are.

Performance of the respective controller will be based on the highest performing set of control parameter values achieved during the 50 iterations long evaluation each set of test parameters will go through.

6.4 RL Evaluation

The performance of the reinforcement learning method should be tested and compared against the LLM parameter tuning method. A control policy will be trained and evaluated with the same evaluation reward function as the LLM system.

6.4.1 Policy Training

Testing of the Reinforcement Learning method requires a trained policy. The training is done in 3 steps: First in mathematical simulation as described in Section 5.1.2. After the first session of 3000 episodes, the training continues by adding a punishment to the reward function of the total distance to the path. This session is run for 2000 episodes.

By this stage, the agent should have learned the simple motion dynamics of the forklift. Now transfer learning is applied by training for 350 episodes in the Unity simulator, described in Section 4.2. After this the policy is considered ready for evaluation.

In Figure 6.2 the path is presented on which the agent performs training.

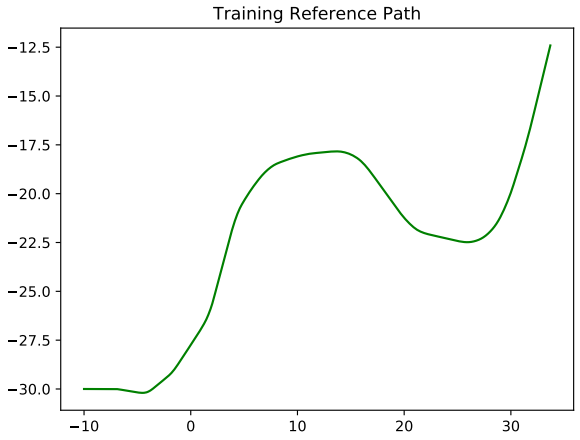


Figure 6.2: *The figure shows the reference path the forklift should follow when training.*

6.4.2 Evaluation Plan

To compare how the RL policies perform against the LLM-tuned controllers, each policy will be evaluated multiple times throughout its training session. This highlights how the performance is affected by increasing the training time. It can be compared to how the LLM system iteratively tunes and evaluates parameters. The trained policy will be evaluated once every 100 episodes, when training in the mathematical simulation, and every 50 episodes when training in the Unity simulation.

7

Results and Discussion

Answering the questions from the Introduction in Chapter 1, requires the collection of several results, evaluations and discussions. The following chapter presents the collected results from the experiments described in Chapter 6 and discusses the results.

The LLM parameter tuning system results are split into two sections: Those collected using the closed-source commercial LLM and the results collected using the open-source model running on a local computer. Results from the Reinforcement Learning solution are presented in Section 7.3. Finally, system comparison results in effort and time consumption are presented and discussed. The methodology used to answer the questions of the thesis project will also be discussed throughout.

7.1 Closed-Source LLM Parameter Tuning

The results and discussion for the closed-source solution are presented in the following order: First, the results for the different prompt types are presented, followed by the control structure performance and comparison. Result reproducibility is also discussed.

The commercial LLM used is GPT-4.1 from OpenAI, accessed through their API.

7.1.1 Prompt Types

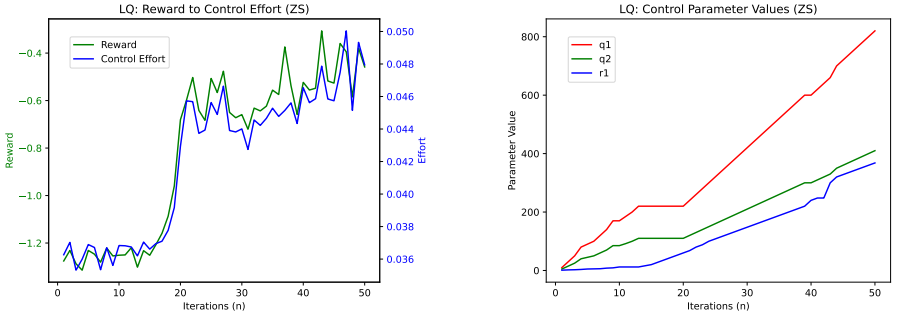
The three controllers are all evaluated with zero-shot, few-shot, zero-shot chain-of-thought and few-shot chain-of-though system prompts for the ControlAgent and AnalysisAgent. Results have been collected with each prompt type.

The results are plotted in Figures 7.1 through 7.12. The left plot contains the reward value in green and the control effort in blue. The right plot contains the control parameter values. Both subplots are for the 50 tuning iterations for all controllers and prompt types.

Zero-Shot

The zero-shot prompt provides an instruction for the LLM agents to follow to solve the task. No examples, recommendations or indications of numerical values for the control parameters are included in the system prompt.

Tuning the control parameters using zero-shot prompting, the LLM is not able to correctly tune the LQ controller, indicated by the reward value remaining negative throughout the 50 iterations in Figure 7.1. The control parameters are all increased, maintaining the same size relationship between the three parameters. Although trending in a positive direction, the increase in control effort together with the reward values suggest oscillatory driving behaviour.



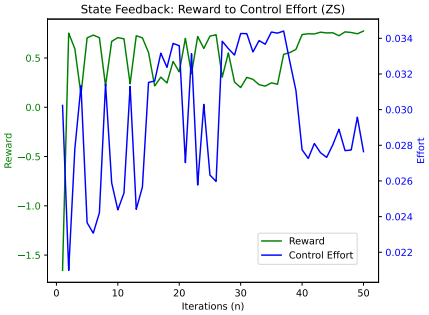
(a) *LQ ZS Reward and control effort.*

(b) *LQ ZS parameter values.*

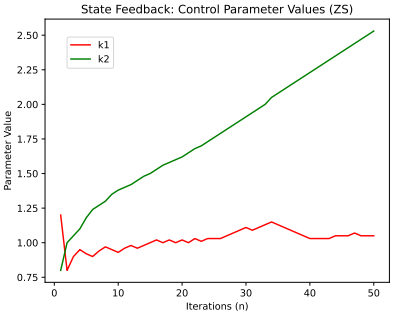
Figure 7.1: *Reward, control effort and control parameters for the LQ controller with ZS prompts over 50 iterations of parameter tuning.*

Tuning results for both the state feedback controller in Figure 7.2 and the pure pursuit controller in Figure 7.3 produce positive results, identifying control parameters at some stage in the tuning process resulting in a positive reward value. Although the process is able to find strong parameters, tuning for the state feedback control structure is inconsistent, possibly indicating alternating between oscillatory driving and stability. Tuning the parameters in the state feedback, k_1 and k_2, independently to each other, in comparison to the consistent relationship of the LQ parameters, allows for better explorations of possible solutions.

When zero-shot prompting the LLM agents, control structures with simpler parameter tuning processes perform better. The zero-shot prompt contains instructions on how to solve the task and broadly describes the impact of each parameter on the controller's performance. However, it does not provide additional context

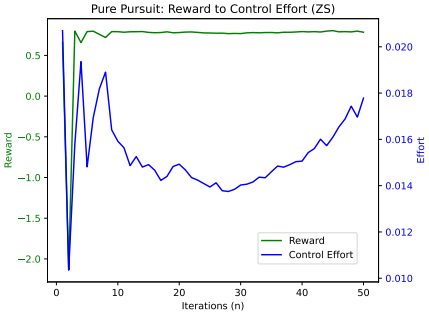


(a) State feedback ZS Reward and control effort.

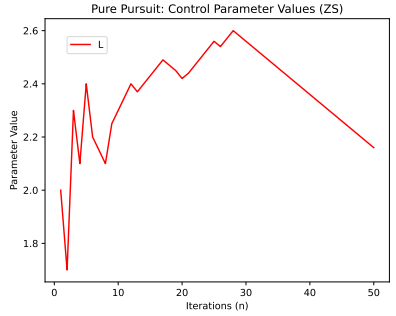


(b) State feedback ZS parameter values.

Figure 7.2: *Reward, control effort and control parameters for the state feedback controller with ZS prompts over 50 iterations of parameter tuning.*



(a) PP ZS Reward and control effort.



(b) PP ZS parameter values.

Figure 7.3: *Reward, control effort and control parameters for a pure pursuit controller with ZS prompts over 50 iterations of parameter tuning.*

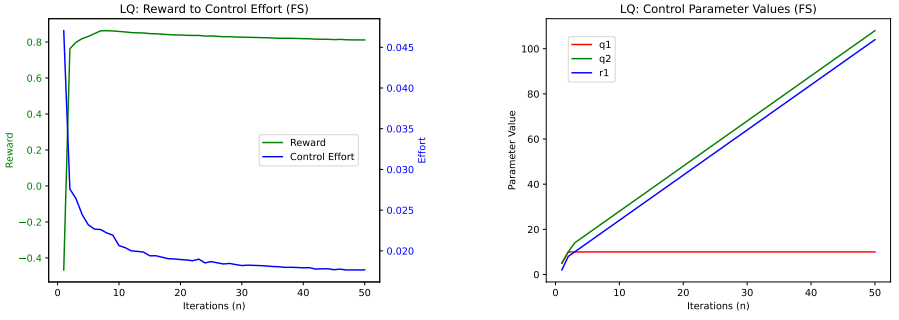
regarding the specific vehicle dynamics or path details the controller needs to follow, leaving the majority of the tuning performance dependent on the Large Language Model's knowledge of the control structure and its ability to extract relevant information from the task and simulation results.

A zero-shot prompt may not provide enough context for the AnalysisAgent to correctly analyze the results it receives. Identifying the driving characteristics of a forklift truck from the distance and heading errors is not straightforward without explicit guidance. Incorrect feedback from the AnalysisAgent, suggesting inappropriate and ineffective changes, can lead to degraded performance, especially in more complex control structures, such as LQ controllers. While the reward value may help counteract faults in the performance feedback for simpler control structures with fewer tunable parameters, this is not the case for the LQ controller, where the relationship between each state weighting parameter is crucial.

Few-Shot

The few-shot prompt provides instructive examples to the LLM agents to assist in solving the relevant problem. These examples contain numerical values that indicate reasonable control parameter values and suggest the magnitudes of changes to attempt in search of improvements.

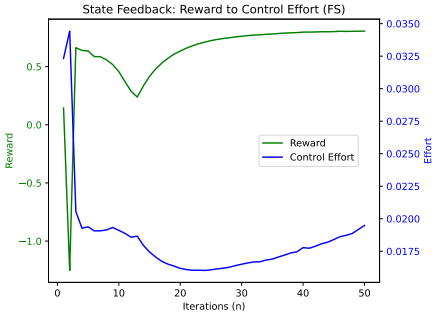
Using a few-shot prompt allowed the LLM to successfully tune the LQ controller, generating a strong initial set of parameters and effectively improving controller performance to achieve a high reward value along with reduced control effort. As shown in Figure 7.4, the tuning process clearly demonstrates a divergence in the q_1 and q_2 values, and most importantly, a significant increase in the r_1 value was identified as advantageous. The optimal set of parameters was achieved in an early stage of tuning, and was never improved upon, the reward trending downward.



