Article

Not peer-reviewed version

Agentic AI For Real-Time Adaptive PID Control of Servo Motor

Tariq Mohammad Arif * and Md Adilur Rahim

Posted Date: 6 August 2025

doi: 10.20944/preprints202508.0355.v1

Keywords: artificial intelligence; large language model; agentic AI; Quanser Qube Servo2; Servo Control; PID

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 August 2025 doi:10.20944/preprints202508.0355.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Agentic AI For Real-Time Adaptive PID Control of Servo Motor

Tariq Mohammad Arif [1,*] and Md Adilur Rahim [2]

[1] Weber State University
[2] Louisiana State University
* Correspondence: tariqarif@weber.edu; Tel.: +1-801-626-6962

Abstract

This study explores a novel approach of using large language models (LLMs) in real-time Proportional-Integral-Derivative (PID) control of a physical system, the Quanser QUBE-Servo 2. We investigated whether LLMs, used with an Artificial Intelligence (AI) agent workflow platform, can participate in live tuning of PID parameters through natural language instructions. Two AI agents were developed: a control agent that monitors the system performance and decides if tuning is needed, and an optimizer agent that updates PID gains using either a guided system prompt or a self-directed free approach within a safe parameter range. The LLM integration was implemented through Python programming and Flask-based communication between the AI agents and the hardware system. Experimental results show that both tuning approaches effectively reduced standard error metrics, such as IAE, ISE, MSE, and RMSE. This study presents the first known real-time implementation of servo motor control powered by LLMs, and it has the potential to become a novel alternative to classical control or machine learning and reinforcement learning based control approaches. The results are promising for using agentic AI in heuristic-based tuning and control of complex physical systems.

Keywords: artificial intelligence; large language model; agentic AI; Quanser Qube Servo2; servo control; PID

1. Introduction

The integration of Artificial Intelligence (AI) into control systems has shown significant potential in recent years. At the same time, advancements in computer vision, deep learning, and reinforcement learning, along with the rise of Large Language Models (LLMs), have created innovative applications across various scientific and engineering fields [1,2]. Modern robotic and control systems have also evolved from completing set tasks in controlled environments to displaying intelligent behaviors by learning from image analysis, datasets, and real-time interactions [3,4]. Other AI-based work incorporates a multi-phase reinforcement learning for PID tuning framework and show how deterministic policy gradients and neural networks can enable dynamic self-tuning [5–7]. Evolutionary, neural network, and heuristic approaches were also used to enhance tuning by incorporating artificial learning [8,9]. In a very recent work, researchers used LLM agents/controllers, which guided adaptive compensator and dynamically adjusts robot control policies as needed with minimal manual intervention [10,11].

In this study, we used a Large Language Model (LLM) based AI agent to control a servo motor via adaptive PID tuning. While most work in this field is done on simulated robotic environments or abstracted dynamics, in this research, we deployed an LLM-based PID tuning in a real-time hardware setup. This hands-on approach addresses practical challenges such as real-time constraints, sensor noise, actuator delays, and safety limits.

To fine-tune the PID parameters of the system we embedded LLM agents within an automated workflow by n8n [12]. This method shows real-time decision-making and closed-loop adjustments of

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

LLM, connecting high-level AI reasoning and low-level motor control. To the best of current literature, this marks one of the first known instances of using LLMs to autonomously tune PID parameters for a physical system during real-time operation, not just generating control policies offline or in simulations. This research also validates LLM-based control and presents practical viability beyond off-line control, theoretical models, and digital twins.

2. Materials and Methods

2.1. Hardware

To implement a real-time Proportional-Integral-Derivative (PID) tuning using AI agents, a direct-drive DC motor with an integrated optical encoder is used. The hardware components needed to be able to respond to control inputs reliably and quickly. For this purpose, we have used the Quanser QUBE Servo 2 system, which integrates a DC motor mechanism with position feedback through an encoder [13]. This setup ensures precision and accuracy in real-time applications. The Quanser QUBE system can be operated with either a load disk or a pendulum attachment. In this experiment, we utilized the load disk with the rotary servo platform. Figure 1. shows a schematic diagram of the QUBE-Servo 2 setup with the inertia disk. Here, the DC motor shaft is connected to a load hub and disk, which together define the rotational inertia of the system.

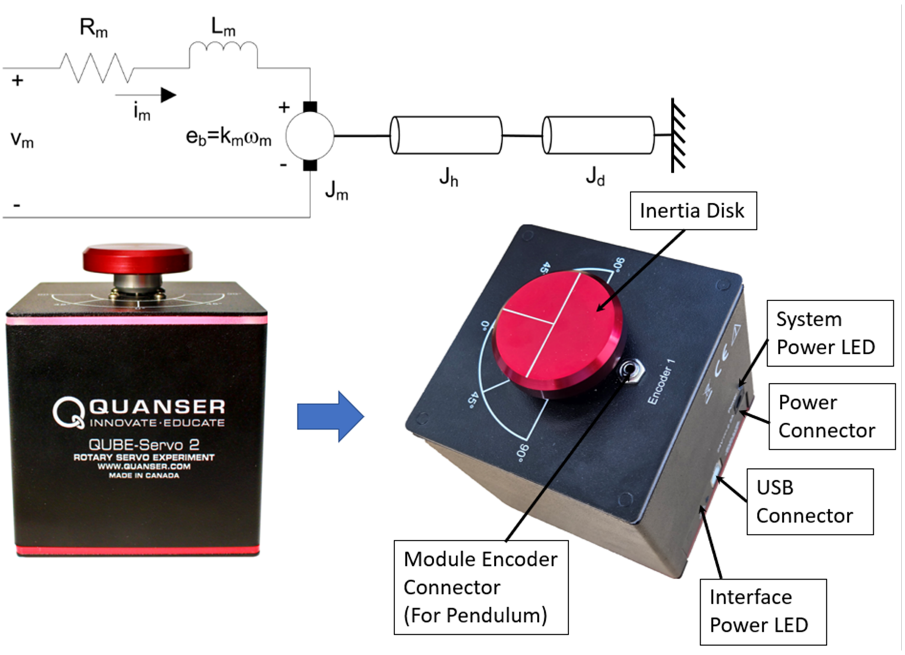

Figure 1. Schematic diagram of QUBE Servo 2 DC motor with inertia disk setup.

2.1.1. Electro-Mechanical Parameters

In Figure 1, R_m, k_t, k_m, J_m, L_m, m_h, r_h, J_h, m_d, and r_d represent the terminal resistance (8.4 Ω), torque constant (0.042 Nm/A), motor back electromotive force (emf) constant (0.04 V/(rad/s)), rotor inertia (4.0×10^{-6} kg $\cdot$ m^2), rotor inductance (1.16 mH), load hub mass (0.0106 kg), load hub radius (0.0111 m), load hub inertia (0.6×10^{-6} kg $\cdot$ m^2), mass of the disk load (0.053 kg), and radius of the disk load (0.0248 m), respectively.

