Physically Accurate Code Generation with Large Language Models for Industrial Control Systems

James McClure
Masters of Embedded and Cyber-Physical Systems
University of California, Irvine
jhmcclur@uci.edu

Ting Husn (Ted) Wu
Masters of Embedded and Cyber-Physical Systems
University of California, Irvine
thwu2@uci.edu

Yanyabobo (Ember) Xue
Masters of Embedded and Cyber-Physical Systems
University of California, Irvine
yanyaobx@uci.edu

ABSTRACT:

The goal of this project is to improve code generation for Industrial Control Systems (ICS) by merging Large Language Models (LLMs) with a physics-based simulation. Our goal is to produce an accurate testbed environment model that can be used to deliver evaluative feedback on the code created by the LLM, using real-world sensor data. This feedback loop enables iterative code revision until the code is confirmed to be correct within the robotics simulator, such as Gazebo. The main focus is on PLCs, specifically using a Siemens S7-1500 PLC in a testbed owned by FischerTechnik. The project's ultimate goal is to create a pipeline that can automatically produce, improve, and implement PLC code, guaranteeing a closed-loop system from creation to use. Understanding how the testbed operates, generating code with LLMs, calibrating the simulator with real-world inputs, and optimizing the LLM based on feedback from the simulator are crucial steps in improving the efficiency and robustness of the code.

Keywords:

OpenPLC; Gazebo; Large Language Models; Industrial Control; Systems; Physics-based simulation; Code generation; Sensor data calibration

I. INTRODUCTION

Programmable Logic Controllers (PLCs) play a critical role in the field of industrial automation and controlling many cyber-physical systems. The traditional approach to developing software for PLCs demands substantial resources, time, and expertise from various fields. To help reduce the amount of resources and speed up the generation of code, we are planning to use an Large Language Model (LLM) coupled with an automated pipeline that will allow us to generate code in an iterative manner; while also providing quality assurance to ensure the code meets all operational requirements and quality assurance needs.

Our pipeline will consist of four main components. The first being a master control script written in Python, aka the “One Script to Run Them All” that will interact with the various elements in this pipeline, and will be responsible for handling dialogue with the LLM. The script will first grab an input prompt (in the form of a text file) that contains a set of instructions, as well as detailed systems description of the desired PLC and system we want to program code to control (in our case, a Fischertechnik robotics testbed).

Our master script will then feed this initial prompt into the LLM (such as LLama-3) to generate structured text code (which will then be read and extracted into an .st file by our script). This structured text file will then be uploaded to OpenPLC runtime (where the script will attempt to have OpenPLC compile the code). Any errors returned from this process will be captured by our master script, categorized, and combined with our own custom constructive criticism to provide feedback for our LLM model to follow when problem solving. Once the LLM has produced a revised version of the code, the process is repeated until the code successfully complies and no errors are found.

At which point, the code and real-world sensor data will be incorporated into our physics-based simulation on Gazebo. Gazebo will then attempt to execute the code in a simulated environment to model how the robot system behaves and try to automatically identify major issues (such as if the structured text code demands the robot to do impossible movements) or if there are other issues related to the simulated performance of the robotic system. This feedback will then be sent back to the LLM. Then the refined code will be run through both OpenPLC (to ensure the code still compiles correctly) and the Gazebo simulator; and the process will be repeated as many times as necessary until the LLM can generate a code that is verified as correct in both our OpenPLC compiler and the Gazebo simulator.

At which point, our master script will then save a final copy of the structured text file, before sending it to Tia Portal (through the Openness API) for final compilation and error checking. Again, if any errors are found at this stage, the error messages and feedback will be sent back to the LLM for problem solving and the cycle will be repeated until the code is accepted by both OpenPLC, Gazebo, and Tia Portal. At which point, our master script will automatically upload the code to the S7-1500 PLC for deployment on our physical test bed, a Fiscthertechnik robotics kit – where the code will then be executed for real-world testing.

II. BACKGROUND:

In recent years, LLMs like ChatGPT, Google Gemini, and LLama have shown major potential for use in programming, software engineering, and code generation. These models are often trained on very large and complex datasets that include programming languages; and coupled with the model's powerful natural language processing capabilities, are capable of interpreting textual prompts from users and use this input to generate code snippets. These capabilities can aid programmers in helping to streamline coding tasks, completing code segments, and in reducing development time. [1][2]

One of the areas where LLM-based code generation is actively being researched is in the software development and verification of code for industrial control systems (ICS) and PLCs. [3]. PLCs are a form of ruggedized, domain-specific, and real-time computer that are used to control machinery and automatic systems in a wide variety of cyber-physical applications – ranging from elevators, swimming pools, automatic gate controllers, amusement park rides, traffic light controllers, industrial robots, critical infrastructure (such as oil pipelines, manufacturing plants, electric grid), etc. [3-6] The programs run by PLCs are highly specialized and are created using one of five IEC 61131-3 programming languages [3][7]. For the purposes of this project, we will be mainly focusing on structured text, due to the fact that its syntax and structure is designed to resemble more conventional programming languages while also making use of the IEC 61131-3 verification schemes to be compatible for use with PLCs.

One of the main challenges with using LLMs to generate structured text code is quality assurance. This is because the software on PLCs is often required to operate both within narrow safety margins (to avoid risk to life and limb) and also strict complexity and timing requirements (to ensure smooth and reliable operation). Normally, software development under these circumstances would be carried out by a team of engineers and domain specialists using a model-based design approach that would incorporate real-world data (and simulations of the physical process) to determine the appropriate algorithm solution prior to programming. This design process would then also be typically followed-up by extensive testing and verification prior to deployment in order to ensure that the synthesized code meets all the strict guidelines and requirements the team had identified and laid out earlier.

However, given that most conventional LLMs lack the domain-specific knowledge of engineers, technicians, and other specialists when it comes to accurately characterizing problems in specialized applications such as industrial control systems (ICS), they can struggle to provide a working solution that meets all the necessary requirements [8]. Furthermore, many LLMs are also known to “hallucinate” from time-to-time, and this may lead to the generation of code with errors or that doesn’t fully address the strict requirements and guidelines of users. [9] Both these issues present obstacles when it comes to using LLMs for PLC code generation.