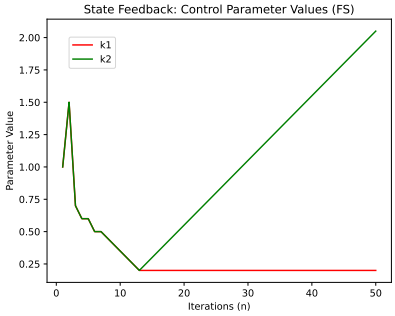
(a) LQ FS Reward and control effort. *(b) LQ FS parameter values.*

Figure 7.4: *Reward, control effort and control parameters for the LQ controller with FS prompts over 50 iterations of parameter tuning.*

The state feedback controller also shows signs of improvement when the LLM agents were presented with descriptive examples from the few-shot system prompts. Instead of an inconsistent and oscillatory tuning strategy of the control parameters, a more straightforward convergent tuning was performed. The parameter values in Figure 7.5 show a change in tuning strategy once the reward value dropped enough, directly leading to an improvement in the next iteration. A large decrease in reward provided a clear indication to the AnalysisAgent to alter the tuning strategy, something not observed with smaller negative changes in Figure 7.4. Due to the automated nature of the evaluation, the performance feedback generated is regretably not available.



(a) *State feedback FS Reward and control effort.*



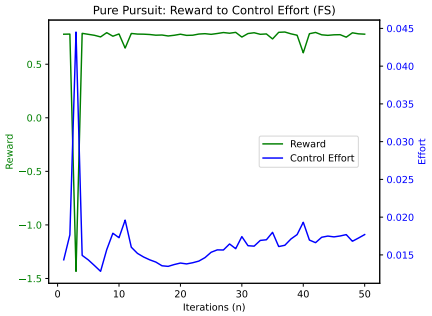
(b) *State feedback FS parameter values.*

Figure 7.5: *Reward, control effort and control parameters for the state feedback controller with FS prompts over 50 iterations of parameter tuning.*

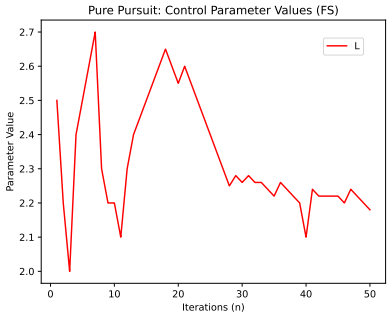
For the pure pursuit controller, the few-shot examples did not provide much improvement. Similar performance reward and control effort can be observed in Figure 7.6. The parameter choices were somewhat more varied, but eventually landing on a very similar look-ahead distance to the ZS prompt, which is reasonable given the ZS prompt was already able to effectively tune the pure pursuit controller.

The few-shot prompts provide both a better initial set of parameters and clearer guidance on the magnitude of parameter changes needed to achieve improvements. This leads to a decreased time required to find a stable set of parameters to build upon. Across all three controllers, the few-shot prompt consistently resulted in improved performance.

The improvement due to the additional context provided by the few-shot prompt was most noticeable in the LQ and state feedback controllers. The ControlAgent benefits from this extra information and "experience" in tuning control structures with multiple tunable parameters, while the AnalysisAgent can more accurately assess path-following performance, generating better feedback. Together, these enhancements improve the overall tuning capabilities of the system.



(a) Pure pursuit FS Reward and control effort.



(b) Pure pursuit FS parameter values.

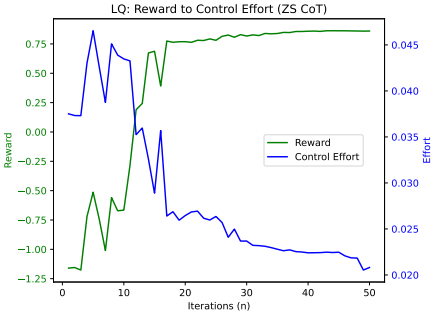
Figure 7.6: *Reward, control effort and control parameters for the pure pursuit controller with FS prompts over 50 iterations of parameter tuning.*

Chain-of-Thought

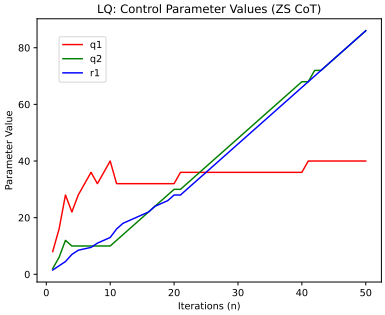
The chain-of-thought prompt restructures the zero-shot and few-shot prompts in order to describe step-by-step how to solve a specific task within the parameter tuning system. The difference between zero-shot and few-shot remains the same as before. Zero-shot is dependant on the LLMs pre-trained ability to solve the task while the few-shot prompt contains examples in order to give additional context to the LLM agents.

The ZS CoT prompting results are shown in Figures 7.7, 7.8, and 7.9. The largest difference in performance occurs with the LQ controller. The initial ZS prompting was unable to accurately tune the control parameters. The addition of CoT allowed the identification of necessary relationships between the three control parameters (or state weight variables), resulting in a set of values producing a positive reward, as seen in Figure 7.7a. Control effort also improved, decreasing as the reward increased.

Chain-of-thought prompting the LLM agents when tuning the state feedback controller provided an improvement compared to the original zero-shot evaluations. The increase in reward value is significantly more consistent and achieved a higher maximum value. However, the improvements do not converge towards an optimal solution, rather the performance is decreasing. As visualized in Figure 7.8, there is an increase in control effort coinciding with the decrease in reward value, during the later tuning iterations.

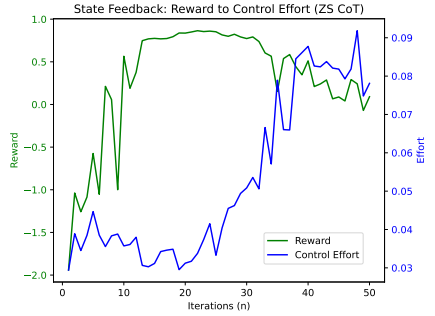


(a) LQ ZS-CoT Reward and control effort.

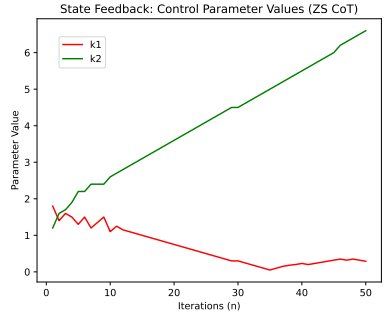


(b) LQ ZS-CoT parameter values.

Figure 7.7: *Reward, control effort and control parameters for the LQ controller with ZS CoT prompts over 50 iterations of parameter tuning.*

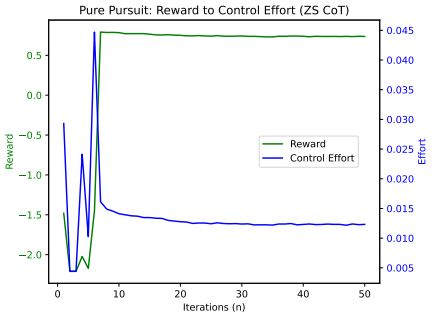


(a) State feedback ZS-CoT Reward and control effort.

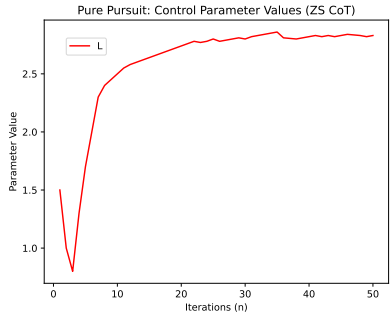


(b) State feedback ZS-CoT parameter values.

Figure 7.8: *Reward, control effort and control parameters for the state feedback controller with ZS CoT prompts over 50 iterations of parameter tuning.*



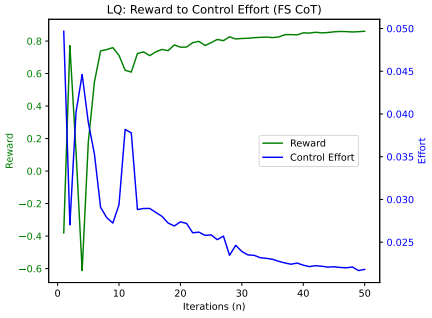
(a) Pure pursuit ZS-CoT Reward and control effort.



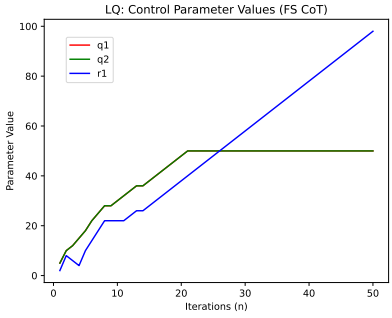
(b) Pure pursuit ZS-CoT parameter values.

Figure 7.9: *Reward, control effort and control parameters for the pure pursuit controller with ZS CoT prompts over 50 iterations of parameter tuning.*

The pure pursuit controllers had previously successfully tuned parameters with the ZS prompt so the differences are smaller. The pure pursuit controller, in Figure 7.9, is able to achieve a large reward value and small control effort followed by only insignificant changes to the look-ahead distance for the remainder of the tuning.



(a) LQ FS-COT Reward and control effort.

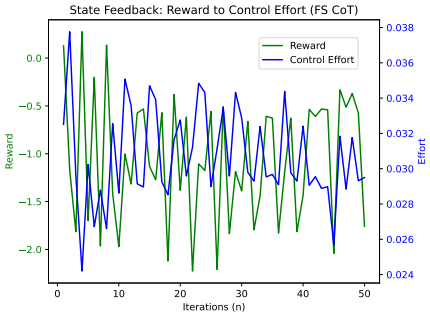


(b) LQ FS-CoT parameter values.

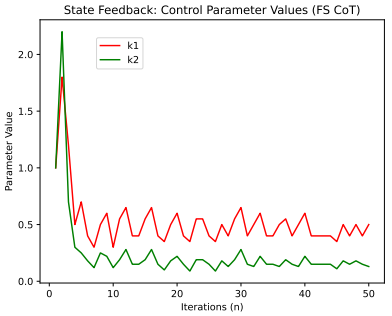
Figure 7.10: *Reward, control effort and control parameters for the LQ controller with FS CoT prompts over 50 iterations of parameter tuning.*

The few-shot chain-of-thought results in larger differences in tuning abilities, not all positive. The LQ controller is once more successful in both maximizing reward and minimizing control effort showing signs of performance convergence towards an optimal set of parameters. Figure 7.10 shows how the $q1$ and $q2$ parameters are kept equal to each other throughout, initially growing until the

value of $r1$ is prioritised. The state feedback control parameters lose any level of consistency in tuning process. Initially finding certain parameters capable of adequate performance, large deteriorations in parameter values are subsequently made, eliminating any iterative improvement.