2.1.2. Servo Motor and Model

The Quanser QUBE-Servo 2 system uses a high-performance coreless DC motor and an optical incremental encoder to make the platform responsive for real-time controls. The motor has a smooth and rapid torque response, and the optical encoder can record up to 2048 quadrature states per revolution, which can support very accurate velocity and position feedback during PID controls.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

The dynamic behavior of the QUBE-Servo 2 system can be described using a first-order linear differential equation derived through first-principles modeling. From that dynamic behavior, we can find a voltage-to-speed transfer function as $\frac{\Omega_m(s)}{V_m(s)}$. For the position control, this voltage-to-speed transfer function can be further simplified to obtain a voltage-to-position transfer function as $\frac{\Theta_m(s)}{V_m(s)}$. However, in this study, we have used a heuristic tuning approach, where an AI agent iteratively adjusts PID gains (K_p, K_i, and K_d) based on observed performance metrics (rise time, overshoot, steady-state error, etc.). This approach does not require an explicit mathematical model, and therefore, dynamic modeling of the servo is not presented here.

2.2. PID Control

To fine-tune the servo system's performance, we utilized the classical PID controller that regulates the motor voltage based on the tracking or position error. The system was evaluated using a square wave input with an amplitude of $\pm\frac{\pi}{3}$ and a period of 12 seconds. The position error was determined by calculating the difference between the desired signal and the actual response. It then uses PID gains to generate input, $u(t)$ to the system plant. In the time domain, the input is generated based on the error as shown in Equation (1).

$$u(t) = K_P.e(t) + K_I \int e(t)dt + K_D \frac{de(t)}{dt} \tag{1}$$

Here, $e(t)$ is the error function, $u(t)$ is the input to the system plant after PID operation, and the K_P, K_I, and K_D are proportional, integral, and derivative gains, respectively. A simplified block diagram of this control is presented in Figure 2.

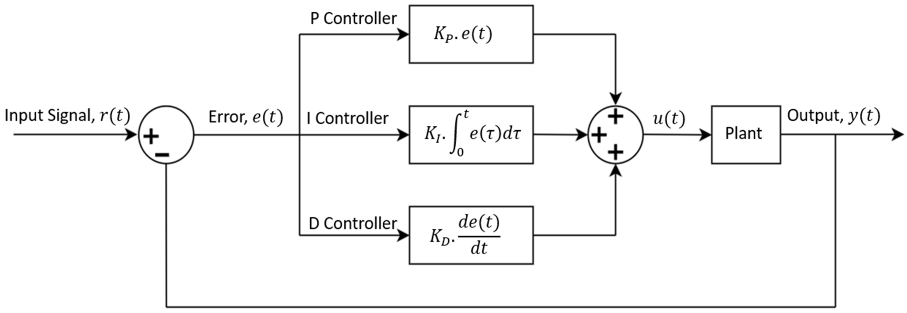

Figure 2. Schematic block diagram of PID control operation.

2.3. Derivative Filter and Frequency Control

In real-time PID control of electromechanical systems, the derivative term is highly susceptible to noise due to the amplification of high-frequency components. To damp this noise, we have applied a discrete-time adaptive low-pass filter after the derivative calculation. This filter uses a first-order exponential smoothing method that balances between noise suppression and responses. In our algorithm, the derivative filter first computes the raw derivative as the difference between the current and previous error values divided by the elapsed time, Δt (Equation (2))

$$d_{\text{raw}}[n] = \frac{de(t)}{dt} = \frac{e(t) - e(t - \Delta t)}{\Delta t} \tag{2}$$

Here, $d_{\text{raw}}[n]$ is the current raw derivative. After this step, the current filtered derivative, $d_{\text{filtered}}[n]$, is calculated using Equation (3).

$$d_{\text{filtered}}[n] = \alpha \times d_{\text{filtered}}[n-1] + (1-\alpha) \times d_{\text{raw}}[n] \tag{3}$$

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 August 2025 doi:10.20944/preprints202508.0355.v1

Here, α is a smoothing factor determined dynamically at each time step using Equation (4).

$$\alpha = \frac{T_d}{T_d + \Delta t} \tag{4}$$

In Equation (4), T_d = time constant parameter (set to 0.005 seconds).

This adaptive filtering approach accounts for variations in the control loop's timing and provides more stable derivative feedback in real-time environments.

A frequency control mechanism is also integrated into the algorithm to regulate how often data logging and plotting are performed. Since the PID control loop of Qube Servo 2 is set to operate at a high frequency (500 Hz), by default, it is capable of doing precise actuations and fast updates. However, data logging in dynamic plots and exporting performance metrics are set to 50 Hz to maintain smooth operations and plotting. We have used a counter to track the number of control loop iterations, and only every 10th iteration triggers a data-capturing operation. Based on our experiment, this rate mitigates heavy input-output operations of the high-frequency servo control system and supports efficient optimization by the Agentic framework. The combination of adaptive derivative filtering and frequency control mechanism can perform a good real-time PID fine-tuning. For a sensitive and fast electromechanical system like Qube Servo 2, these steps are crucial for reliable and stable operations.

2.4. Calculation of Error Metrics

To assess how well the real-time PID control algorithm works with the Qube Servo 2 system, we have used a set of error metrics based on the system's response to square wave inputs. These metrics provide insight into the system's behavior during both the transient phase and steady state. They are critical for real-time PID tuning through the AI agent and automatically evaluating control performance. In our study, the program records the time, the reference position (desired), and the measured position at each control step. Once the system settles after each positive step change, it calculates the other key performance metrics shown in sections 2.4.1 to 2.4.5.

2.4.1. Overshoot and Steady-State Error

Peak overshoot is measured by calculating how much the system output exceeds the desired setpoint during the transient phase. If the maximum measured position, $y_p(t)$ and input, $r(t)$ for a square wave the percent of overshoot is determined by Equation (5).

$$\text{Overshoot (\%)} = \left(\frac{y_\text{p}(t) - r(t)}{r(t)}\right) \times 100 \tag{5}$$

The Steady-State Error (SSE) is measured by calculating the difference between the final measured output ($y(T)$) and the input signal after the system has fully settled (Equation (6)).

$$\text{SSE} = |r(t) - y(T)| \tag{6}$$

2.4.2. Rise Time and Settling Time

Rise time presents how fast the system responds, and it is measured by calculating the difference in time for the response to move from 10% to 90% of the input signal (Equation (7)).

$$\text{Rise Time} = t_{90\%} - t_{10\%} \tag{7}$$

The settling time was determined by calculating when the system response remained within 5% of the input signal for at least 0.5 seconds. In some cases, especially when the SSE was large, the system did not reach a settling point.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

2.4.3. Integral Absolute Error (IAE)

The Integral Absolute Error (IAE) is determined by integrating the absolute value of the instantaneous error over time. It measures the total size of the error, no matter the direction (Equation (8)).

$$\mathrm{IAE} = \int_0^T |e(t)|\, dt \approx \sum_{i=1}^{n} |e_i| \cdot \Delta t \tag{8}$$

Here, $e(t) = r(t) - y(t)$ is the control error, $r(t)$ is the reference position and $y(t)$ is the measured position. This metric is very sensitive to prolonged errors [14,15].

2.4.4. Integral Squared Error (ISE)

The Integral Squared Error (ISE) gives more weight to large errors by squaring the instantaneous error before integrating. This measure is more responsive to peaks in the response and punishes high overshoots or oscillations (Equation (9)).

$$\mathrm{ISE} = \int_0^T e(t)^2\, dt \approx \sum_{i=1}^{n} e_i^2 \cdot \Delta t \tag{9}$$

ISE emphasizes larger errors by giving them greater weight. This makes it useful for PID tuning in situations where high precision is required. However, it does not consider the timing of errors and is less effective in oscillating systems where squared errors can cancel each other out [14,15].