In this project, we aim to tackle both these problems by demonstrating a model-based feedback loop that will aim to enhance the code generation capabilities of conventional LLMs (like LLama-3) to allow the AI to generate high-quality ST code capable of satisfying the strict operational requirements of industrial control systems. We plan to do this by employing an iterative process where an engineer will textually outline the requirements and system specifications in an initial prompt that will then be sent to the LLM. Then, the code from the LLM will be automatically run through both a syntax checker (OpenPLC) and a physics-based robotics modeling tool (Gazebo) to generate and provide feedback to the LLM to help it iteratively solve the relevant problems as part of an automated fine-tuning process. Then, once the LLM generates code that passes all our quality assurance checks, our pipeline will then automatically deploy the code onto a testbed environment (in our case, a Siemens S7-1500 PLC connected to a miniature FischerTechnik testbed).

III. OBJECTIVES:

3.1: Overall Objectives of Project

Our main objective, as described above, is to create an automated end-to-end programming pipeline that can accept a textual input prompt from an user (such as a .txt file containing a list of requirements for the LLM to follow and a description of the robotic system and what it is doing). This prompt will then be sent to the LLM and then the output will be recorded and saved as a structural text file. This structural text file will then be checked for IEC 61131-3 compliance, syntax errors, and grammar by our IEC2PLC compiler (in our case, OpenPLC). If errors are returned during the process, they are sent back (along with other actionable feedback) to our LLM model, which will then attempt to solve the problem and modify the code. This is done as many times until we get a code that is free of syntax or grammar errors.

At which point, the code is fed into our physics-backed Gazebo simulation (which will use real-world sensor data to create a precise model of the testbed environment and simulate how the code will perform in the real-world. If issues are found by the physics-based simulation, evaluative feedback will be given back to the LLM to help it revise the code and fix the issues. Once the LLM is able to come up with code that passes both our syntax check and meets our model-based design requirements, the code will automatically be uploaded by our pipeline onto our real-world testbed environment (in our case a FischerTechnik testbed, that includes a Siemens S7-1500 PLC for practical, hands-on testing).

3.2. Overall and Original Objectives for Spring Quarter

Our main focus for Spring Quarter was to have completed all the major components for the LLM4PLC feedback loop. The milestones we have completed this quarter are as follows:

- Identifying all the hardware and software we need for the project
- Familiarize ourselves with the structured text programming language
- Familiarize ourselves with the physics-based Gazebo simulator
- Identifying the types of LLMs and datasets we will be using for this project
- Successfully installed LLMA-3 and set up API calls on our machines
- Trained a LoRa to improve structured text code generation from LLAMA-3.
- Demonstrated a functional proof-of-concept for our OpenPLC Runtime feedback loop (including API calls and automated feedback generation)
- Completed a customizable plugin for Gazebo to return the status of the simulation

- Gained a general sense of how the various puzzle pieces will fit together to form our fully integrated LLM4PLC pipeline

3.3. Updated Objectives for Fall Quarter based on Progress

For the following Fall Quarter, we will be working on completing the entire feedback loop between three components, as well as working on final integration of the pipeline to our testbed for deployment.

For each major components of our project, we will have the following objectives:

1. Large Language Model: I want to explore more sophisticated fine-tuning techniques in the upcoming quarter to see how well our LLM performs. This will include exploring various training techniques, adjusting hyperparameters, and incorporating more diverse and extensive datasets. Furthermore, I plan to explore the possibility of integrating several LLMs with OpenPLC in order to evaluate their efficacy and determine which model produces the most correct and dependable ST code. By taking a comprehensive approach, we hope to improve the LLM's overall performance and reliability by making it more capable of producing error-free code and more adept at responding to error feedback.
2. OpenPLC: Figure out how to implement our pipeline to allow automated compiling of the finalized .st code to the Tia Portal software and deployment to the S7-1500 PLC.
3. Gazebo: Establish a complete pipeline to be able to receive command from OpenPLC server and interact based on the command. Generate the status of simulation data and send it back to prompt the Large Language model through the server.

IV: SETUP:

4.1. Software Packages Used:

For each of the major components of our project, we are using the following software package described below:

4.1(a) Large Language Model:

Ollama: Used to enhance the language model's performance by providing advanced natural language processing capabilities.

Llama3: A state-of-the-art language model that significantly improves text generation and understanding.

Google Colab: An online platform that allows for the execution of Python code in a Jupyter notebook environment, facilitating the development and testing of models.

Hugging face: A widely-used library for deploying and fine-tuning transformer models, providing a comprehensive suite of tools and pre-trained models.

Unsloth: A specific library optimized for running large language models efficiently, focusing on resource management and performance optimization.

Google Gemini: For initial prototyping with OpenPLC and the python script, we also used the free API provided by Google for their Gemini Pro model.

4.1(b) OpenPLC Runtime:

Virtual Machine: WSL2 VM running Ubuntu version 22.04.4 LTS "Jammy Jellyfish"

OpenPLC: OpenPLC Runtime Version 3

Visual Studio: this was the programming IDE we used for coding our python script.

Python: for our master script, we are using Version 3.11.9 of Python.

4.1(c) Gazebo:

Gazebo Simulator Harmonic: CMAKE version 3.10, gz-sim version 8.4, gz-plugin2 version 2, gz-transport version 13

Mosquitto: Mosquitto version 2.0.18 for MQTT server

Visual Studio: Programming IDE for modifying script with C++ version 11

Autodesk Fusion 360: Create simulation model

4.2. Hardware Packages Used

For our testbed, we are using a Fischertechnik Training Factory Industry 4.0 9V kit (which includes a Siemens S7-1500 PLC) to conduct real-world testing.

V. STANDARDS:

5.1. Software Revision Control

Version-Specific Python Packages: I ensure that all Python packages used in the project are of specific versions to maintain dependency consistency. For example, I resolved conflicts like google-colab 1.0.0 requires requests 2.31.0, but we have requests 2.32.3 which is incompatible by specifying compatible versions.

Llama3: I utilize Llama3 as the primary language model, which is one of the most advanced open-source LLMs available. Ensuring the correct version of Llama3 is crucial for leveraging its full capabilities and maintaining compatibility with other tools in the project. By tracking the specific version of Llama3 used, I ensure reproducibility of the results.

Colab Notebooks with T4 GPU and Cloud RAM: I use Google Colab notebooks for development and testing, specifically leveraging the T4 GPU instances and cloud RAM. The cloud GPU provides significant computational power, accelerating model training and inference tasks, while the cloud RAM allows handling larger datasets and models without running into memory constraints.