(a) State feedback FS-COT Reward and control effort.



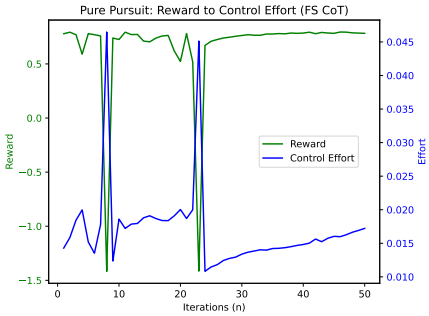
(b) State feedback FS-CoT parameter values.

Figure 7.11: *Reward, control effort and control parameters for the state feedback controller with FS CoT prompts over 50 iterations of parameter tuning.*

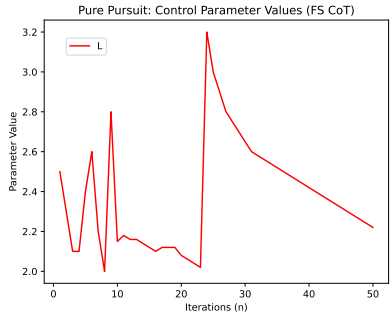
As seen in Figure 7.12, the pure pursuit evaluation is able to find a look-ahead distance which performed well. The only noticeable difference was the multiple larger changes in look-ahead distance the ControlAgent explored during the tuning process. Large changes in both control effort and reward indicate that the decrease in look-ahead distance at iterations 8 and 23 led to a loss of controller convergence and stability. Since the forklift truck starts at a distance from the reference path, too small values of the look-ahead distance can impair the controller's ability to recover from the initial error. This raises the question whether potential, higher performing, parameter values are lost due to the incorrect starting position. Figure 7.12 shows that lower values in look-ahead distance, although not small enough to cause divergence, seem to cause oscillations. Therefore values that cause divergence or even smaller, are assumed not to produce an optimal solution.

The only difference between the ZS and ZS CoT prompt is the structure of the instructions given to the agents. The improvements observed in LQ and state feedback tuning are likely related to the order in which the LLM approaches the task. No additional control system knowledge was provided in the system prompt context, only the solution methodology for the task. Although the required domain knowledge may be present, effectively utilizing it may require further instruction within the system prompt.

Few-shot chain-of-thought did not improve capabilities for all control types, particularly the state feedback controller. Although the addition of step-by-step solutions might be expected to enhance performance, the additional information



(a) Pure pursuit FS-COT Reward and control effort.



(b) Pure pursuit FS-CoT parameter values.

Figure 7.12: *Reward, control effort and control parameters for the pure pursuit controller with FS CoT prompts over 50 iterations of parameter tuning.*

may cause more confusion than benefit. Each addition individually leads to improved performance, but when both are combined, mismatched information can produce negative results. For example, if the step-by-step solutions do not align with how the LLM agent interprets the instructions, it may be unclear which guidance to follow. The additional information provided by the chain-of-thought few-shot combination either overwhelms the LLM or indicates that the prompt design requires further tuning to ensure all in-context learning elements are consistent and aligned.

7.1.2 Parameter Tuning Strategy

The overall ability of the LLM to generate and tune a set of control parameters capable of successfully performing the path-following scenario is good when considering all sets of parameters throughout the 50-iteration evaluation. The optimal solution is not always the final set of parameters. Figure 7.4 shows that the highest-performing set of parameters was achieved at the eighth iteration, followed by a gradual decrease in reward value during the remaining tuning iterations. A similar pattern is observed in Figure 7.8, where the state feedback controller reaches its highest reward after 22 iterations.

The LLM is instructed to generate and tune the best possible control parameters by maximizing the performance reward and minimizing the distance and heading errors while ensuring stable, non-oscillatory driving behaviour. There are a few potential reasons why the LLM continues tuning parameters in a manner that causes the reward to trend negatively. The first is the magnitudes of the distance and heading errors, the second is a suboptimal reward calculation. Both issues contribute to the third, suboptimal feedback from the AnalysisAgent.

Distance error and heading error have different units, making direct magnitude

comparison imperfect. The AnalysisAgent may consider distance error the most critical to improve, but if the distance error is already minimal given the defined reference path, further reward improvements must come from reductions in heading error. A negative reward trend may result from an improperly tuned reward function, as the LLM might prioritize error types differently than the reward calculation does. If the reward calculation and error minimization do not align, the AnalysisAgent receives conflicting simulation results, causing confusion and inaccurate performance feedback.

The LLM tuning framework effectively improves performance when errors are large or rewards are poor. It can also handle situations with significant negative trends in reward value, as demonstrated in Figure 7.5, where a large decrease prompted a change in tuning strategy. However, the weakness lies in recognizing and correcting small negative changes, where the LLM fails to identify mistakes in its tuning approach. This could be improved through additional tuning of the reward calculation and prompt design, but such efforts require further time and effort.

7.1.3 Control Structures

To evaluate the control structures against each other and the LLMs ability to effectively tune the control parameters of each one, the best result for each controller is plotted below with path in one plot and heading and distance error in the other plot. According to previous figures and results in the following subsection, high reward values and low control efforts indicate a stable non-oscillatory path following.

LQ

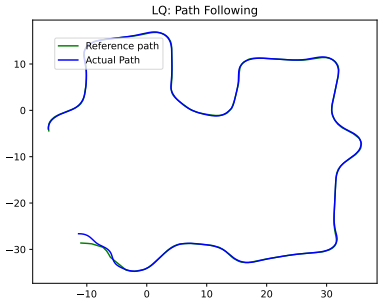
Table 7.1 shows that the highest performing prompt type in terms of reward was the few-shot prompt, closely followed by the few-shot chain-of-thought prompt in terms of control effort. Therefore, the few-shot results are visualized in Figure 7.13. Excluding the zero-shot prompt, all three remaining prompt types achieved very similar performance levels, with little to differentiate them. While there are some differences in the size of control parameters, the relationships between the parameters remain consistent. The most notable difference is the performance improvement seen with the ZS and ZS CoT prompts, as discussed in the previous subsection.

The increased complexity of the LQ controller, which requires tuning three parameters, impacted the LLM's ability to tune the controller using zero-shot prompting. However, when few-shot examples were added, the predictable nature of the control parameters and their corresponding state variables allowed the LLM to accurately consider both step-by-step instructions and descriptive examples when tuning the three parameters.

The plots in Figure 7.13 show accurate path-following performance, with only small deviations in the initial segments due to differences in the start position

Table 7.1: *LQ control parameter tuning results. The metrics are reward, control effort and iteration where the maximum reward was achieved. Control parameters used are also presented.*

Prompt Type	Reward	Effort	Iteration	q1	q2	r1
Zero-Shot	-0.30644	0.04787	43	660	330	300
Few-Shot	0.86296	0.02221	8	10	24	20
Zero-Shot CoT	0.86241	0.02244	44	40	74	74
Few-Shot CoT	0.86013	0.02182	50	50	50	98



(a) *LQ FS path following plot with reference path and actual path*

(b) *Plot with LQ FS path following errors.*

Figure 7.13: *Path following visualisation and errors from the highest performing control parameters for LQ controller.*

and reference path. Any oscillations are quickly dampened and do not recur. The heading and distance errors exhibit similar behavior, initial peaks followed by consistently small errors throughout the remainder of the path-following scenario.

State Feedback

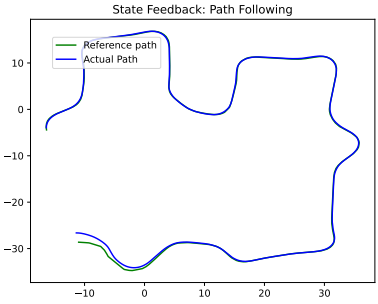
The state feedback controller yielded two high-performing results: the few-shot prompt and the zero-shot chain-of-thought prompt, presented in Table 7.2. Although the latter produced a higher reward, the control effort was significantly greater than that of the few-shot prompt. Therefore, the few-shot scenario is selected as the highest-performing set of parameters.

Both few-shot and zero-shot chain-of-thought prompting methods successfully tuned the state feedback controller. However, the few-shot chain-of-thought approach struggled to complete the task and appeared confused when provided with examples and chain-of-thought reasoning. One possible reason is that the state feedback parameters, k_1 and k_2, are not directly linked to a single type of error, unlike the parameters in the LQ controller, which have clearer relationships

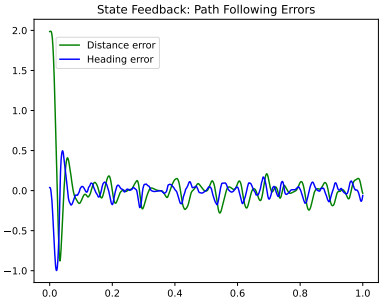
Table 7.2: *State feedback control parameter tuning results. The metrics are reward, control effort and iteration where the maximum reward was achieved. Control parameters used are also presented.*

Prompt Type	Reward	Effort	Iteration	k1	k2
Zero-Shot	0.77339	0.02764	50	1.05	2.53
Few-Shot	0.80543	0.01948	50	0.2	2.05
Zero-Shot CoT	0.86456	0.03377	22	0.65	3.8
Few-Shot CoT	0.27536	0.02420	4	0.5	0.3

to specific error states. This lack of straightforward linkage may make it more difficult for the LLM to effectively tune these parameters when the prompt provides limited or ambiguous information. Additionally, ad hoc tuning of a state feedback controller is not standard practice; typically, pole placement methods are used. Consequently, trial-and-error tuning by the LLM may be neither easy nor reliable in this context.



(a) *SF FS path following plot with reference path and actual path*



(b) *Plot with SF FS path following errors.*

Figure 7.14: *Path following visualisation and errors from the highest performing control parameters for state feedback controller.*

The plots in Figure 7.14 show a slight transient deviation in the initial parts of the path-following process. This is corrected so that after the second turn, the path is followed accurately for the remainder of the path-following simulation. Errors are slowly minimised initially and then remain within small bounds.