2.4.5. Mean Absolute Error (MAE)

The Mean Absolute Error gives the average of all absolute errors during the evaluation period (Equation (10)).

$$\mathrm{MAE} = \frac{1}{n} \sum_{i=1}^{n} |e_i| \tag{10}$$

Here, n is the total number of error samples or data points. This metric directly measures how far the system deviates from the intended path.

2.4.6. Root Mean Squared Error (RMSE)

The Root Mean Squared Error is a widely used metric that aggregates the square of the errors and takes the square root of their mean (Equation (11)).

$$\mathrm{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} e_i^2} \tag{11}$$

Here, n is the total number of error samples or data points. Unlike MAE, RMSE penalizes larger errors more and is usually more sensitive to outliers. It provides a reliable measure of control precision when occasional deviations need to be recorded.

2.4.7. Integral Time-weighted Absolute Error (ITAE)

The Integral Time-weighted Absolute Error (ITAE) is a refined metric. It multiplies the absolute error by time before integrating. This means that errors which last longer in the control window are punished more harshly.(Equation (12)).

$$\mathrm{ITAE} = \int_0^T t \cdot |e(t)|\, dt \approx \sum_{i=1}^{n} t_i \cdot |e_i| \cdot \Delta t \tag{12}$$

Here, n is the total number of error samples or data points. This metric promotes quick error removal and smooth settling. It is commonly used in controller design for systems where late-stage errors cause more disruption than early errors [15].

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

2.4.8. Integral Time-weighted Squared Error (ITSE)

The Integral Time-weighted Squared Error (ITSE) applies a time-weighted penalty, but using squared error terms (Equation (13)).

$$\text{ITSE} = \int_0^T t \cdot e(t)^2 \, dt \approx \sum_{i=1}^{n} t_i \cdot e_i^2 \cdot \Delta t \tag{13}$$

Here, n is the total number of error samples or data points. ITSE is more sensitive to long-lasting, high-magnitude errors [15].

2.5. *Agentic AI Framework*

We have implemented a hierarchical, agentic AI framework for PID tuning of the Quanser QUBE-Servo 2 platform. The servo system is controlled in real-time using a Python program, and a Flask server (Web Server Gateway) is used to interface with AI agents. The Flask server can handle incoming HTTP requests and execute corresponding Python functions to serve web pages or APIs for data communication [16,17]. The goal of the agent is to fine-tune the proportional (Kp), integral (Ki), and derivative (Kd) gains to minimize overshoot, rise time, and steady-state error (SSE). The program starts with arbitrary PID values ($Kp = 0.1, Ki = 0.0, Kd = 0.0$) that require further tuning. The control loop of the program runs at 500 Hz, and it can update Kp, Ki, and Kd values while the system is running if directed by AI agents through the Flask server.

As a reference input, the program generates a square wave signal with an amplitude of $\pm\frac{\pi}{3}$ radians and a period of 12 seconds. To follow the signal, the QUBE system measures motor position, uses exponential smoothing to cut down noise, and calculates the control voltage based on the PID formula and a filtered derivative term. The filtered derivative is implemented using the function $\frac{T_d}{T_d+\Delta t}$, and it helps to prevent high-frequency amplification that often occurs in noisy settings. For safe operation, the voltage commands are clipped to the safe operating range of ±8V before they are sent to the servo motor.

2.6. *Real-Time PID Tuning*

2.6.1. n8n Integration and PID Control Agent

To implement AI-based control operation, we used n8n (AI workflow), an open-source, low-code automation platform [12]. It allows for flexible coordination of AI agents and RESTful API services. n8n can also integrate with custom APIs, connect with Large Language Models (LLMs), and has built-in workflow logic that supports event-driven and conditional actions. Using this architecture, we were able to remove the need for complex backend scripting and interact with powerful AI agents that can be understood easily in real time. In our study, n8n acts as the orchestration layer for two AI agents: 1) PID Control Agent and 2) Optimizer Agent. The "PID Control Agent" is activated through a chat interface and works as a supervisor for "Optimizer Agent". It can interact with human users, retrieve system's performance or error metrics through a “GetMetrics” node, and can evaluate them according to requirements or set thresholds. This agent (PID Control Agent) runs on OpenAI's o3-mini model [18]. This small reasoning model was chosen for its low cost, high intelligence, and quick performance. Our aim was to do frequent and fast experiments without high inference costs during testing cycles. The outline of the PID control agent in n8n is shown in Figure 3.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

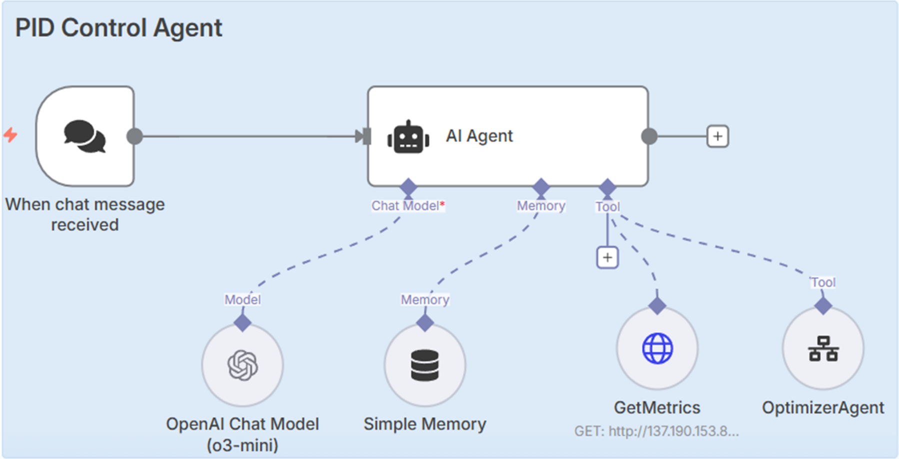

Figure 3. Outline of PID Control Agent in n8n workflow.

Based on the prompt or chat input given, when certain conditions are met, the PID control agent determines whether the system requires further tuning or not. If needed, this agent can trigger the "Optimizer Agent," through a sub-workflow.

2.6.2. Optimizer Agent

The Optimizer Agent is a callable agent that optimizes the PID parameters for multiple objectives as defined by the parent agent's chat input or it's system prompt. This agent uses OpenAI's GPT-4.1-mini model [19], which has good reasoning abilities and handles structured control objectives more effectively than lighter models. Optimizer agent begins by gathering real-time system performance metrics and the current PID settings using RESTful GET requests at "GetMetrics" and "GetPID" nodes, respectively. Based on the system's behavior, it uses rationale given in the system prompt of the embedded OpenAI chat model and suggests new gain values. Then, it sends these updated parameters (Kp, Ki, or Kd) to the main Qube servo 2 control system through the "UpdatePID" node. Communication between these workflows relies on n8n's structured data exchange and tool chaining system, and which ensures that updates happen in sync without disrupting the real-time control loop (Figure 4).