Unsloth: I use the Unsloth library to optimize the performance and resource management of Llama3. Ensuring that the specific version of Unsloth is compatible with Llama3 and other dependencies helps in efficient utilization of resources and improving model performance.

Vscode: All the sdformat file for the simulation and the plugin script is saved and controlled within folders.

Control Scripts: All control scripts (before major edits and revisions) have been thus far saved under a different file name and version number. Though we later plan to transition to Github.

5.2. Communications Protocols

5.2(a) Current Protocols Used:

SSH: Secure Shell is used to remotely connect to the WSL virtual machine from the Windows 11 host machine, as well as execute commands to run the OpenPLC Runtime compiler (in the form of a shell script) on the virtual machine; and read the feedback from the compiler.

SCP: Used to transfer .st files between the Windows 11 host machine and the WSL virtual machine.

Mosquitto: MQTT server to have gazebo listen to the command/message. Mainly used in ROS-MQTT

5.2(b) Future Potential Protocols

NodeRed: Using NodeRed to connect OpenPLC with the MQTT server and send the message/command to the gazebo

TCP/IP: OpenPLC Runtime has an initial TCP protocol built, but can only access in Windows or Linux systems. Likewise, the data-link level protocols used by the S7-1500 PLC (PROFINET and S7) also utilize TCP/IP for communication.

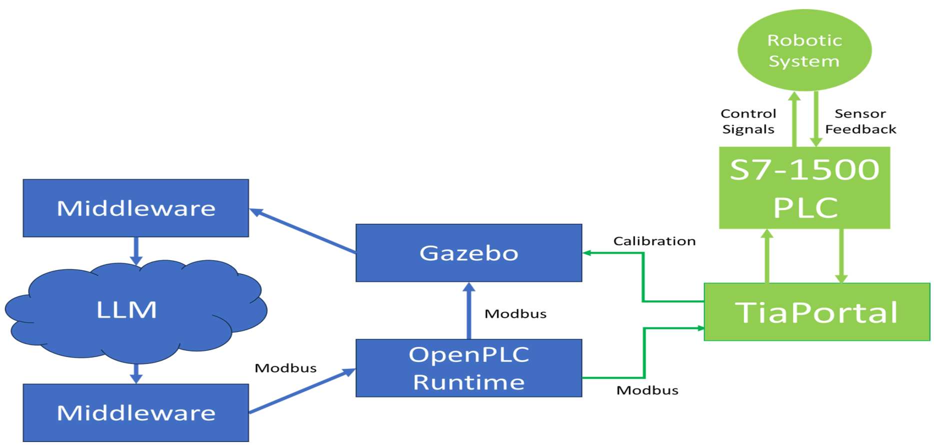
PROFINET: This is an industrial ethernet protocol that will be used to communicate between the S7-1500 PLC and the computer running CompTia Portal. This will be our primary method of uploading the code to the PLC.

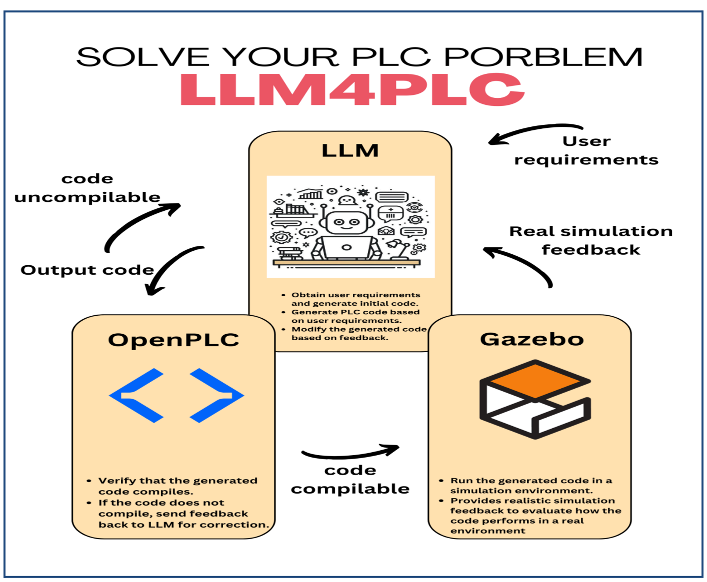
Siemens S7: This is a protocol developed by Siemens for transferring data into Supervisory Control and Data Acquisition (SCADA) systems. If needed, we could potentially use the Snap7 library (in Python or C/C++) to read real-world sensor data from the PLC.

VI. PROTOTYPES:

6.1. Iteration of Prototypes:

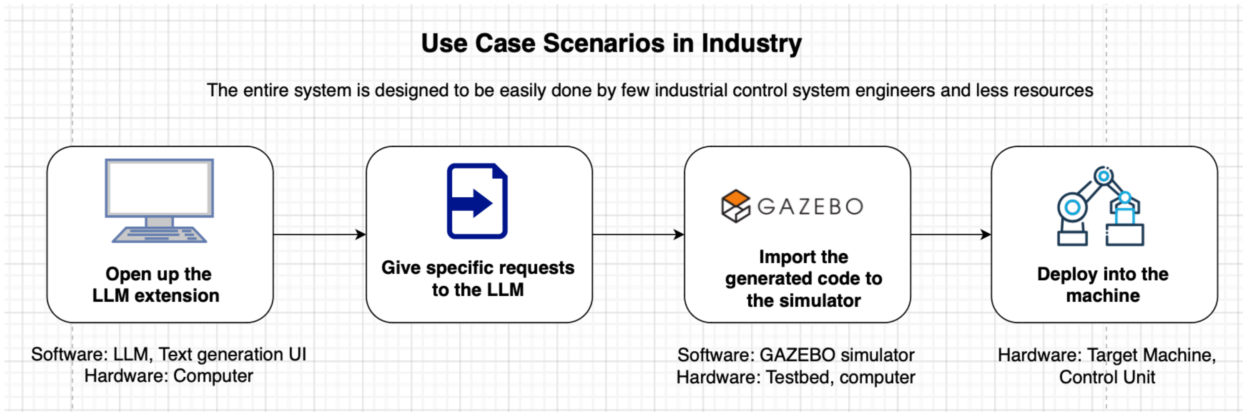
1. *System Flow Chart:*





The message and the data flow within the system start from giving command to the LLM and send it compiling within OpenPLC and result in Gazebo.