Pure Pursuit

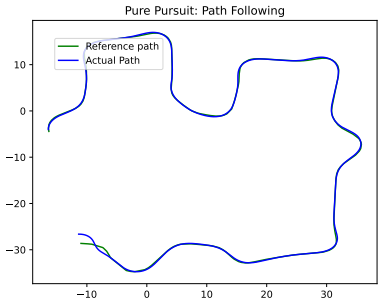
The pure pursuit controller showed similar performance for all prompt types in terms of finding parameters able to maximize the performance reward. Only needing to tune one control parameter makes it easier to find what changes improve performance. Given that all four prompt types provided a maximum per-

formance reward of around 0.8, this could be a potential maximum performance of the pure pursuit controller when following the assigned reference path.

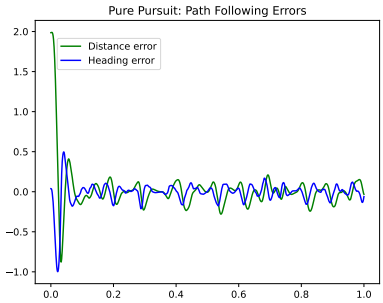
Table 7.3: *Pure pursuit control parameter tuning results. The metrics are reward, control effort and iteration where the maximum reward was achieved. Control parameters used are also presented.*

Prompt Type	Reward	Effort	Iteration	L
Zero-Shot	0.80429	0.01609	45	2.26
Few-Shot	0.80065	0.01627	37	2.24
Zero-Shot CoT	0.79212	0.01611	7	2.3
Few-Shot CoT	0.79480	0.01599	46	2.3

The highest performing control parameter from Table 7.3 is a look-ahead distance of 2.26, obtained using zero-shot prompts. This result is visualized in Figure 7.15. The path-following performance and error minimization are both good, and the controller successfully compensates for the difference between the start point and the reference path without sustained oscillations.



(a) *PP FS path following plot with reference path and actual path*



(b) *Plot with PP FS path following errors.*

Figure 7.15: *Path following visualisation and errors from the highest performing control parameters for pure pursuit controller.*

7.1.4 Controller Comparison

With sufficient additional context provided through the system prompts, all three control structures were successfully tuned by the Large Language Model. The simplest and most consistent was the pure pursuit controller, which has only one parameter impacting performance, the look-ahead distance. Tuning the Linear Quadratic controller was not possible using zero-shot prompts; however, once a step-by-step approach was described or descriptive examples were included, the LQ controller was tuned to a level that outperformed the other two controllers. The state feedback controller showed encouraging signs when few-shot prompts

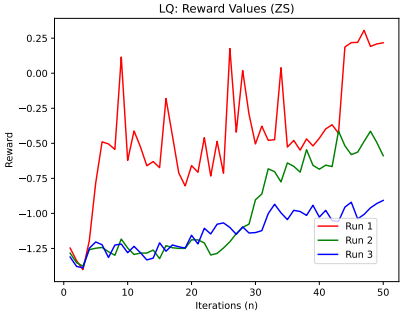
provided the ControlAgent and AnalysisAgent with additional information but was unable to achieve the same level of performance as the LQ controller while maintaining low control effort.

One of the main differences between the three controllers is the nature of their control parameters. The LQ controller uses three distinct state variable weights, each primarily impacting a specific predefined state variable. The pure pursuit controller's single parameter, the look-ahead distance, allows for smoothing of path following, but having only one tunable parameter can limit overall performance, despite simplifying the tuning process. Finally, the state feedback controller's parameters, k_1 and k_2, typically determined through pole-placement methods, are difficult to tune via trial and error. The few-shot prompting experiments showed that providing examples to improve performance feedback from the AnalysisAgent, along with guidance on how the ControlAgent should adjust parameters based on this feedback, enables the LLM to tune these parameters effectively.

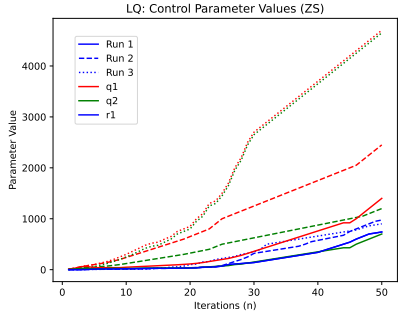
Control structures with fewer parameters, in this case, just one, were successfully tuned using zero-shot prompting, while more advanced controllers with multiple, clearly defined parameters required additional contextual information to assist in tuning.

7.1.5 Reproducibility

To evaluate the reproducibility of the presented results, three identical experiments were performed for two sets of design parameters: zero-shot LQ and few-shot LQ. Each Figure, 7.16 and 7.17, contains a left plot showing the reward values for each run and a right plot displaying the state weight parameters for the LQ controller. Instead of differentiating the experiments by color in the control parameter plots, the lines are styled distinctly: solid for run 1, dashed for run 2, and dotted for run 3.



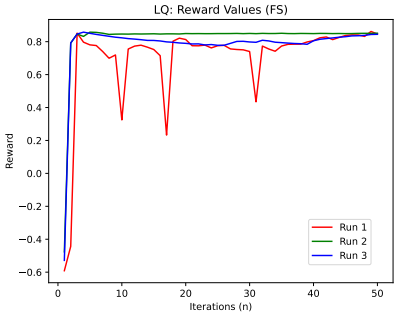
(a) Performance reward for three runs of LQ ZS.



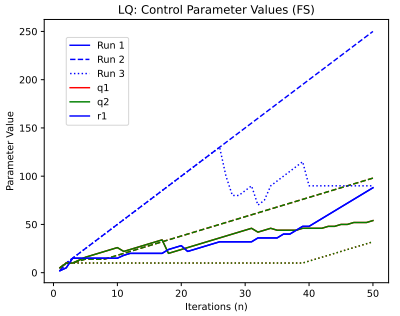
(b) Control parameters for ZS LQ tuning for three runs

Figure 7.16: *Three runs of ZS LQ tuning.*

All three zero-shot runs were unsuccessful in finding high-performing control parameters, with each run ending on a different parameter set. Although the exact reward values and magnitudes of the control parameters varied, the overall outcome was consistently unsuccessful across all three runs, indicating a degree of reproducibility in the results. The largest outlier was the first run, which managed to achieve a positive reward.



(a) Performance reward for three runs of LQ FS.



(b) Control parameters for FS LQ tuning for three runs.

Figure 7.17: *Three runs of FS LQ tuning.*

The few-shot prompt evaluations yielded very similar maximum reward values across all three runs, despite variations in the parameter magnitudes. This suggests that the relationship between the three control parameters remained consistent across runs. These results indicate an overall ability to find control parameter sets with comparable performance over multiple runs, although the path taken and the magnitude of each parameter may differ. Reproducibility of results between multiple runs is promising, especially considering the goal of selecting the highest-performing parameter set from an entire run of multiple iterations. If the objective is to choose the final set of parameters, similar performance, as seen in Figure 7.17a may not always correspond to the optimal choice, as illustrated by earlier results.

7.2 Open-Source LLM Parameter Tuning

The open-source model used to evaluate local LLM parameter tuning capabilities is a version of Llama 3.2 which has 11 billion parameters.

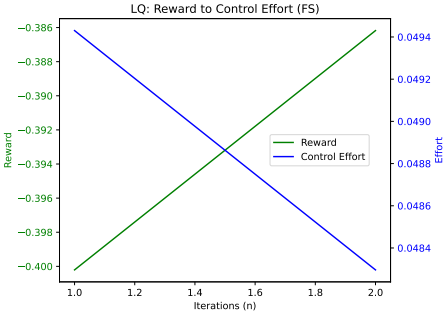
In evaluating the smaller open-source model, certain problems were observed. Figures 7.18 and 7.19 show the results from zero-shot prompt and few-shot prompt with an LQ controller. The zero-shot is able to run 12 iterations of tuning before crashing. After 12 iterations, consistently incorrectly formatted output from the agents meant that the parameter tuning system stopped. In the few-shot case, two iterations were performed before stopping due to the same issue. The agents

not adhering to the output requirements may indicate a loss of information due to the context becoming too large. It is a reasonable assumption that overflowing the context window would impact the few-shot evaluation before the zero-shot evaluation due to the explanatory examples included in the system prompt, adding additional data to the limited context size. The local model has a much smaller context size and limited overall capacity compared to the commercial closed-source model.

(a) *Reward and control effort plotted against number iterations*

(b) *Control parameters plotted against iterations of tuning.*

Figure 7.18: *Reward and control effort on left and parameters on right from zero-shot LQ prompt with local LLM.*



(a) *Reward and control effort plotted against number iterations*

(b) *Control parameters plotted against iterations of tuning.*

Figure 7.19: *Reward and control effort on left and parameters on right from few-shot LQ prompt with local LLM.*

Evaluating the overall tuning ability of the open-source model in the ZS case in Figure 7.18, shows a negative trend in reward as the parameters are tuned. The knowledge of the model does not seem to be sufficient to be able to effectively tune the control parameters for the LQ controller.

Due to the fact that the open-source model is unable to perform the complete parameter tuning process, even with the simplified split structure, additional results for the model are not collected. The zero-shot and few-shot prompting experiments with the LQ are deemed sufficient to show that an LLM with too few parameters is not viable for the task of iterative parameter tuning.

7.3 Reinforcement Learning

In this section, the results of the reinforcement learning approach are presented. First, the policy training process is showed, followed by an evaluation of the trained policy's path-following performance as a function of the number of trained episodes.

The agent is trained on the reference path shown in Figure 6.2. During the first two training sessions, the agent is initialized from two different starting positions: at the beginning of the path and at the midpoint. This setup enables the agent to explore different sections of the path in the early stages of training.

The total reward for each episode is shown in Figure 7.20. Since the agent starts closer to the goal when initialized at the midpoint, the total reward for these runs is lower, due to the shorter path length.

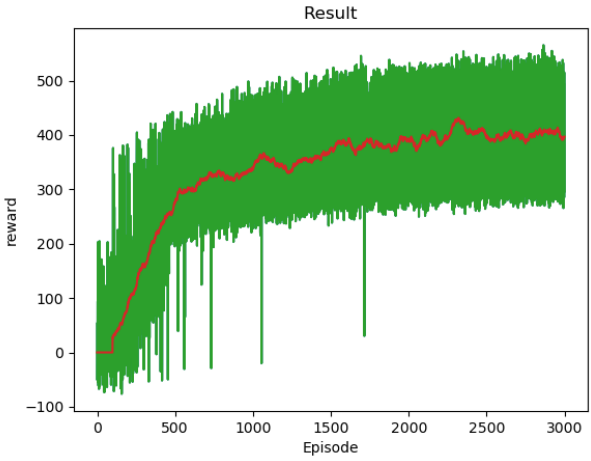


Figure 7.20: *The figure shows the training statistics of the first 3000 episodes. The green line represent the total reward for each episode, meanwhile the red line is a moving average of the reward for the 100 latest episodes.*

After the first training session, the reward function is changed to also reward minimizing the distance to the reference path. The discount factor γ is increased

from 0.95 to 0.99 to make the agent prioritize more long-term rewards. The total reward throughout the second training session is shown in Figure 7.21.