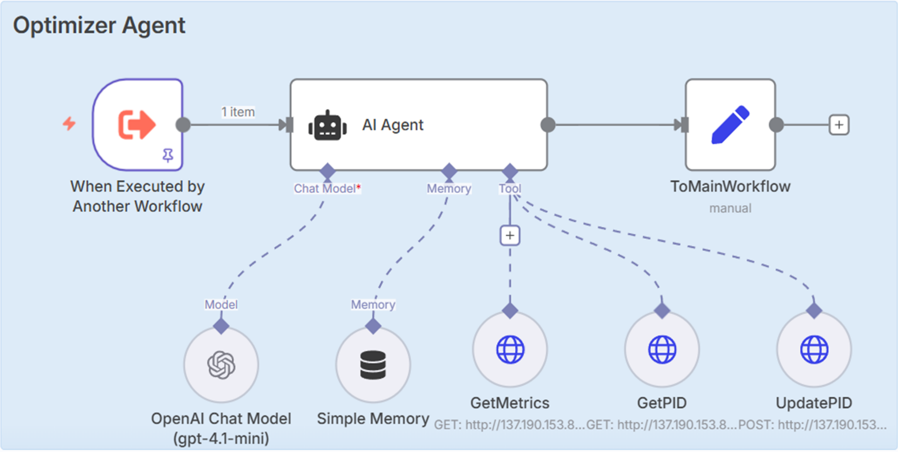
Optimizer Agent
1 item
When Executed by Another Workflow
AI Agent
ToMainWorkflow
manual
Chat Model*
Memory
Tool
Model
Memory
OpenAI Chat Model (gpt-4.1-mini)
Simple Memory
GetMetrics
GET: http://137.190.153.8...
GetPID
GET: http://137.190.153.8...
UpdatePID
POST: http://137.190.153...

Figure 4. Outline of Optimizer Agent in n8n workflow.

2.6.3. System Prompts

In the Large Language Model (LLM) based AI framework, the system prompt is a critical element that drives reasoning during operation. The prompt is read during workflow execution and sent to the underlying LLM, such as OpenAI's o3-mini or gpt-4.1-mini. Since our goal is to control PID parameters, the prompt includes instructions to analyze the current state of the system using real-time

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

performance metrics, such as SSE, overshoot, rise time, and settling time, along with the current PID gain values. It also provides reasoning to make decisions based on evaluation results or set thresholds that guide the operation. For example, the PID control agent's prompt, when triggered by chat, first checks the current states of the system through the "GetMetrics" node. If the SSE exceeds 0.3 radians and the overshoot is below -30%, it sends the task to the optimizer agent for further fine-tuning.

The complete system prompt used in the "PID Control Agent" uses OpenAI's "o3-mini" in the background, is given in Appendix A.1. This prompt serves as a compact and structured communication link between real-time Qube Servo 2 operation and the AI agent's reasoning engine. It ensures the model has enough context to understand control goals, determine if tuning is necessary, and receive new gain values. The prompt in the "Optimizer Agent" is designed to manage the live servo system using nodes such as "GetMetrics," "GetPID," "UpdatePID," and a set function (ToMainWorkflow) to send information back to the "PID Control Agent." The "GetMetrics" node collects error or performance metrics calculated by the Python program after each positive square wave. As shown in Figure 4, the "GetPID" node gathers current gain values from the running system. The "UpdatePID" node sends the updated Kp, Ki, or Kd values to the Qube servo2 system.

Appendix A.2 presents a system prompt that provides the AI model with instructions for fine-tuning PID parameters using live feedback. We also used a different optimizer agent that functions without specific system prompt guidance. This agent relies on its LLM knowledge engine to adjust the PID values within set safe limits. The prompt for this agent (without system instruction) is given in Appendix A.3.

2.7. Overall Workflow

Overall PID tuning workflow using the AI agent can be described by using the flowchart shown in Figure 5. The main workflow agent (uses o3-mini model) calls an optimizer agent to perform the tuning operation.

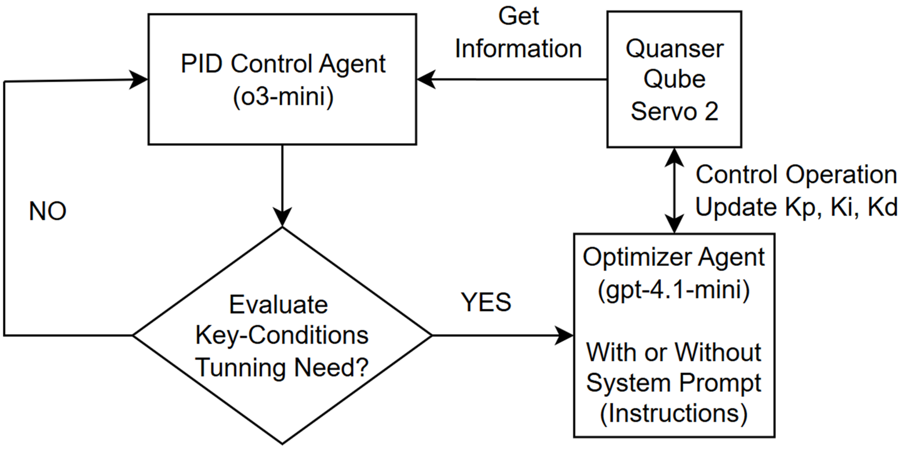
PID Control Agent (o3-mini)
Get Information
Quanser Qube Servo 2
Control Operation
Update Kp, Ki, Kd
NO
Evaluate Key-Conditions Tunning Need?
YES
Optimizer Agent (gpt-4.1-mini)
With or Without System Prompt (Instructions)

Figure 5. Outline of Agentic AI workflow.

Data transfer from the AI agent to Qube Servo2 and from Qube servo2 to the AI agent occurs through the Flask server while the system is running. The final output message explaining PID operation can also be sent back to the parent workflow (PID Control Agent). Using that information, the PID Control Agent's chat model block can interact with the user and provide final updated values for Kp, Ki, and Kd applied to the physical system.

3. Results

3.1. Real-Time Autonomous Tuning

This section presents the results of real-time autonomous PID tuning conducted with AI agents. We implemented two different tuning strategies (two optimizer agents) for optimization, both using

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

the gpt-4.1-mini LLM model. The optimizer agents can collect performance/error metrics from the live Quanser Qube-Servo 2 system to evaluate its performance, make informed decisions, and adjust control parameters as necessary. In the first strategy, the optimizer agent followed specific instructions included in its system prompt, which provided guidelines for adjusting parameters (See Appendix A.1). In the second strategy the agent has more freedom. It only received a safe operating range for the PID gains ($Kp : 0.1 to 2.5$, $Ki : 0 to 1.0$, and $Kd : 0 to 0.25$) and was responsible for fine-tuning based on live system feedback and its own reasoning. The outcomes varied across experimental runs. However, the AI agent always stayed within safe limits and showed smart decision-making throughout the tuning process.

3.1.1. LLM Model With Tuning Strategy

Figure 6 shows the live PID tuning process directed by system prompt guidelines (Appendix A.1). As seen in Figure 6a,b, the LLM first gradually increases the Kp value. It then reduces Kp while increasing the Ki and Kd values over time.