2. *Sample use case on the system within the industry:*



6.1(a) Large Language Model:

Initially, I utilized MyChatGPT to set up an LLM specifically for PLCs, referred to as LLM4PLC. The prompt used was designed to generate and explain ST code for advanced users. It explicitly gathered all necessary information regarding specific requirements, programming practices, and standards before generating code. This approach ensured that the model did not make assumptions and included comments in the code to explain the logic. ST was used as the default programming language.

Next, I attempted to enhance performance by using Flowise. The goal was to leverage links to improve the model's capabilities. However, this approach was not feasible due to limitations in the tool's performance and integration capabilities. Despite these challenges, this step provided valuable insights into the potential and limitations of Flowise for this application.

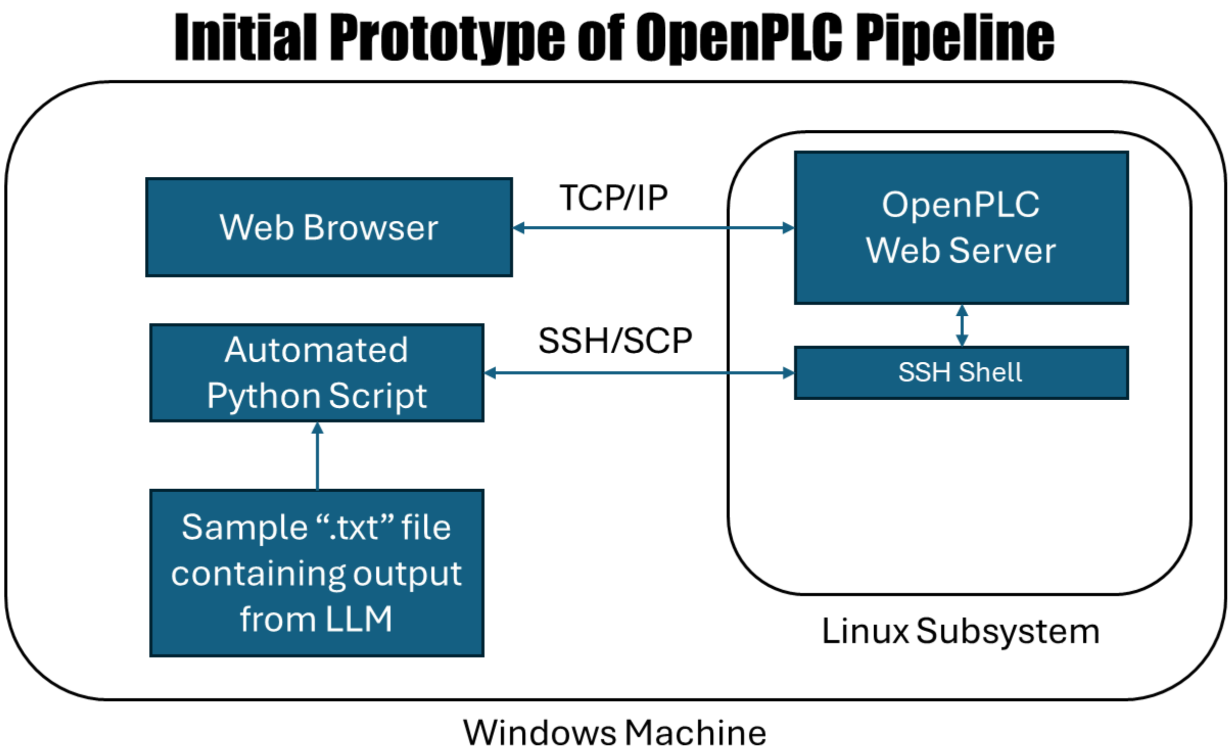
In the third phase, I explored fine-tuning an open-source model to improve the LLM's abilities. I attempted to perform this fine-tuning directly within the Ollama platform. However, it became apparent that Ollama did not support direct fine-tuning of models, limiting the effectiveness of this approach.

To overcome the limitations encountered in Ollama, I downloaded the original weights of the model from Hugging Face. The objective was to fine-tune the model on my local machine.

Unfortunately, the model was too large, and my hardware could not handle the fine-tuning process. To address this, I utilized low-rank adaptation (LoRa) techniques to manage the model size and resources. This process involved using Hugging Face's Transformers library to load the model and tokenizer, loading the configuration from a JSON file, and incrementally loading model weights using safetensors to mitigate memory constraints. Additionally, I implemented a function to generate text on the CPU to further reduce memory usage.

Finally, I turned to Google Colab to utilize its cloud GPU and RAM resources, enabling successful fine-tuning of the model. The code for this step, named llm4plc_lora.py, included installing necessary packages such as unsloth, xformers, trl, peft, accelerate, and bitsandbytes. I configured model parameters for maximum sequence length and 4-bit loading, and applied LoRa adapters to update a small percentage of the model's parameters. I then loaded and pre-processed a dataset from Hugging Face, conducted the fine-tuning on Colab's GPU, and monitored the training loss to ensure effective learning. The results showed significant improvements in the model's performance, and I saved the fine-tuned model and tokenizer for future use.

6.1(b) OpenPLC Runtime:



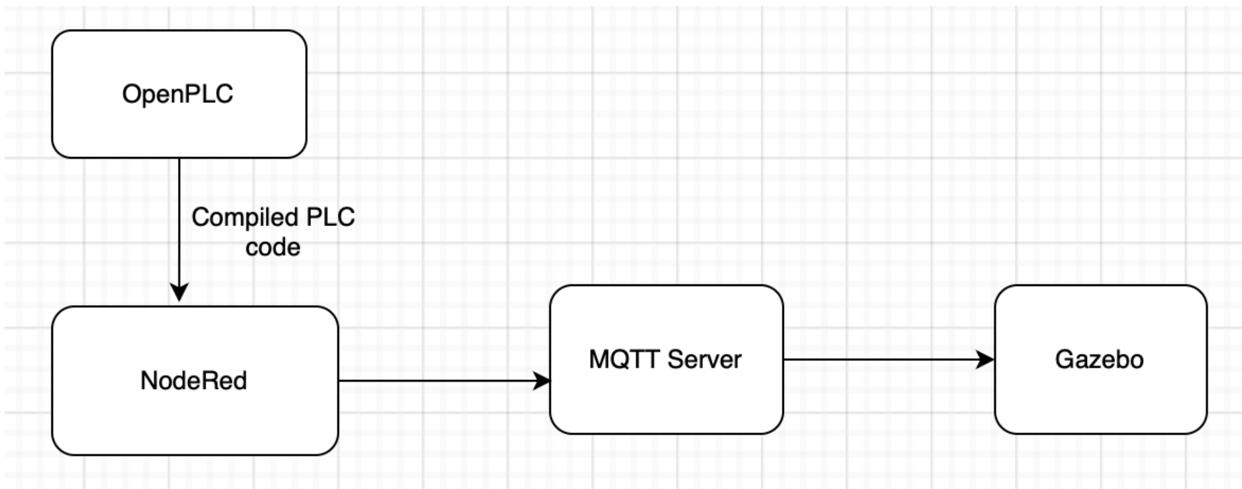
This figure shows our initial prototype system for communicating with the OpenPLC runtime system. Initially, we were manually typing our prompt into ChatGPT or Microsoft Copilot, and then copying and pasting the response from the LLM into a .txt file that will then be read and run by the program. Later iterations of the python script could automate the dialogue between the LLM and OpenPLC using automated API calls with Google Gemini. Future plans are to switch over to LLama-3 and move everything onto a single machine.