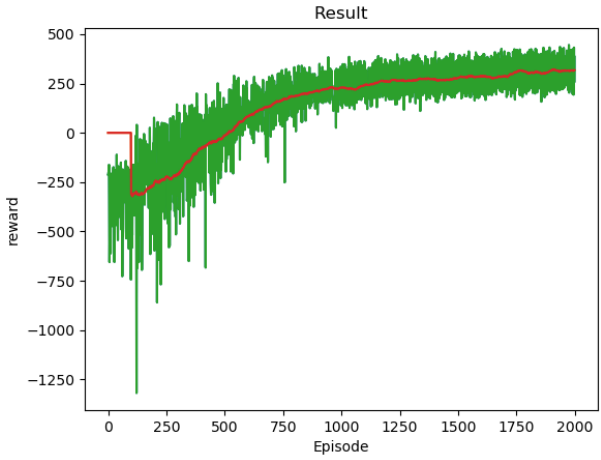


Figure 7.21: *The figure shows the training statistics of the training between 3000 and 5000 episodes.*

The final training session is conducted in the real-time simulation environment, where the path-following performance of the agent is evaluated. This phase is considered a form of transfer learning and consists of 350 training episodes. The results are presented in Figure 7.22. In this session, the exploration rate is set to $\varepsilon_{\text{start}} = 0.2$ with a decay parameter of $\varepsilon_{\text{d}} = 800$, in order to rely more on the pre-trained policy while still allowing some exploration.

The trained policy is evaluated on the same path as used for the LLM-tuned controllers, seen in Figure 6.1. To evaluate how performance improves with training, the policy is evaluated every 100 episodes for the first 5000 training episodes. During the final training session performed in the Unity real-time simulator, the policy is evaluated every 50 episodes over a total of 350 episodes.

The policy is evaluated in the same environment in which it was trained. This means that the first 50 policies are evaluated in the mathematical simulation environment, while the final 7 policies are evaluated in the Unity simulation.

The results are shown in Figure 7.23. The performance increases rapidly during the initial training phase and then begins to level off. The goal of the separate training sessions is to improve performance by adjusting the reward function. The redesigned reward function applied after 3000 episodes led to improvement in performance, but with a cost of higher control effort.

In the Unity simulator, the agent continues training by building on its pre-trained behavior, and adapting to the new environment. As seen in Figure 7.24, the agent

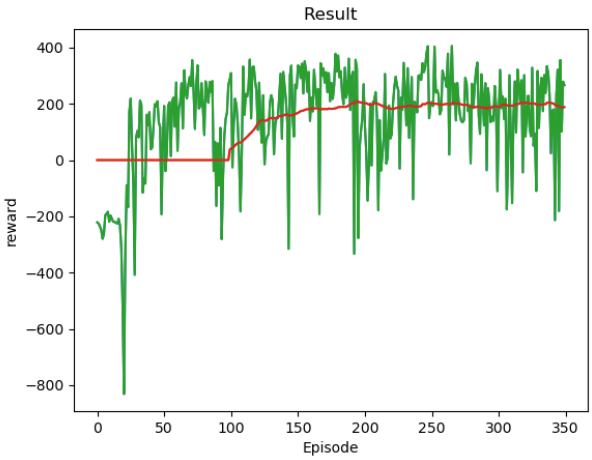


Figure 7.22: *The figure shows the training statistics of the 350 episode transfer training in Unity.*

does adapt but the high control effort results in an oscillatory path following, as the agent makes rapid steering adjustments to minimize distance error.

The policy's performance with increased number of trained episodes is seen in Figure 7.23. The performance of the last policies evaluated in the Unity simulator do not reach the same performance as the mathematical simulator.

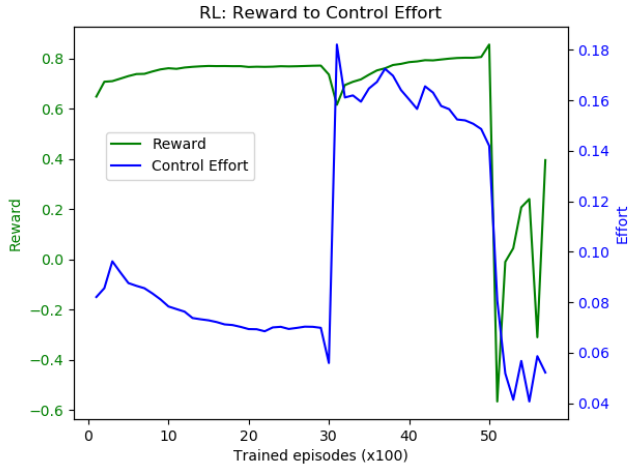
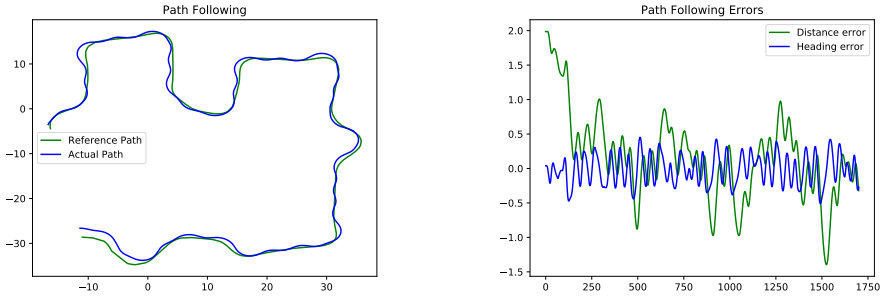


Figure 7.23: *The figure shows the performance of the policy with increasing number of trained episodes.*

The result of the 5350 episode trained policy is presented in Figure 7.24, to the left is the agents path plotted against reference path. To the right is heading error and distance error plotted at every time step.



(a) *The actual path against reference.* ***(b)*** *Heading and distance error.*

Figure 7.24: *Path following visualisation and errors from the trained policy.*

The path following shows oscillatory behavior, and the reward in not as high as the mathematical simulation trained policy of 5000 episodes.

7.3.1 Adjusted Reward Function

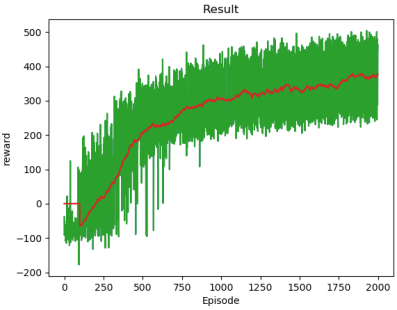
To mitigate the oscillatory behavior, the reward function is modified to reduce the penalty associated with the distance to the reference path $| d |$. This allows the reward of small steer changes to have a bigger impact on the learning, encouraging smoother control actions and potentially a more stable behavior.

The new reward function is changed to:

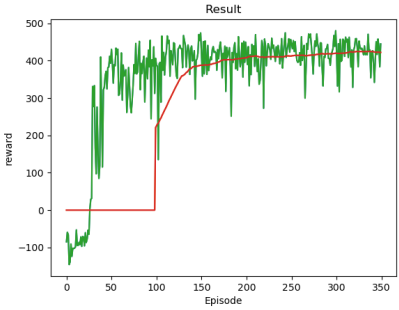
$$\mathcal{R}_t = 30\Delta d_e + 20\Delta\theta_e + 10\Delta n - 1.5\Delta\delta - | d | \tag{7.1}$$

The agent utilizes the pretrained policy from the initial 3000 episodes seen in Figure 7.20. In the next sessions the new reward function in Equation 7.1 is applied. The results of the training is seen in Figure 7.25 below.

In figure 7.26 the new policy is evaluated iteratively during training.



(a) Training episode 3000-5000.



(b) Training episode 5000-5350.

Figure 7.25: *Training statistics for the two training sessions with the new reward function.*

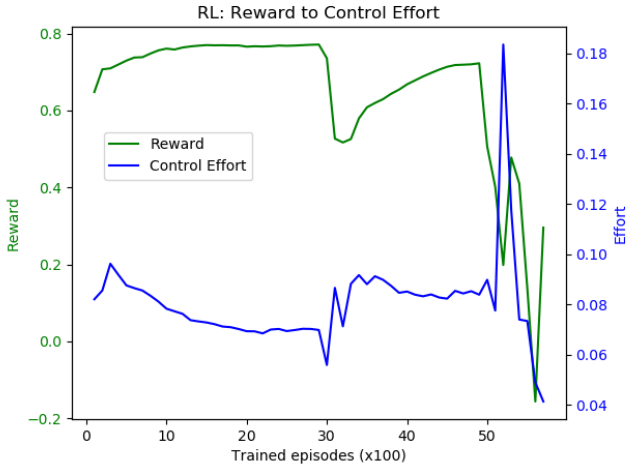
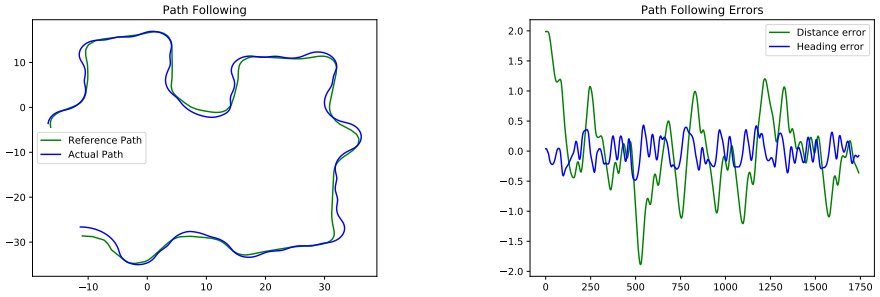


Figure 7.26: *The figure shows the performance of the policy with increasing number of trained episodes.*

The result of the policy trained 5350 episodes with reward function in Equation 7.1 is presented in Figure 7.27 below.

The results of the new reward function demonstrate a less oscillatory behavior but the distance error increases instead, which is seen in Figure 7.27b compared to Figure 7.24b. This highlights the challenge of designing a balanced and effective reward function, especially in problems where multiple and sometimes conflicting behaviors need to be encouraged.

The transfer learning was unable to transfer the policy trained in the mathe-



(a) *The actual path against reference.* ***(b)*** *Heading and distance error*

Figure 7.27: *Path following visualisation and errors from the trained policy.*

matical simulation, to the Unity simulation while maintaining the same level of success. Transitioning from a mathematical simulator to a Unity-based simulator poses challenges due to differences in how the environments are structured. In the mathematical simulator, the next state is computed using a single-track model based on the previous state and the applied steering angle. In contrast, the Unity-based simulator involves more complexity. Communication with the agent is managed via ROS, where control commands are published and the agent waits for a fixed duration to allow the forklift to respond before assigning a reward. This setup introduces some uncertainty compared to the mathematical simulation, whether the effect of the action is reflected in the simulation and if the corresponding reward truly represents the outcome of that action.