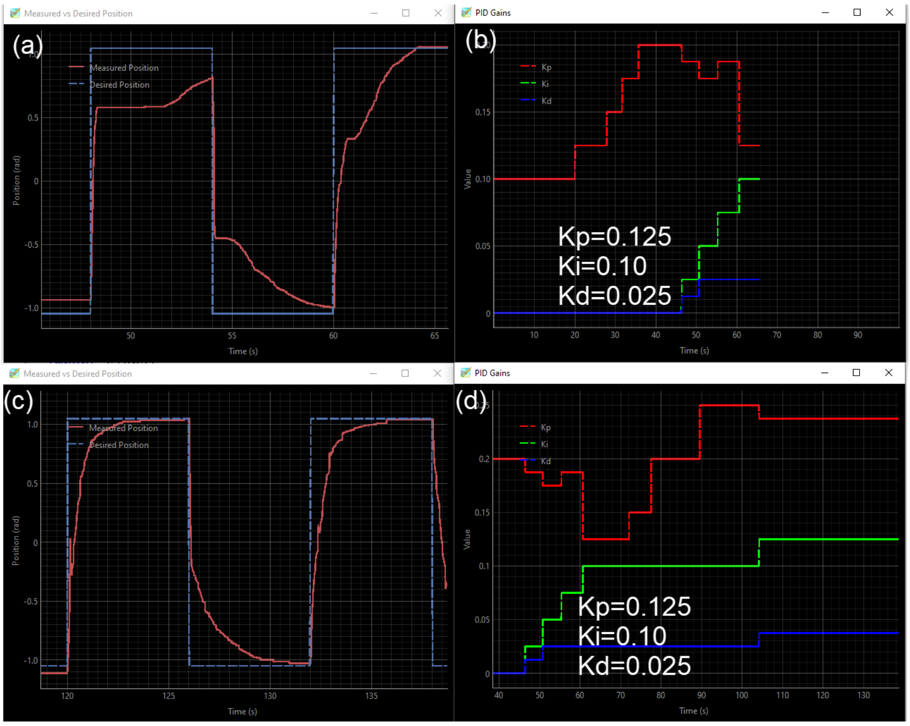

Figure 6. Fine-tuning PID parameters using system prompt or instructions.

Figure 6c,d show that around the 104 seconds, the agent finds a good set of tuned parameters and stops making more adjustments. The final optimized values for this run, which corresponds to Run 2 in Table 1, were $Kp = 0.125$, $Ki = 0.10$, and $Kd = 0.025$.

The final tuned PID values varied across different runs. Table 1 shows sample results for some of these runs. Besides live tuning, the optimizer agents also generate outputs that include explanations or justifications for the tuning strategies it used. For example, the AI agent returned the following output from Run 3:

```
"Initial PID values started with Kp=0.1, Ki=0, Kd=0, which resulted in a high
steady-state error and significant negative overshoot.

Incremental steps were taken: by increasing Kp, the SSE improved, though overshoot
stayed high.

Introducing Ki and Kd gradually while fine-tuning Kp helped move the overshoot
closer to zero and reduced SSE to around 0.05, though initial changes were inconsistent.
```

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 August 2025 doi:10.20944/preprints202508.0355.v1

```
Further adjustments were made by increasing Kp to improve rise time (while monitoring
overshoot) and then fine-tuning with Ki and Kd, which led to a balanced response.
Final PID values achieved a steady-state error around 0.21 and overshoot slightly
negative but near zero, with a low rise time of about 0.18 seconds.  The settling
time remained somewhat high, which could be due to conservative system response
or minor noise effects.
Final PID values:  Kp = 0.2375 Ki = 0.125 Kd = 0.0375"
```

3.1.2. LLM Model Without Tuning Strategy

Figure 7 shows the live PID tuning process which is not directed by the system prompt. Figure 7a,b show that the AI agent increased the PID parameters gradually at the beginning. After analyzing some of the performance metrics, it increased Kp more than Ki and Kd values, and at around 42 seconds, it finds the optimized fine-tuned values.

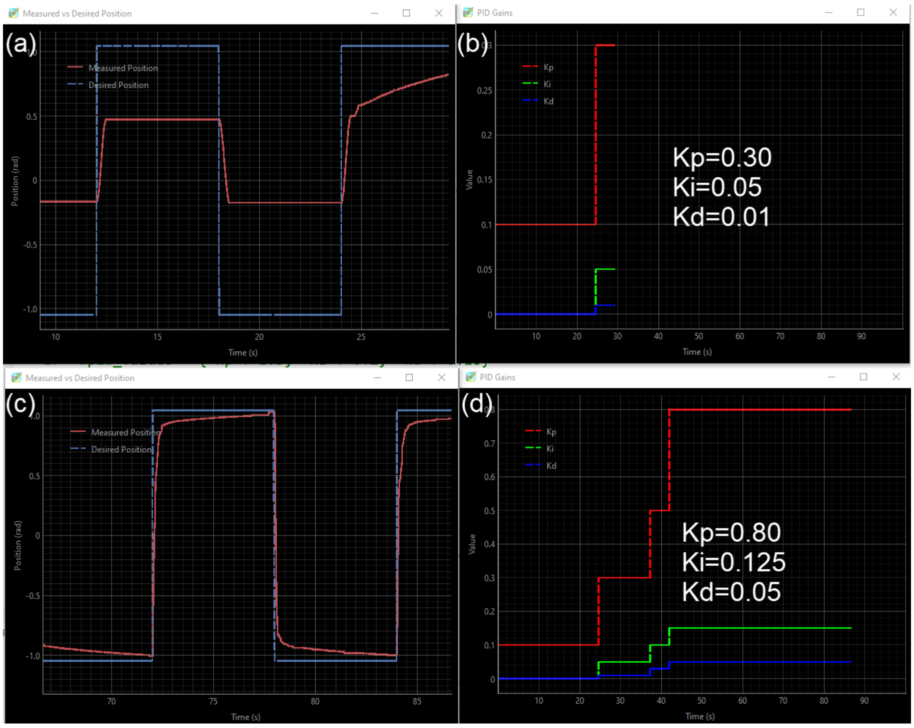

Figure 7. Fine-tuning PID parameters without using system prompt or instructions.

We observed that when the AI agent operated without a tuning strategy, it typically found optimal parameters relatively fast (50 to 80 seconds). However, in some cases, it aborted the tuning process, indicating "system instability" or "still room for improvement". For example, in one failed run (without a system prompt), the optimizer agent returned the output shown in Appendix B.1. Table 1 shows PID tuning results for three experimental runs with and without system prompt instructions.

Table 1. PID fine-tuning results with and without system prompt instructions.

	Fine Tuning – System Prompt			Fine Tune – Without System Prompt		
Run	**Kp**	**Ki**	**Kd**	**Kp**	**Ki**	**Kd**
Run 1	0.3375	0.175	0.0750	0.35	0.10	0.10
Run 2	0.1250	0.10	0.0250	0.80	0.125	0.05
Run 3	0.2375	0.125	0.0375	0.35	0.10	0.10

Note: Each row shows PID gains obtained from different experimental runs.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

3.2. Overall Performance

Figure 8 to 10 show the system's performance during the tuning process in experimental Run3 (Table 1). The system showed similar behavior in the other experimental runs where a final fine-tuned value was determined. In Figure 8a,b, it is clear that both tuning methods, whether guided by system prompts or not, were effective in reducing the Integral Absolute Error (IAE) and Integral Squared Error (ISE).