Currently, we are on Version 18 of our code, and the complete program is provided in Appendix A.

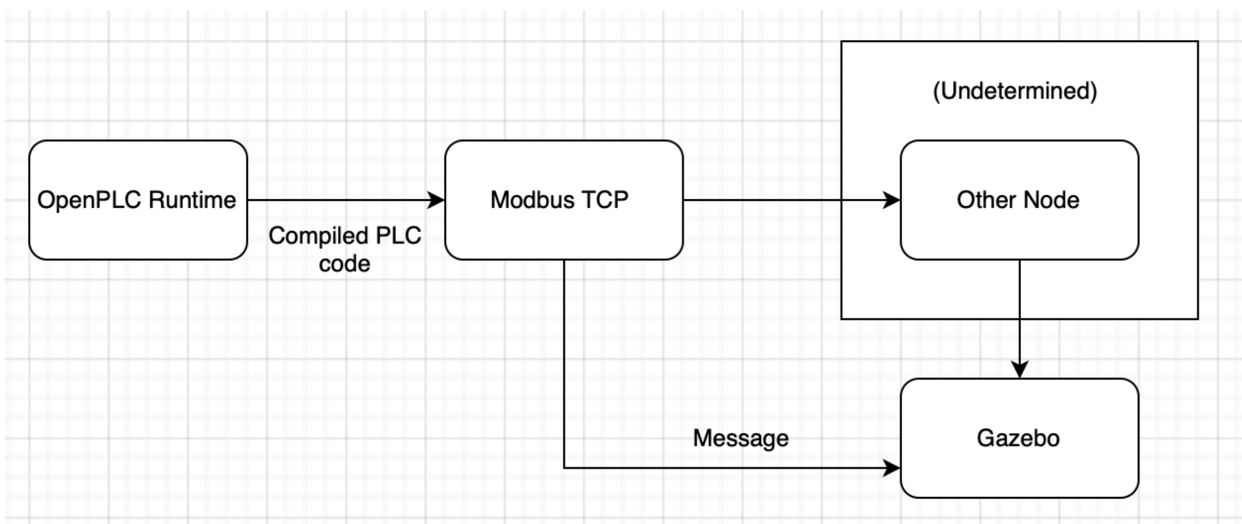
6.1(c) Gazebo:

Communication Protocol Iteration:

1. OpenPLC connects with NodeRed: Using NodeRed as bridge software to read data from OpenPLC and publish to an MQTT server and be received by Gazebo.



2. OpenPLC Runtime connect with TCP Server: Using OpenPLC Runtime with built-in TCP modbus service potentially to connect with additional nodes to link with Gazebo.



6.2. Explanation of Figures

In this experiment, we fine-tuned the FastLanguageModel to generate more accurate Structured Text (ST) code. The fine-tuning process began with configuring the model parameters, including setting the maximum sequence length to 2048, specifying the data type, and enabling 4-bit loading for efficiency. We then loaded the pre-trained model and tokenizer and configured the model to support Low-Rank Adaptation (LoRA) by adjusting parameters such as the rank, target modules, and dropout rate.

During the training phase, we used a specialized dataset tailored for ST code generation. We set the training parameters to ensure stability and effectiveness: the per-device batch size was set to 2, gradient accumulation.

In this experiment, we fine-tuned the FastLanguageModel to generate more accurate Structured Text (ST) code. The fine-tuning process began with configuring the model parameters, including setting the maximum sequence length to 2048, specifying the data type,

and enabling 4-bit loading for efficiency. We then loaded the pre-trained model and tokenizer and configured the model to support Low-Rank Adaptation (LoRA) by adjusting parameters such as the rank, target modules, and dropout rate.

During the training phase, we used a specialized dataset tailored for ST code generation. We set the training parameters to ensure stability and effectiveness: the per-device batch size was set to 2, gradient accumulation steps to 4, learning rate to 2e-4, and enabled mixed precision training with fp16 and bf16 support. These configurations were essential to handle the large dataset and complex computations efficiently.

Throughout the training process, we meticulously recorded the loss values at each step, resulting in a detailed training loss curve. This curve provides valuable insights into the model's performance over time. Initially, the loss was relatively high, indicating a significant prediction error. However, as training progressed, the loss values gradually decreased, reflecting the model's improving accuracy.

The curve also exhibited some fluctuations, which can be attributed to challenging samples or noise in the dataset. These fluctuations are typical and indicate the model's ongoing adjustments to minimize the loss. In the later stages of training, the loss values began to stabilize, suggesting that the model had reached an optimal state where further training provided diminishing returns.

VII. RESULTS (SO FAR):

7.1. Test Setup

To validate the PLC code generated by the LLM, we established a test setup involving the integration of Llama3 with OpenPLC, ensuring both systems are correctly configured to communicate. This includes verifying that OpenPLC is properly installed and functional for syntax validation. Additionally, we developed a Python script to automate the iterative process, which involves generating code with the LLM, sending it to OpenPLC for syntax checks, receiving feedback, and refining the code generation until it passes all syntax validations. This automated loop continues until the generated code is syntactically correct, ensuring reliable and ready-to-deploy PLC code.

7.2. Software and Hardware Used

Llama3: A state-of-the-art language model that significantly improves text generation and understanding.