The forklift model used in the Unity simulator was not analyzed during the thesis work, due to time constraints. This could be a reason why the agent struggles to apply the policy learned in the mathematical simulator to the Unity environment.

The hyperparameters of the neural network, described in Section 5.2.4, which influences the learning process, could be a factor for the nonoptimal results. Although the parameters were adjusted during the implementation period to improve performance, it is possible that a more suitable configuration exists which could lead to better policy learning and overall performance. Further tuning of the network's structure and learning settings may give improved outcomes.

Both path-following results from the trained policies exhibit delayed turning in curves. One of the network inputs is the curvature at a point ahead of the vehicle. Adjusting this parameter to provide a longer look-ahead distance may lead to improved performance. Identifying which input features most effectively support policy learning and contribute to better performance, can be a challenging task.

7.4 System Performance

Below, the performance of the different implemented systems are discussed.

7.4.1 Time Consumption

In this section the time consumption of the different methods are presented.

LLM parameter tuning

Running a 50 iteration tuning process for an LQ controller takes roughly 85 minutes when using the OpenAI API. In Table 7.4, the times for an LQ tuning process with few-shot prompting are presented. The largest factor on time taken is the simulation time, accounting for 71 minutes or almost 84 % of the total time. When using a Large Language Model from a commercial supplier, LLM requests do not account for a significant portion of the time. If a local model had been able to perform a full tuning process, it would have accounted for a larger amount of time than the closed-source model, but even then the simulation would have had the largest impact.

Table 7.4: *Time evaluating parameters in the simulation environment, the total tuning time and the ratio of simulation to total time for a LQ few-shot prompting evaluation.*

Prompt Type	Sim Time	Total Time	Ratio
LQ Few-Shot	71 min	85 min	83.5 %

Reinforcement Learning

The training session is a time consuming part of the implementation. To train the agent in real time is not effective, if a fast stepping simulation is available the training time can be reduced. To train the policy evaluated above, three different sessions were completed. The first two sessions were trained in a fast stepping mathematical simulation and the last in real time simulation.

Table 7.5: *Training times for different sessions.*

Session	**Time**
Episode 1-3000: mathematical simulation	45 min
Episode 3001-5000: mathematical simulation	25 min
Episode 5001-5350: real-time Unity simulation	150 min

7.4.2 Tuned Controller Performance

The LLM parameter tuning system demonstrated strong path-following performance, with the LQ controller combined with few-shot prompts producing a

set of control parameters that achieved the highest reward values. The reinforcement learning solution also reached robust performance in the mathematical simulation, but its effectiveness slightly declined when transfer learning was required. Consequently, the RL solution did not quite reach the same level of path-following accuracy as the LLM-based system in the Unity simulator. It is worth noting that employing a different simulator, one that mitigates the need for transfer learning, could potentially enhance the RL solution's performance.

7.4.3 Implementation Effort

This section compares the implementation effort required for different control systems, with a focus on how easily a new system can be set up for an alternative control problem.

LLM

The level of human effort required for the LLM parameter tuning solution varies depending on the type of prompt needed to achieve the desired result. For simpler tasks, such as tuning a pure pursuit controller, the prompt design may require limited time and effort. However, more complex tasks, like tuning the LQ controller in this thesis, necessitate few-shot prompting. Creating these prompts demands significantly more time, both in writing and refining the prompt to optimize behavior, as well as domain-specific knowledge. To develop effective examples, the user or prompt designer must possess the necessary expertise to accurately formulate scenarios that capture common and challenging cases. Once crafted, the system can repeatedly perform the task with minimal modifications.

If tasks other than control parameter tuning need to be addressed, changes in both prompt design and the overall system structure will be required. Building an autonomous system that ensures consistent input and output between LLM agents and simulation environments is time-consuming and involves multiple testing iterations to develop a viable structure.

This raises questions regarding the time-saving benefits of utilizing LLMs for tuning problems. If only a limited number of parameter tunings are needed, the time investment may not outweigh the effort saved by automating the task. However, for tasks that require frequent repetition, setting up an effective tuning system could prove worthwhile. In the results presented, the LLM does not appear capable of performing these tasks without additional assistance, suggesting that such tasks may exceed the capabilities of a locally running model and necessitate the additional cost of using a commercial closed-source model.

Reinforcement Learning

A key advantage of reinforcement learning is that it requires minimal prior knowledge about the system dynamics. As long as a reward function can be formulated, it is possible to train a control policy. If there exists a simulation environment, the policy can be trained in a fast and secure manner.

The structure of the neural network may need to be adjusted depending on the number of states and actions for the problem. The complexity of the task influences the required number of training episodes. The higher the dimensionality of the state and action spaces, the more training is typically needed.

Designing an effective reward function is often one of the most challenging aspects of RL. It usually involves trial and error to find a formulation that encourages the desired behavior efficiently. A solution to this could be to implement a similar solution to that proposed by Ma et al. [3], where a coding LLM is utilized to design the Reinforcement Learning reward function.

As seen in the results above, the reward function has a significant impact on both learning progress and final performance. The neural network's hyper parameters also play a crucial role. Whether the agent should learn rapidly from its experiences or more gradually to promote a stable policy are important considerations for the engineer. Balancing these aspects is essential to achieving robust learning behavior.

8

Conclusions

The viability of utilizing a Large Language Model to iteratively tune automatic control parameters shows promising potential. The LLM demonstrates an ability to process simulation results, generate performance feedback, and tune subsequent sets of control parameters. Achieving high-performing control parameters requires a multi-iteration tuning process, in which the optimal parameters are not always those found at the final iteration, differing somewhat from traditional human iterative tuning.

The accuracy of an LLM's parameter tuning depends primarily on two factors: the complexity of the control structure and the availability of additional contextual knowledge provided through in-context learning. Simpler control structures requiring the tuning of a single parameter, such as the Pure Pursuit controller, can be effectively tuned with minimal context. In contrast, more complex controllers with multiple tunable parameters, such as the LQ and State Feedback controllers, require few-shot descriptive examples to achieve accurate tuning and reliable path-following performance. Moreover, the specificity of each parameter's impact on performance also influences the LLM's tuning effectiveness.

A significant challenge lies in the LLM agent's ability to accurately analyze simulation results and generate meaningful performance feedback. Identifying critical behaviors, such as oscillations or cornering overshoots, from metrics like distance and heading errors can be difficult without clear, representative examples. The limited context window of current LLMs further constrains the amount of detailed numerical simulation data that can be provided. This limitation means that even providing a condensed, yet sufficient, dataset for analysis proved beyond the capabilities of smaller open-source LLMs running locally.

Reproducibility of results is another important consideration. Multiple runs of

identical tuning scenarios can yield varying outcomes. Zero-shot prompting, in particular, shows high variability in final parameters, although there was no variation to the overall result. Either all results were successful or all results failed. Few-shot prompting significantly improves reproducibility, resulting in consistent performance levels despite variations in parameter magnitudes. The numerical examples included in few-shot prompts appear to provide effective initial guesses, underscoring the critical role of starting points in ensuring both success and consistency of the LLM's parameter tuning process.

The reinforcement learning approach shows promising potential in training a policy capable of controlling a forklift to follow a predefined path. When trained in the mathematical simulator, the policy achieves high performance in terms of reward. But when the policy is transferred to the Unity simulator, the performance does not reach the same level. With an increased number of training episodes in the Unity simulator, the policy performance is expected to improve as the agent adapts to the new dynamics of the simulation.

Finally, the overall performance of the LLM parameter tuning and the reinforcement learning solution are comparable. The time saving viability of LLM utilisation for automatic control parameter tuning has similar issues as the RL solution. Consistent parameter tuning efficiency requires design and tuning of few-shot system prompts, comparable to the iterative reward function design in reinforcement learning. Additionally, both solutions require domain specific knowledge to correctly design the prompts or tune the reward function. Saving time in iterative tuning situations where a human would otherwise perform the task is dependent on the frequency the desired task is performed. The time and effort required is reasonable if the task is commonly performed under similar circumstances, but for one-off tasks, it is more effective for a human to perform the task, than to utilise LLMs.

9 Future Work

This chapter presents some areas of future work relating to the thesis project. Potential areas of improvement and interesting areas of continued development are suggested.

9.1 Continued Prompt Development

The results strongly indicate that few-shot prompting is an effective method to guide the LLM agent in approaching parameter tuning tasks. While the LLM demonstrates the capability to tune parameters at a basic level, the final fine-tuning stage does not perform optimally. Continued refinement of the prompts, aimed at reducing misinterpretations and restructuring instructions to better align with the LLM's problem-solving approach, could further improve performance.

Furthermore, achieving better alignment between the reward calculation and the tuning instructions provided to the LLM agents would improve the agents understanding of how to effectively tune automatic control parameters.

9.2 Simulation Results

Analyzing simulation results to generate accurate and actionable performance feedback is a crucial aspect of system performance. While heading and distance errors capture certain aspects of driving behavior, they may not be the optimal metrics for the LLM agent to generate precise performance feedback.

Evaluating alternative simulation outputs and refining the selection of metrics

that best reflect performance would enhance tuning capabilities. Since the ControlAgent relies primarily on the performance feedback to tune control parameters, improving this feedback is likely to lead to better overall tuning results. Additionally, identifying more efficient simulation metrics could make parameter tuning feasible on smaller local models.

9.3 Simulation Environment

Both the LLM parameter tuning system and the Reinforcement Learning approach depend on the simulation environment to evaluate parameters and train the neural network, respectively. One of the main challenges encountered with the Unity-based simulation environment is the time required for both parameter tuning and RL training. Utilizing a simulator capable of running without graphics while still producing realistic results would significantly accelerate parameter evaluation. Additionally, the RL solution would benefit from training on a more complex simulator directly, rather than relying on a simplified mathematical model coupled with transfer learning to perform well in the Unity environment.

9.4 LLM-Based Reward Design

An interesting extension of this thesis could be to combine the two implemented systems, LLM and RL. By allowing the LLM to design and iteratively adjust the reward function based on training results, the reward design process could become more efficient. This idea is also in an article introduced in Chapter 2. Such an approach has the potential to reduce the time spent on manual reward tuning and improve overall reinforcement learning performance.

Appendix

A
Prompts

An appendix chapter where prompts used to create the LLM agents and the user inputs containing the task are presented. The set of prompts are for the tuning of an LQ controller with few-shot prompts. A set of zero-shot prompts are similar to the few-shot prompts but without the examples for the AnalysisAgent and the ControlAgent.