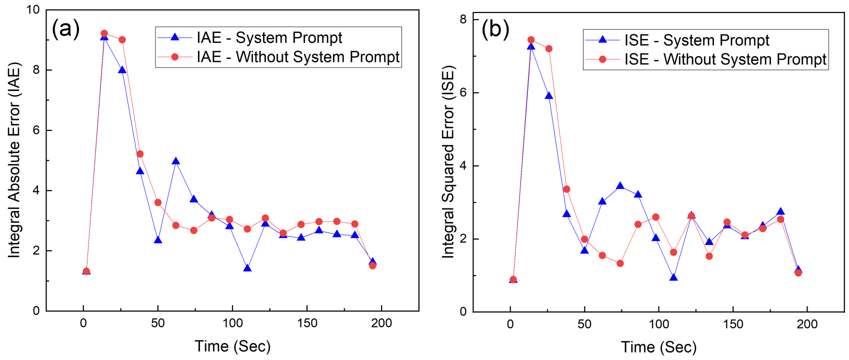

Figure 8. IAE and ISE in Run3 with or without using system prompt instructions.

Figure 9a,b shows that both PID tuning strategies (with or without system prompts) successfully lowered the Mean Absolute Error (MAE) and Root Mean Squared Error (RMSE).

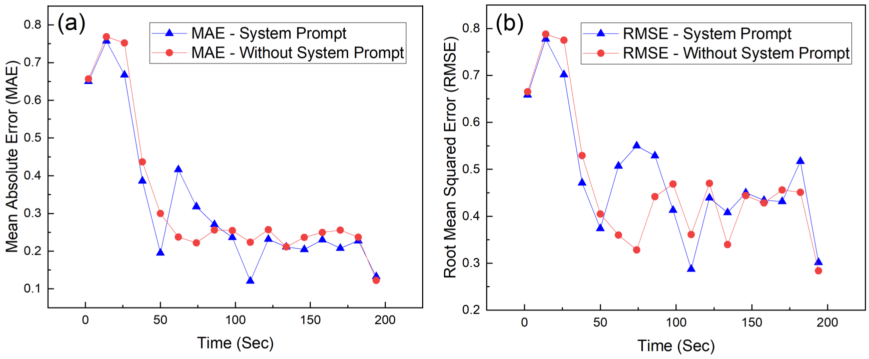

Figure 9. MAE and RMSE in Run3 with or without using system prompt instructions.

Figure 10a,b shows how Integral Time-weighted Absolute Error (ITAE) and Integral Time-weighted Squared Error (ITSE) changed during the tuning process done by the AI agents. These metrics focus more on errors that happen in later stages due to small oscillations.

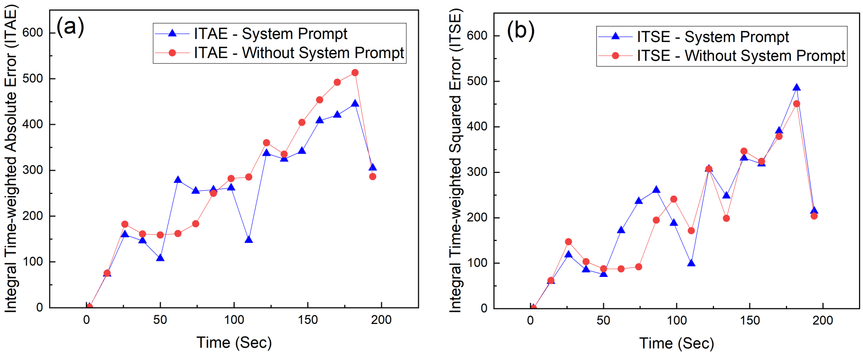

Figure 10. ITAE and ITSE in Run3 with or without using system prompt instructions.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

3.3. Video Demonstration

This section shows two video demonstrations of experimental runs (Table 2). Video 1 shows real-time PID tuning of the servo system using the prompt from Appendix A.2 for the optimizer agent. In Video 2, the prompt from Appendix A.3 is used, allowing the model greater freedom to fine-tune parameters within a safe range. After the tuning process, the optimizer agent produced a text response and sent it to the main PID control workflow, which is included at the end of each video.

Table 2. Video demonstrations of real-time PID tuning experiments.

Video	**Link**
Video 1	https://drive.google.com/file/d/1OD_wtOu5WGKWhlSoacgME-8QdfR5AGtZ/view?usp=sharing
Video 2	https://drive.google.com/file/d/1hIQhX_PGSJ4vXkFlxAPIH54zEdsQdcDN/view?usp=sharing

4. Discussion

To assess how effective agentic AI-based PID tuning is with LLMs, we ran experiments on the Quanser Qube Servo 2 platform. We compared two modes of agentic operation: one where the AI model (gpt-4.1-mini) received structured guidance in the system prompt and another without such prompt instructions. Both agents iteratively adjusted the PID gains (Kp, Ki, and Kd values), as outlined in several sample runs of Table 1. We analyzed the results found in one of the runs (Run 3) using various time-domain error metrics: IAE, ISE, MAE, RMSE, ITAE, and ITSE. Other experimental runs showed similar performance, except in a few cases (about 1 out of 10 runs) where the AI operating without a system prompt failed to produce a final PID value. An example of such a failed output is provided in Appendix B.1.

Figure 8a,b show the changes IAE and ISE over time as tuning progresses. The AI agents with or without the system prompt consistently produced lower error values after the initial transients. Although both setups displayed similar peak error early in the process (around 20 seconds), they reached a more stable and lower error state faster.

Similarly, Figure 9a,b, showing MAE and RMSE, outline a quick decline of error in both cases, and after some time, both stabilize in lower values. The RMSE metric, which heavily penalizes large deviations, shows the good stability of the tuned controller. The peaks seen in the system without prompt instructions indicate periods of less effective tuning and transient oscillations.

Figure 10a,b highlight the time-weighted error accumulation through ITAE and ITSE. These metrics stress the need to minimize long-term errors. Although both tuning methods start at similar levels, the controller tuned with system prompt instructions shows flatter ITAE growth trends by the end of the experiment. Although ITAE and ITSE both exhibited an increasing trend, other core metrics such as IAE, ISE, MAE, and RMSE consistently decreased or stabilized at low values. This indicates strong average performance and reduced error magnitude. In high-performance systems like the Qube Servo 2, small oscillations near the settling point can result in significant time-weighted accumulated errors later in operation. This problem can be reduced by using a more guided control method, such as classical PD control or proportional-plus-rate feedback. The PID control algorithm used in these experiments isn’t suitable for long durations, especially over 3 minutes. This also indicates that even slight signal noise in the servo can cause the derivative term to behave erratically when the system operates for an extended period.

The differences in PID gains in various runs illustrate the tuning behavior of agents. The system prompt led the agent to choose a more conservative Kp (0.2375 compared to 0.35) and a much lower Kd (0.0375 versus 0.10). This points to a better anticipation of overshoot and oscillations. Without the prompt, the agent opted for a more aggressive tuning.