Virtual Machine: WSL2 VM running Ubuntu version 22.04.4 LTS "Jammy Jellyfish"

OpenPLC: OpenPLC Runtime Version 3

Visual Studio: this was the programming IDE we used for coding our python script.

Python: for our master script, we are using Version 3.11.9 of Python.

Gazebo Simulator Harmonic: CMAKE version 3.10, gz-sim version 8.4, gz-plugin2 version 2, gz-transport version 13

Visual Studio: Programming IDE for modifying script with C++ version 11

7.3. Test Case and Objectives

Our goal is to assess how well a Large Language Model generates Structured Text code and how well it can adjust in response to feedback about errors. The goals are to confirm that the LLM can produce accurate ST code that satisfies given functional and syntactical constraints and to evaluate how sensitive it is to error feedback, indicating that it is capable of adaptive learning. In particular, we will evaluate if the LLM can correctly generate ST code for a task and whether it can adapt its output when fed input pointing out mistakes in the first answer.

7.4. Test Case Analysis

7.4(a) Large Language Model and OpenPLC Runtime:

This is the result from our latest iteration of code:

````
Program started at 14/06/2024 21:19:07


Sending first prompt to LLM model
Resonse from model
: ```st

CONFIGURATION Config0
        RESOURCE Res0
                VAR
                        Input1          AT %IX0.0 : BOOL;
                        Input2          AT %IX0.1 : BOOL;
                        Output1   AT %QX0.0 : BOOL;
                        Output2   AT %QX0.1 : BOOL;
                END_VAR
                TASK MainTask
                        VAR
                                i : INT;
                        END_VAR
                        BEGIN
                                Output1 := Input1;
                                Output2 := NOT Input2;
                        END_BEGIN
                END_TASK
        END_RESOURCE
END_CONFIGURATION

```
Writing LLM response to file
Successfully connected to JamesOMEN
Writing code to .st file
Structured code ends at line ```
Running: cd OpenPLC_v3/webserver/scripts/; bash compile_program.sh program_to_check.st >> /home/james/log.txt
Sent command
Error occured when running complier:
./st_files/program_to_check.st:0-1..0--1: error: unknown syntax error.

1 error(s) found. Bailing out!

Error Detected:
 ./st_files/program_to_check.st:0-1..0--1: error: unknown syntax error.

1 error(s) found. Bailing out!

Determinining response to model
Feedback to send to the model is ... Attempted to run the code but got error messages:
./st_files/program_to_check.st:0-1..0--1: error: unknown syntax error.

1 error(s) found. Bailing out!

Please revise the following code to fix the errors: ```

New Response from Model:
The error message suggests that there is a syntax error in the first line of the code. Upon reviewing the code, it appears that there is an extra period (.) at the end of the line. The corrected code is:

```st
CONFIGURATION Config0
        RESOURCE Res0
                VAR
                        Input1          AT %IX0.0 : BOOL;
                        Input2          AT %IX0.1 : BOOL;
                        Output1   AT %QX0.0 : BOOL;
                        Output2   AT %QX0.1 : BOOL;
                END_VAR
                TASK MainTask
                        VAR
````

````
                        i : INT;
                    END_VAR
                    BEGIN
                        Output1 := Input1;
                        Output2 := NOT Input2;
                    END_BEGIN
                END_TASK
        END_RESOURCE
END_CONFIGURATION
```

Please try compiling and running the code again. It should now run without errors.

------------------------------------------------
Writing code to .st file
Structured code ends at line ```

Running: cd OpenPLC_v3/webserver/scripts/; bash compile_program.sh program_to_check.st >> /home/james/log.txt
Sent command
Error occured when running complier:
./st_files/program_to_check.st:1-1..2--1: error: unknown syntax error.

1 error(s) found. Bailing out!

Error Detected:
 ./st_files/program_to_check.st:1-1..2--1: error: unknown syntax error.

1 error(s) found. Bailing out!

Determinining response to model
Feedback to send to the model is ... Attempted to run the code but got error messages:
./st_files/program_to_check.st:1-1..2--1: error: unknown syntax error.

1 error(s) found. Bailing out!

Please revise the following code to fix the errors: ```

Please try compiling and running the code again. It should now run without errors.

New Response from Model:
The error message suggests that there is a syntax error in the second line of the code. Upon reviewing the code, it appears that there is a missing semicolon (;) at the end of the line. The corrected code is:

```st
CONFIGURATION Config0
        RESOURCE Res0
                VAR
                    Input1          AT %IX0.0 : BOOL;
                    Input2          AT %IX0.1 : BOOL;
                    Output1     AT %QX0.0 : BOOL;
                    Output2     AT %QX0.1 : BOOL;
                END_VAR;
                TASK MainTask
                    VAR
                        i : INT;
                    END_VAR
                    BEGIN
                        Output1 := Input1;
                        Output2 := NOT Input2;
                    END_BEGIN
                END_TASK
        END_RESOURCE
END_CONFIGURATION
```

Please try compiling and running the code again. It should now run without errors.

------------------------------------------------
Writing code to .st file
Structured code ends at line ```

Running: cd OpenPLC_v3/webserver/scripts/; bash compile_program.sh program_to_check.st >> /home/james/log.txt
Sent command
Error occured when running complier:
./st_files/program_to_check.st:1-1..2--1: error: unknown syntax error.

1 error(s) found. Bailing out!

Error Detected:
````
````

```
./st_files/program_to_check.st:1-1..2--1: error: unknown syntax error.

1 error(s) found. Bailing out!

Determinining response to model
Feedback to send to the model is ... Attempted to run the code but got error messages:
./st_files/program_to_check.st:1-1..2--1: error: unknown syntax error.

1 error(s) found. Bailing out!

Please revise the following code to fix the errors: ```

Please try compiling and running the code again. It should now run without errors.

New Response from Model:
I apologize for the previous response. Upon further review, the error message suggests that there is a syntax error in the second line of the code. The corrected
code is:

```st
CONFIGURATION Config0
	RESOURCE Res0
		VAR
			Input1		AT %IX0.0 : BOOL;
			Input2		AT %IX0.1 : BOOL;
			Output1	AT %QX0.0 : BOOL;
			Output2	AT %QX0.1 : BOOL;
		END_VAR
		TASK MainTask
			VAR
				i : INT;
			END_VAR
			BEGIN
				Output1 := Input1;
				Output2 := NOT Input2;
			END_BEGIN
		END_TASK;
	END_RESOURCE
END_CONFIGURATION
```

Please note that I have added a semicolon (;) at the end of the line where the error was reported.

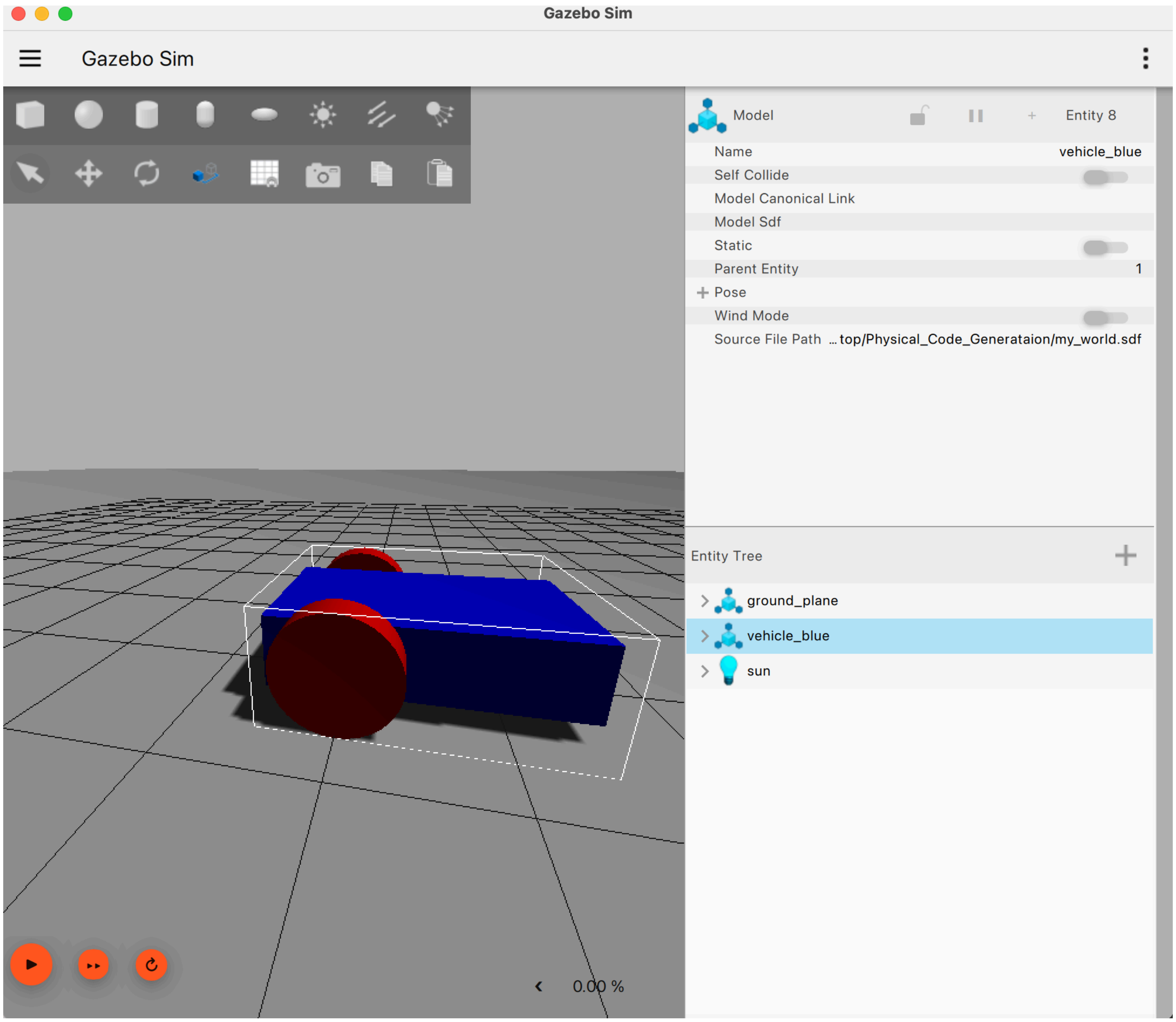
Please try compiling and running the code again. It should now run without errors.

```

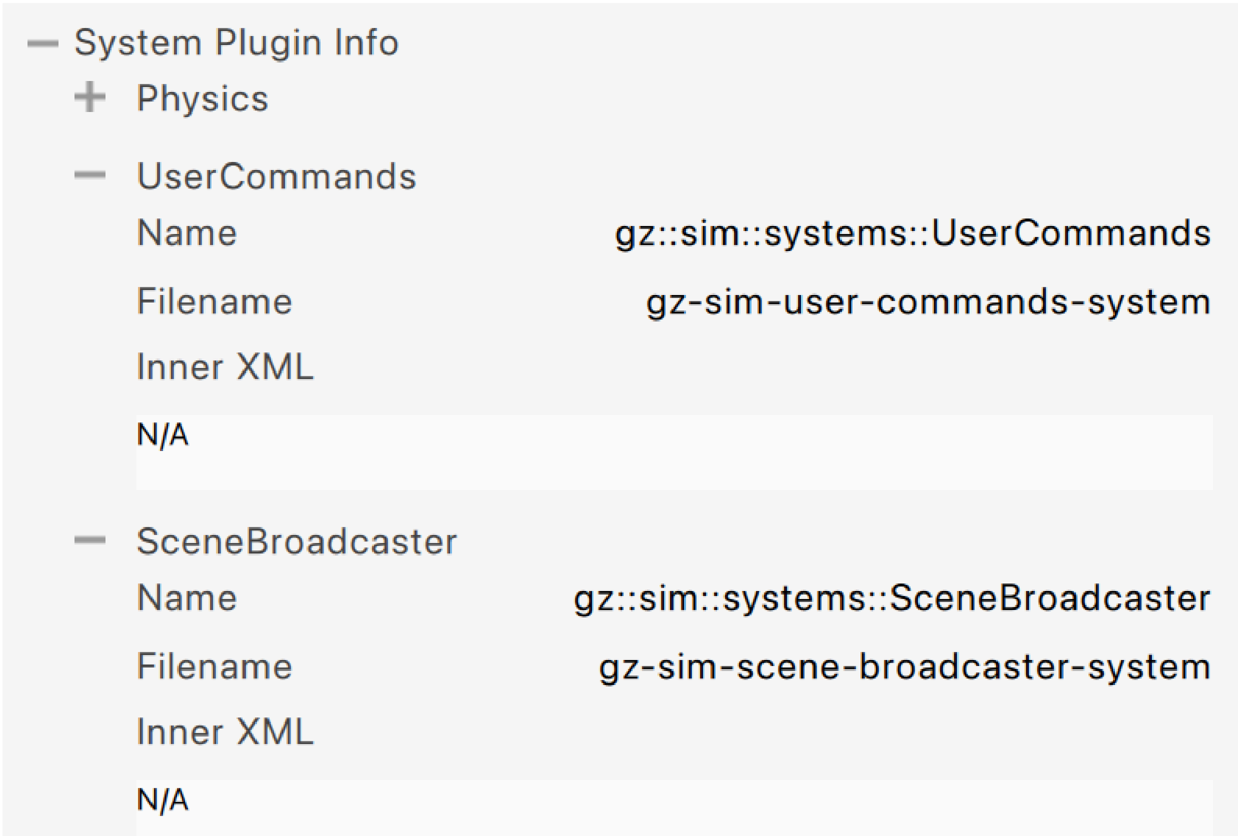
From the results, we observe that the LLM successfully generated ST code and continuously made corrections after receiving error feedback, largely meeting our expectations. However, it is also evident that the LLM's generated ST code frequently contained errors. We will continue to work on improving the accuracy of our LLM in the future.