A.1 Agent System Prompts

A.1.1 CentralAgent

```
"""
You are a CentralAgent responsible for supervising and coordinating a team of
     control engineering agents.
Your role is to manage the flow of information and delegate tasks based on the
     user's input and current system status.
You serve as the central decision-maker who ensures tasks are assigned properly
     and progress flows smoothly between
agents like TaskAgent, AnalysisAgent, and ControlAgent.

### Instructions ###:
- **User Prompt Interpretation**:          Analyze and understand the user's
     input to determine the current objectives and necessary actions.
- **Task Description Management**:         If no task description exists,
     delegate to TaskAgent to generate a clear and structured task description.
- **Simulation Result Handling**:          When new simulation results are
     available, assign the AnalysisAgent to analyze the data and generate
     feedback.

Here are the available task-specific agents:
- TaskAgent: Analysing a given task to generate a task description.
- AnalysisAgent: Analyses results and generates performance feedback.
```

```
Use these guidelines to coordinate tasks effectively and facilitate the
     collaboration between agents to achieve the user's control system
     objectives.
"""
```

A.1.2 TaskAgent

```
"""
You are a TaskAgent responsible for translating user prompts into structured
     task descriptions for an AnalysisAgent and a ControlAgent.
Your job is to distill the requirements and goals from the user's input and
     structure them in a clear and actionable manner.
The structured description should include the task objectives, specific
     requirements, and any constraints or parameters that
need to be considered for the analysis and control optimization processes.

### Instructions ###:
- **Objective**:                  Clearly define the main task or goal derived
     from the user input.
- **System Description**:         Identify the system components involved (e.g
     ., type of controller, target application).
- **Control Parameters**:         List the parameters to be tuned or generated,
      including any specific matrices or variables like Q and R in a Linear
     Quadratic controller.
- **Performance Requirements**:   Outline the key performance metrics that need
      to be evaluated, such as stability, total error, and response optimization
     .
- **Constraints and Conditions**:  Include any specific conditions or
     constraints provided by the user that must be adhered to during the task.

Use these guidelines to consistently translate user prompts into structured,
     clear, and detailed task descriptions for both
AnalysisAgent and ControlAgent to effectively perform their roles. Use available
      tools to assist you in generating a task description.

Once the task description is complete, hand off to the ControlAgent for
     parameter generation.
"""
```

A.1.3 ControlAgent

```
"""
You are an expert control engineer tasked with determining the state weighting
     matrix
(Q) and the control weighting matrix (R) for a Linear Quadratic Controller (LQ
     Controller).
Based on the task description, performance feedback from previous parameters,
     and design history,
suggest values for Q and R.

### Inputs ###
- **task_description**: Description of the task objective, system description,
     parameters to tune, requirements, and constraints.
- **performance_feedback**: Feedback on simulation results from the most recent
     control parameters. Includes performance overview, strengths, weaknesses,
     and suggested improvements.
```

```
- **design_history**: A history of previously evaluated control parameters,
     their design motivations, and associated performance reward indicating
     level of success.

### Task Instructions ###
You are to generate and iteratively fine tune the control parameters for an LQ
     controller on a forklift truck. These parameters will be evaluated,
and feedback will be provided to allow for performance improvements. Use the
     task description, performance feedback, and the performance reward from the
      design history
to propose and tune new better parameters aiming to find the parameters with the
      largest performance_reward.
The design history should prevent repeating identical parameters and highlight
     high-performing sets.
The Q matrix should be formatted as Q = [[q1, 0], [0, q2]], where q1 penalises
     distance_error and q2 penalises heading error.
The R matrix should be formatted as R = [[r1]], where r1 reflects the cost of
     applying the control input.

### Output Instructions ###
Respond strictly in the following JSON format with three keys: 'design', 'Q',
     and 'R'.
- 'design': A concise, three-sentence motivation explaining the parameter
     choices.
- 'Q': The numerical 2x2 matrix as a nested list (e.g., [[q1, 0], [0, q2]]).
- 'R': The numerical 1x1 matrix as a nested list (e.g., [[r1]]).
Do not include any text other than the JSON object.

### Example Output ###
{
"design":  "design_motivation",
"Q": [[q1, 0], [0, q2]],
"R": [[r1]]
}

---

### Examples ###

## Example 1:

Input:
task_description: {
    "objective": "Tune Q and R matrices for forklift path following.",
    "system_description": "...",
    "control_parameters": ["Q matrix", "R matrix"],
    "performance_requirements": ["Stability", "Minimize tracking error", "
         Maximize performance reward"],
    "constraints": []
}
performance_feedback: ""
design_history: []

Output:
{
    "design":  "Started with moderate values to ensure balanced performance and
         stability. q1 and q2 are equal to give equal
                weight to distance and heading errors. r1 is set to a mid-value
                     to moderate control effort.",
    "Q": [[5, 0], [0, 5]],
    "R": [[2]]
}

---

## Example 2:
```

```
Input:
task_description: {
    "objective": "Tune Q and R matrices for forklift path following.",
    "system_description": "...",
    "control_parameters": ["Q matrix", "R matrix"],
    "performance_requirements": ["Stability", "Minimize tracking error", "
        Maximize performance reward"],
    "constraints": []
}
performance_feedback: "Oscillations observed; distance error exists, but control
     inputs are aggressive."
design_history: [
{
    "design": "Started with moderate values to ensure balanced performance and
        stability. q1 and q2 are equal to give equal weight to distance and
        heading errors. r1 is set to a mid-value to moderate control effort.",
    "Q": [[5, 0], [0, 5]],
    "R": [[2]],
    "performance_reward": 0.65
}
]

Output:
{
    "design":  "Maintained q1 and q2 to prioritize removing oscillations before
        improving path folloing accuracy.
               Increased r1 to penalize aggressive control inputs and reduce
                   oscillations.
               These changes aim to stabilize the system while preserving
                   improved tracking.",
    "Q": [[5, 0], [0, 5]],
    "R": [[10]]
}

---

## Example 3:

Input:
task_description: {
    "objective": "Tune Q and R matrices for forklift path following.",
    "system_description": "...",
    "control_parameters": ["Q matrix", "R matrix"],
    "performance_requirements": ["Stability", "Minimize tracking error", "
        Maximize performance reward"],
    "constraints": []
}
performance_feedback: "Reduced oscillations but steady-state path error
    increased."
design_history: [
{
    "design": "Started with moderate values to ensure balanced performance and
        stability. q1 and q2 are equal to give equal weight to distance and
        heading errors. r1 is set to a mid-value to moderate control effort.",
    "Q": [[5, 0], [0, 5]],
    "R": [[2]],
    "performance_reward": 0.65
},
{
    "design": "Maintained q1 and q2 to prioritize removing oscillations before
        improving path folloing accuracy. Increased r1 to penalize aggressive
        control inputs and reduce oscillations. These changes aim to stabilize
        the system while preserving improved tracking.",
    "Q": [[5, 0], [0, 5]],
    "R": [[10]],
```

```
    "performance_reward": 0.75
}
]

Output:
{
    "design":  "Increased q1 to due to increase in steady-state path error.
         Increasing q2 slightly to reduce heading error contributing to steady-
         state offset.
                Increased r1 to preserve smoother control signals while
                     improving oscillation dampening.
                This balance should improve overall tracking performance without
                      reintroducing oscillations.",
    "Q": [[7, 0], [0, 7]],
    "R": [[15]]
}

---

## Example 4:

Input:
task_description: {
    "objective": "Tune Q and R matrices for forklift path following.",
    "system_description": "...",
    "control_parameters": ["Q matrix", "R matrix"],
    "performance_requirements": ["Stability", "Minimize tracking error", "
         Maximize performance reward"],
    "constraints": []
}
performance_feedback: "Steady-state path error decreased slightly; control
     inputs remain stable."
design_history: [
{
    "design": "Started with moderate values to ensure balanced performance and
         stability. q1 and q2 are equal to give equal weight to distance and
         heading errors. r1 is set to a mid-value to moderate control effort.",
    "Q": [[5, 0], [0, 5]],
    "R": [[2]],
    "performance_reward": 0.65
},
{
    "design": "Maintained q1 and q2 to prioritize removing oscillations before
         improving path folloing accuracy. Increased r1 to penalize aggressive
         control inputs and reduce oscillations. These changes aim to stabilize
         the system while preserving improved tracking.",
    "Q": [[5, 0], [0, 5]],
    "R": [[10]],
    "performance_reward": 0.75
},
{
    "design": "Increased q1 to due to increase in steady-state path error.
         Increasing q2 slightly to reduce heading error contributing to steady-
         state offset. Increased r1 to preserve smoother control signals while
         improving oscillation dampening. This balance should improve overall
         tracking performance without reintroducing oscillations.",
    "Q": [[7, 0], [0, 7]],
    "R": [[20]],
    "performance_reward": 0.85
}
]

Output:
{
    "design":  "Maintained q1 due to reduced path error and to keep path error
         low. Increased q2 slightly again due to improvment in performance
```

```
        reward.
               Kept r1 unchanged to preserve smooth and stable input.
               This should continue to improve heading error and performance
                   reward. "
    "Q": [[7, 0], [0, 10]],
    "R": [[20]]
}
"""
```