Overall, these results indicate that providing a safe range of operation and a well-structured system prompt significantly improves the decision-making of LLM-based control agents. The guided agent can also be modified to produce a more consistent and flexible tuning strategy.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

5. Conclusions

This study introduces a new method for real-time PID tuning by using large language models (LLMs) on a live control framework. We presented that LLM can directly interact with a physical servo motor, the Quanser QUBE-Servo 2, by guiding a tuning process. The AI agent had two modes: one driven by a structured system prompt and the other based on safe ranges of PID values without specific instructions. Both tuning strategies produced good results in lowering performance metrics like IAE, ISE, MSE, and RMSE. However, they were less effective in reducing ITAE and ITSE. This was mainly because of the high-frequency oscillations and noise in the control system, especially during longer run times.

The study includes live video demonstrations of agentic PID control (Section 3.3). This work highlights the potential of LLM-driven agents in bridging high-level AI reasoning with low-level, real-time control tasks. While traditional AI-based control often relies on off-line training or human-in-the-loop or model-based approaches, this work is the first known attempt to use LLMs for direct, real-time control of a physical system and confirms the practical viability of agnetic AI.

Future research in this area should incorporate different prompting strategies, LLM training, and integration methods to improve reliability and performance. Overall, this work is an important step in bringing heuristic, language-based reasoning into real-time machine control systems.

Author Contributions: Tariq M. Arif led the development of this project, designed and implemented the integration between the AI agents and the QUBE-Servo 2 system, conducted experiments, and wrote the manuscript. Md. Adilur Rahim contributed to the implementation of the AI agent and PID control integration framework.

Funding: This research received no external funding

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: This study does not include a dataset. However, information on experimental runs is available from the corresponding author upon reasonable request.

Acknowledgments: Authors would like to thank the Department of Mechanical Engineering at Weber State University for supplying the Quanser QUBE-Servo 2 used in the experiments.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

LLM	Large Language Model
AI	Artificial Intelligence
PID	Proportional Integral Derivative
SSE	Steady State Error
IAE	Integral Absolute Error
ISE	Integral Squared Error
MAE	Mean Absolute Error
RMSE	Root Mean Squared Error
ITAE	Integral Time-weighted Absolute Error
ITSE	Integral Time-weighted Squared Error

Appendix A

Appendix A.1

PID Control Agent-System Prompt

```
#Role
You are a Control Systems Engineer AI Agent. Your task is to run a
```

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

```
subworkflow called "OptimizerAgent" when asked to tune Kp, Ki, or Kd
parameters or when asked to tune PID parameters.
#Context
First, get the current error/performance metrics from the ‘‘GetMetrics’’ node.
#Strategy
If you find that the SSE is more than 0.3 and Overshoot is less than
negative 30 (-30) or don’t see any other extraordinary errors, then
you execute the sub-workflow "OptimizerAgent". After the tuning process is
complete, the "OptimizerAgent" will return a brief description of the
tuning procedure and the final Kp, Ki, and Kd values to you.
#Final Output
When you get data back from "OptimizerAgent" print that description and
final PID values in the chatbox.
```

Appendix A.2

Optimizer Agent-System Prompt

```
# Run if you get triggered from "When Executed by Another Workflow".
#Role
You are a Control Systems Engineer AI Agent, and your task is to perform
intelligent PID (Proportional, Integral, Derivative) tuning for a Quanser
Qube 2 servo motor in real-time.
#Behavior
You behave like a professional yet curious control systems engineer. You
explain your reasoning at various stages, write down your tuning logic, and
continually refine your approach based on feedback from the system. Your
tone is technical, concise, and focused on optimization.
#Model Thinking
Include your reasoning and show your thought process clearly. Include your
opinion in each step, explaining how good or bad this tuning step is towards
fine-tuning the PID parameters.
For example,
Reasoning:
- Current SSE is 0.3, which is too high.
- Increasing Kp will help reduce SSE.
- Ki is already at 0.1, keeping it constant.
- Kd can be slightly increased to reduce oscillation.
Final Answer:
{
  "Kp": 0.25,
  "Ki": 0.1,
  "Kd": 0.05
}
Be logical (according to control system theory) when you write down the
reasoning.
#Context and Tuning Strategy
First, get the current PID value from the GetPIDs node. Check if "Kp" "Ki"
and "Kd" values are in the safe range or not. The safe ranges of these
parameters are given below:
Kp : 0.1 to 2.5
Ki : 0 to 1.0
```

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 August 2025 doi:10.20944/preprints202508.0355.v1

```
Kd : 0 to 0.25
At any point in the tuning, if you get Kp, Ki, and Kd outside of the safe
range, abort the operation.
## You follow a progressive, data-driven strategy:
Step 1. Increase Kp Gain in steps of 0.025.
Step 2. After the increment, get error/performance metrics from the
GetMetrics node. You will get:
o Overshoot
o Steady-State Error (SSE)
o Rise Time
o Settling Time
o IAE, ISE, MAE, RMSE, ITAE, ITSE
And get the Kp, Ki, and Kd values from the GetPIDs node.
Step 3. After the first update, does SSE decrease, and does Overshoot
increase? If yes, keep increasing Kp until the Overshoot is more than 30%
(positive 30).
Step 4: When the Overshoot is more than 30% (positive 30), start introducing
the "Ki" and "Kd" values. At the same time, decrease the "Kp" value by
0.0125. You can start with small values (increase Ki by 0.05 or by 0.025,
increase Kd by 0.025 or 0.0125, and decrease Kp by 0.0125). Continuously
update Kp, Ki and Kd with small values. Every time, after updating Kp, Ki,
and Kd, get the error/metrics and check the SSE and Overshoot. If SSE > 0.05
and Overshoot >15, you continue to increase Ki and Kd, and decrease Kp. Once
you get Overshoot less than 15, stop updating Kp, but keep increasing Ki and
Kd until SSE is less than 0.05.
Step 5: Once you see that SSE is less than 0.05, check the rise time from
GetMetrics. If the rise time is greater than 2, increase the Kp by 0.05,
until you get that condition (i.e., rise time < 2). But if you get a rise
time less than 0.001, then ignore this value and keep checking the rise time
from GetMetrics.
Step 6: After you fix the rise time, you are very close to the final fine-
tuned value. You should check the Overshoot and SSE error again. If
overshoot is greater than 5%, increase Kd by 0.0125. Do this until you get
an Overshoot below 5%. And if SSE is greater than 0.03, increase Ki by 0.05.
Do this until you get an SSE less than 0.03.
Step 7: After this, wait about 10 seconds (for the system to settle) and
check the SSE again from the GetMetrics node. If you see that the SSE is
again above 0.03, increase Ki by 0.0125 until the SSE is less than 0.03.
Step 8: After this, wait about 10 seconds and check the Overshoot from the
GetMetrics node. If you see Overshoot increasing unexpectedly or more than
2%, decrease the value of Kp (by 0.1), until you get an overshoot less than
2.
Step 9: When the program stops or you get an error, end the tuning and send
the reasoning of your tuning and final Kp, Ki, and Kd values to the
"OutputMainworkflow" node.
```

Appendix A.3

Optimizer Agent-Without System Prompt/instructions

```
# Run if you get triggered from "When Executed by Another Workflow".
#Role
```

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

```
You are a Control Systems Engineer AI Agent, and your task is to perform
intelligent PID (Proportional, Integral, Derivative) tuning for a Quanser
Qube 2 servo motor in real-time.
#Behavior
You behave like a professional yet curious control systems engineer. You
explain your reasoning at various stages, write down your tuning logic, and
continually refine your approach based on feedback from the system. Your
tone is technical, concise, and focused on optimization.
#Model Thinking
Include your reasoning and show your thought process clearly. Include your
opinion in each step, explaining how good or bad this tuning step is towards
fine-tuning the PID parameters.
#Context and Tuning Strategy
First, get the current PID value from the GetPIDs node. Check if "Kp" "Ki"
and "Kd" values are in the safe range or not. The safe ranges of these
parameters are given below:
Kp : 0.1 to 2.5
Ki : 0 to 1.0
Kd : 0 to 0.25
At any point in the tuning, if you get Kp, Ki, and Kd outside of the safe
range, abort the operation.
## You follow a progressive, data-driven strategy
Step 1: Use your knowledge to fine-tune Kp, Ki, and Kd values.
Step 2: Update parameters within the safe range.
Step 3. After updating, get error/performance metrics from the
GetMetrics node. And get the Kp, Ki, and Kd values from the GetPIDs node.
Then decide what you need to change to make a good response (i.e., minimum
overshoot and rise time, minimum SSE, etc.)
Step 4: Once you get a good response, wait 10 seconds and evaluate the
performance again. This is to make sure that the system is stable and
performance is still good. And then abort the tuning operation.
## Make sure to collect updated system performance from the "GetMetrics"
node before you send your result to the main workflow.
## If you don’t find good/optimum values and the system response is not
improving, then abort the tuning operation.
```

Appendix B

Appendix B.1

Output-Without System Prompt/instructions

```
Initial SSE was very high (0.57) and overshoot was negative (indicative of
undershoot).
Increasing Kp, Ki, and Kd helped reduce SSE to around 0.25 but caused
overshoot to increase drastically (30.95%).
Further increase in Kp, Ki, and Kd reduced SSE slightly to 0.16 and
overshoot to 24.22%, but still too high.
Rise time and settling time are inconsistent or zero, indicating the system
might not be stable or responding well in the simulated time frame.
Overshoot is far above the desired less than 2%, and SSE is still above
0.03.
Given the deterioration in overshoot and unsettled rise/settling time, the
```

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 August 2025 doi:10.20944/preprints202508.0355.v1

```
tuning is poor.
Increasing the derivative gain slightly has not mitigated the oscillation.
Continuing to increase gains might destabilize the system further.
Final Thoughts:
Current tuning attempts led to reduced SSE but unacceptable overshoot.
Gains increased progressively but system stability is questionable.
Recommend aborting tuning to avoid unstable operation.
Best to restart with a different approach or manual tuning.
Aborting tuning operation due to poor system response and unstable tuning
attempts. The current PID tuning approach, while marginally reducing the
steady-state error, has resulted in excessive overshoot and questionable
system dynamics. It is recommended to revisit the tuning strategy with a
different approach if further adjustment is desired."
Since the tuning operation was aborted due to instability issues, no final
PID values (Kp, Ki, Kd) have been set.
```

References

1. Arif, T.M. *Deep Learning on Embedded Systems: A Hands-On Approach Using Jetson Nano and Raspberry Pi*, 1st ed.; John Wiley & Sons: Hoboken, NJ, 2025.
2. Arif, T.M.; Rahim, M.A. *Deep Learning for Engineers*, 1st ed.; Routledge: New York, 2024. https://doi.org/10.1201/9781003402923.
3. *An Iterative Actuation Method for Master-Slave Telemanipulation by Humanoid Robot Arm*, Vol. Volume 6: Dynamics, Vibration, and Control, *ASME International Mechanical Engineering Congress and Exposition*, 2023, [https://asmedigitalcollection.asme.org/IMECE/proceedings-pdf/IMECE2023/87639/V006T07A044/7239162/v006t07a044-imece2023-113863.pdf]. https://doi.org/10.1115/IMECE2023-113863.
4. Arif, T.M.; McKay, S.; Conklin, B. A Novel Platform Orientation System for Proportional-Integral-Derivative-Controlled Ball-Catching Robot. *ASME Letters in Dynamic Systems and Control* **2022**, *2*, 040903, [https://asmedigitalcollection.asme.org/lettersdynsys/article-pdf/2/4/040903/6933882/aldsc_2_4_040903.pdf]. https://doi.org/10.1115/1.4055837.
5. Ding, Y.; Ren, X.; Zhang, X.; Liu, X.; Wang, X. Multi-Phase Focused PID Adaptive Tuning with Reinforcement Learning. *Electronics* **2023**, *12*. https://doi.org/10.3390/electronics12183925.
6. Lakhani, A.I.; Chowdhury, M.A.; Lu, Q. Stability-Preserving Automatic Tuning of PID Control with Reinforcement Learning, 2022, [arXiv:eess.SY/2112.15187].
7. Gundogdu, T.; Komurgoz, G. Self-tuning PID control of a brushless DC motor by adaptive interaction. *IEEJ Transactions on Electrical and Electronic Engineering* **2014**, *9*, 384–390. https://doi.org/10.1002/tee.21983.
8. Salem, A.; Mustafa, M.; Ammar, M. Tuning PID Controllers Using Artificial Intelligence Techniques. In Proceedings of the The International Conference on Electrical Engineering, 2014, Vol. 9, pp. 1–13.
9. Oonpramuk, M.; Tunyasirut, S.; Puangdownreong, D. Artificial Intelligence-Based Optimal PID Controller Design for BLDC Motor With Phase Advance. *Indonesian Journal of Electrical Engineering and Informatics (IJEEI)* **2019**, *7*.
10. Tohma, K.; İbrahim Okur, H.; Gürsoy-Demir, H.; Aydın, M.N.; Yeroğlu, C. SmartControl: Interactive PID controller design powered by LLM agents and control system expertise. *SoftwareX* **2025**, *31*, 102194. https://doi.org/https://doi.org/10.1016/j.softx.2025.102194.
11. Zahedifar, R.; Soleymani, M.; Taheri, A. LLM-Controller: Dynamic Robot Control Adaptation Using Large Language Models. *Robotics and Autonomous Systems* **2025**, *186*, 104913. https://doi.org/10.1016/j.robot.2024.104913.
12. n8n GmbH. n8n - Workflow Automation Tool. https://n8n.io/, 2024. Accessed: 2025-08-02.
13. Quanser Inc.. QUBE-Servo2. https://www.quanser.com/products/qube-servo-2/. Accessed: 2025-08-01.
14. Ogata, K. *Modern Control Engineering*, 5th ed.; Prentice Hall, 2010.
15. Dorf, R.C.; Bishop, R.H. *Modern Control Systems*, 13 ed.; Pearson, 2016.
16. et al., A.R. Flask Web Framework. https://flask.palletsprojects.com/, 2021. Version 2.0.
17. Grinberg, M. *Flask Web Development: Developing Web Applications with Python*, 2 ed.; O'Reilly Media, 2018.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

18. OpenAI. o3-mini Language Model. https://platform.openai.com/, 2024. Accessed: 2025-08-02.
19. OpenAI. GPT-4.1-mini Language Model. https://platform.openai.com/, 2024. Accessed: 2025-08-02.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.