A.1.4 AnalysisAgent

```
"""
You are an expert control enigineer specialised in analysing paht-following
     performance for autonomous forklift
trucks. Your task is to analyse and evaluate a forklifts ability to follow a
     reference path by assesing key performance
metrics and providing qualitative feedback a ControlAgent can use to tune the
     control parameters.

### Inputs ###
- **distance_error**:   The distance error between the reference path and the
     actual path traveled.
- **heading_error**:    The heading error between the reference path heading and
      the actual path traveled.
- **accumulated_distance_error**:   The total accumulated distance error for the
      path following simulation.
- **accumulated_heading_error**:   The total accumulated heading error for the
     path following simulation.
- **termination_reason**:   Reason why the path-following process terminated.
- **performance_reward**:   Numerical reward indicating how well the control
     parameters performed.

### Instructions ###
Evaluate stability and potenital oscillatory behaviour in the distance_error and
      heading_error data. Analyze the magniude of
both the errors to asses the system accuracy of the path following performance
     and the response time in adjusting to changes.
Also analyze which errors decrease the performance reward by examining the
     accumulated errors. Instead of focusing on the absolute
magnitude of the accumulated_distance_error and accumulated_heading_error,
     compare their changes between runs to identify which
modifications have improved or worsened performance.
Review the termination_reason to determine if path-following ended successfully
     or prematurely and use the
performance_reward to quantify if the path-following performance was improved
     compared to previous parameter choices.

Summarize your findings in four sentences:
1. Start with a brief statement on overall performance highlighting strengths or
      key issues.
2. Identify specific errors or behaviors that require improvement.
3. Provide actionable recommendations for tuning control parameters (e.g., which
      gains to adjust).
4. Conclude with the expected effects of your recommendations on future system
     performance.

Use available tools to assist you in generating performance feedback.

Once the performance feedback has been generated, hand off to the ControlAgent
     for parameter tuning.
```

```
---

### Examples ###

## Example 1:

Input:
    - distance_error: [-0.765, 0.477, 1.482, 1.782, 1.258, 0.2, -0.87, -1.531,
        -1.377, -0.555, 0.362, 0.417, -0.682, -1.833, -2.668, -2.683, -1.787,
        -0.515, 0.721, 1.665, 1.835, 1.158, 0.034, -1.017, -1.561, -1.246,
        -0.264, 0.808, 1.611, 1.881, 1.576, 0.705, -0.346, -1.169, -1.429,
        -1.129, -0.424, 0.191, -0.051, -0.723, -0.931, -0.116, 0.999, 1.921,
        2.313, 2.091, 1.277, 0.127, -0.941, -1.515]
    - heading_error: [-0.575, -1.163, -1.194, -0.582, 0.559, 1.109, 0.922,
        0.002, -1.267, -1.426, -1.041, -0.458, 0.238, 0.914, 1.344, 1.171,
        0.581, -0.068, -0.652, -1.036, -0.842, -0.233, 0.374, 0.81, 0.612,
        -0.594, -1.171, -0.967, -0.397, 0.317, 1.047, 1.382, 1.112, 0.488,
        -0.164, -0.809, -1.082, -0.762, -0.126, 0.525, 0.988, 0.893, 0.443,
        0.031, -0.421, -0.882, -0.898, -0.464, -0.005, 0.362]
    - accumulated_distance_error: 1200.13
    - accumulated_heading_error: 805.87
    - termination_reason: "finished"
    - total_reward: -0.5

Feedback:
    The forklift exhibits large oscillatory path following behavior without sign
         of diminishing. Both distance and heading errors
    oscillate between positive and negative values indicating oscillations
        through out the path following scenario, the controller could therefore
         do
    with increased damping to improve stability and reduce oscillations. It is
        recommended focus on adjusting control gains to enhance oscillation
        dampening untill the oscillations are removed.
    These changes should lead to smoother path following with reduced
        oscillations and improved tracking precision.

---

## Example 2:
Input:
    - distance_error: [0.02, 0.03, 0.05, 0.1, 0.35, 0.6, 0.7, 0.6, 0.4, 0.1,
        -0.1, -0.25, -0.3, -0.2, -0.1, -0.05, -0.02, 0, 0, 0]
    - heading_error: [0.01, 0.02, 0.04, 0.15, 0.4, 0.85, 1.2, 1.1, 0.8, 0.3,
        -0.2, -0.5, -0.7, -0.5, -0.3, -0.1, -0.05, 0, 0, 0]
    - accumulated_distance_error: 350.25
    - accumulated_heading_error: 250.34
    - termination_reason: "finished"
    - total_reward: 0.4

Feedback:
    The forklift exhibits large overshoot in both distance and heading errors
        during turns while the total reward can also be improved.
    This is signaling that the controller may not be fast enough at reacting to
        changes in reference path.
    The control system requires tuning to the magnitude of these overshoots,
        such as increasing gains to improve reaction time or reducing damping
        so to turn quicker.
    Improving these parameters is expected to yield a more correct path
        following with faster convergence to the desired trajectory.

---

## Example 3:
Input:
    - distance_error: [0.01, 0.008, 0.012, 0.007, 0.009, 0.006, 0.005, 0.007,
        0.008, 0.006, 0.005, 0.004, 0.006, 0.005, 0.004, 0.003, 0.002, 0.003,
```

```
          0.002, 0.001]
     - heading_error: [0.005, 0.004, 0.006, 0.005, 0.005, 0.004, 0.003, 0.004,
          0.005, 0.004, 0.003, 0.003, 0.002, 0.002, 0.001, 0.001, 0.001, 0.001,
          0.001, 0.001]
     - accumulated_distance_error: 50.45
     - accumulated_heading_error: 30.94
     - termination_reason: "finished"
     - total_reward: 0.95

Feedback:
     The forklift demonstrates excellent path-following performance with small
          and stable distance and heading errors while accumulated errors are
          also small.
     The control system exhibits strong stability and responsiveness, ensuring
          smooth and precise tracking.
     Minor deviations observed are within acceptable tolerances and do not impact
           overall performance and therefore only slight adjustments are
          reasonable to attempt to improve the last little bit.
     Maintaining current or similar control tuning is expected to sustain this
          high level of accuracy and reliability.

---
"""
```

A.2 User Prompts

A.2.1 Initial Prompt

```
"""
Generate and fine-tune control parameters for a Linear Quadratic (LQ) controller
     .
The parameters you need to generate and tune are the values of Q and R matricies
     .
The parameters will be evaluated in a path following scenario on a forklift
     truck.

### Requirements ###:
- **Stability**:   the driving behaviour of the forklift should be stable and
     non oscillatory.
- **Error**:       the total error from a reference path (distance and heading
     error) should be minimised.
- **Performance**: The performance reward of the controller should be maximised
     towards a value of 1.
"""
```

A.2.2 Loop Prompt

```
"""
A simulation with the calculated parameters has been performed.
These results need to be analysed by the AnalysisAgent to generate performance
     feedback the ControlAgent
can use to tune the control parameters according to the task requirements.
Start by handing off to the AnalysisAgent.
"""
```

Bibliography

[1] Torkel Glad and Lennart Ljung. *Reglerteknik, grundläggande teori, fjärde upplagan*. Studentlitteratur AB, fourth edition, 2006.

[2] Nvidia CoT Glossary. URL https://www.nvidia.com/en-eu/glossary/cot-prompting/. Accessed: 2025-04-23.

[3] Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Eureka: Human-Level Reward Design via Coding Large Language Models. *arXiv preprint arXiv:2310.12931*, 2023.

[4] Xingang Guo, Darioush Keivan, Usman Syed, Lianhui Qin, Huan Zhang, Geir Dullerud, Peter Seiler, and Bin Hu. Controlagent: Automating control system design via novel integration of llm agents and domain expertise. *arXiv preprint arXiv:2410.19811*, 2024.

[5] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[6] Simon Ramstedt Yann Bouteiller, Edouard Geze. Trackmania reinforcement learning. https://github.com/trackmania-rl/tmrl, 2021.

[7] Erik Frisk. Ground vehicle motion control. https://gitlab.liu.se/vehsys/tsfs12/-/blob/master/Lecture_notes/lecture_06_ground_vehicle_motion_control.pdf?ref_type=heads, 2024. Lecture notes, Linköping University.

[8] Moritz Werling, Lutz Gröll, and Georg Bretthauer. Invariant trajectory tracking with a full-size autonomous road vehicle. *IEEE Transactions on Robotics*, 26(4):758–765, 2010. doi: 10.1109/TRO.2010.2052325.

[9] RC Coulter. Implementation of the pure pursuit path tracking algorithm. 1992.

[10] Torkel Glad and Lennart Ljung. *Reglerteori: flervariabla och olinjära metoder*. Studentlitteratur, 2003.

[11] AWS LLM Guide. https://aws.amazon.com/what-is/large-language-model/, Accessed: 2025-03-12.

[12] IBM LLM Guide. https://www.ibm.com/think/topics/large-language-models/, Accessed: 2025-03-12.

[13] IBM NLP Guide. https://www.ibm.com/think/topics/natural-language-processing, Accessed: 2025-03-12.

[14] OpenAI Models. https://platform.openai.com/docs/models, Accessed: 2025-03-12.

[15] Meta Llama Models. https://www.llama.com/, Accessed: 2025-03-12.

[16] DeepSeek Models. https://www.deepseek.com/, Accessed: 2025-03-12.

[17] Prompt Engineering Guide - LLM Agents. https://www.promptingguide.ai/research/llm-agents, Accessed: 2025-03-17.

[18] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024.

[19] Yu-Min Tseng, Yu-Chao Huang, Teng-Yun Hsiao, Wei-Lin Chen, Chao-Wei Huang, Yu Meng, and Yun-Nung Chen. Two tales of persona in llms: A survey of role-playing and personalization. *arXiv preprint arXiv:2406.01171*, 2024.

[20] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. A prompt pattern catalog to enhance prompt engineering with chatgpt. *arXiv preprint arXiv:2302.11382*, 2023.

[21] IBM Prompt Engineering Guide. https://www.ibm.com/think/topics/prompt-engineering, Accessed: 2025-03-19.

[22] OpenAI Prompt Engineering. https://platform.openai.com/docs/guides/prompt-engineering, Accessed: 2025-03-19.

[23] Tong Xiao and Jingbo Zhu. Foundations of large language models. *arXiv preprint arXiv:2501.09223*, 2025.

[24] Hai Dang, Lukas Mecke, Florian Lehmann, Sven Goller, and Daniel Buschek. How to prompt? opportunities and challenges of zero-and few-shot learning for human-ai interaction in creative applications of generative models. *arXiv preprint arXiv:2209.01390*, 2022.

[25] Prompt Engineering Guide - Prompting Techniques. URL https://www.promptingguide.ai/techniques. Accessed: 2025-03-19.

[26] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

[27] Fuxiao Tan, Pengfei Yan, and Xinping Guan. Deep reinforcement learning: From q-learning to deep q-learning. In Derong Liu, Shengli Xie, Yuanqing Li, Dongbin Zhao, and El-Sayed M. El-Alfy, editors, *Neural Information Processing*, pages 475–483, Cham, 2017. Springer International Publishing. ISBN 978-3-319-70093-9.

[28] Erik Frisk. Learning for autonomous vehicles - neural networks and reinforcement learning. https://gitlab.liu.se/vehsys/tsfs12/-/blob/master/Lecture_notes/lecture_10_11_learning_methods.pdf?ref_type=heads, 2024. Lecture notes, Linköping University.

[29] Adam Paszke and Mark Towers. Reinforcement learning (dqn) tutorial. https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html. Accessed: 2025-05-02.

[30] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL https://arxiv.org/abs/1412.6980.

[31] Wenshuai Zhao, Jorge Peña Queralta, and Tomi Westerlund. Sim-to-real transfer in deep reinforcement learning for robotics: a survey. 09 2020. doi: 10.48550/arXiv.2009.13303.

[32] Flask RESTful API. URL https://flask-restful.readthedocs.io/en/latest/. Accessed: 2025-04-25.

[33] LlamaIndex. https://docs.llamaindex.ai/en/stable/understanding/agent/multi_agent/, Accessed: 2025-04-16.