7.4(b) Gazebo:

Loaded Own Iteration into Gazebo World:



Custom Plugin Loaded and can be used within Gazebo:



VIII. CONCLUSION

So far, our progress towards our ultimate goal of an end-to-end automated LLM4PLC pipeline has been good. We have been able to design and train a LoRa to optimize our structured text code generation, we also demonstrated how OpenPLC can be used with an LLM as part of our syntax-checking feedback loop, and developed a basic prototype for our master python script. We also completed a customizable plugin for Gazebo to return the status of the simulation and a sdf world model that can load our own iteration design from Fusion 360.

Over the summer, we plan to continue to make improvements to our initial subsystem prototypes and work out bugs. We plan to also begin work on integrating our various subsystems together to form the complete pipeline. Our goal for the summer is to ideally have a fully connected LLM4PLC pipeline going into Fall Quarter.

During Fall Quarter, our plan will be to focus on integrating our system with Tia Portal and getting our pipeline to talk to the S7-1500 PLC. Then hopefully, by the end of Fall Quarter, we will be able to demonstrate the operation of the full pipeline from start to finish, and have it successfully execute code on the FischerTechnik testbed.

IX: ACKNOWLEDGEMENTS:

For this project, we are grateful to have the support of Professor Mohammad Al Farque, Mohamad Habib Fakih, and Rahul Sirivas Dharmaji as our project mentors and advisors at the University of California, Irvine for providing us with advice, guidance, encouragement, and access to needed equipment (such as the S7-1500 PLC and the FischerTechnik Testbed). We would also like to thank Professor Quoc-Viet Dang and the University of California, Irvine for providing us with financial support.

Additionally, we also are thankful to have the assistance, advice, and technical support provided by Dr. Gustavo Quiros Araya from Siemens.

X. REFERENCES:

[1] Soliman, A., Shaheen, S. & Hadhoud, M. Leveraging pre-trained language models for code generation. Complex Intell. Syst. 10, 3955–3980 (2024). https://doi.org/10.1007/s40747-024-01373-8

[2] "Leveraging Large Language Models (LLMs) for Efficient Coding: A Deep Dive into Code Generation and its Impact on the BFSI Industry," www.linkedin.com. https://www.linkedin.com/pulse/leveraging-large-language-models-llms-efficient-coding-deep-anuyc (accessed Jun. 15, 2024).

[3] Mohamad Fakih1 , Rahul Dharmaji1∗ , Yasamin Moghaddas1∗ , Gustavo Quiros Araya2 , Oluwatosin Ogundare2 , and Mohammad Abdullah Al Faruque1 . 2024. LLM4PLC: Harnessing Large Language Models for Verifiable Programming of PLCs in Industrial Control Systems. In 46th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '24), April 14–20, 2024, Lisbon, Portugal. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3639477.3639743

[4] D. A. Pozhitkov, "PLC-Based Swimming Pool System," Circuit Cellar, Jan. 01, 2022. https://circuitcellar.com/research-design-hub/projects/plc-based-swimming-pool-system/ (accessed Jun. 15, 2024).

[5] "How PLC Applications Impact Our Daily Lives," www.gbctechtraining.com. https://www.gbctechtraining.com/blog/world-plcs-closer-you-think-plc-applications-our-everyday-lives

[6] "PLCs Programmable Logic Controllers - A Complete Guide," *Rs-online.com*, 2022. https://uk.rs-online.com/web/content/discovery/ideas-and-advice/plcs-programmable-logic-controllers-guide

[7] wendelien, “IEC 61131-3 | PLCopen,” Plcopen.org, Jul. 24, 2018. https://plcopen.org/iec-61131-3

[8] Z. Zheng *et al.*, “A Survey of Large Language Models for Code: Evolution, Benchmarking, and Future Trends,” *arXiv.org*, Nov. 17, 2023. https://arxiv.org/abs/2311.10372

[9] L. Huang et al., “A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions,” arXiv (Cornell University), Nov. 2023, doi: https://doi.org/10.48550/arxiv.2311.05232.

[10] “Gazebo Sim: Gazebo Sim,” gazebosim.org. https://gazebosim.org/api/sim/8/index.html (accessed Jun. 15, 2024).

[11] Koziolek, H., Gruener, S., & Ashiwal, V. (2023, September). Chatgpt for plc/dcs control logic generation. In *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)* (pp. 1-8). IEEE.

[12]Kim, Y., Lee, S. Y., & Lim, S. (2020, September). Implementation of PLC controller connected Gazebo-ROS to support IEC 61131-3. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)* (Vol. 1, pp. 1195-1198). IEEE.

APPENDIX A: MASTER PYTHON CODE (THUS FAR)

```
import paramiko, time
from scp import SCPClient
import google.generativeai as genai
from datetime import datetime

#Important files
Keychain = r'C:\Users\james\OneDrive\Desktop\Capstone_Project_Work\Temp_Keychain.txt'
#API Key and SSH Passcode
Windows_log_file = r"C:\Users\james\OneDrive\Desktop\Capstone_Project_Work\Feedback.txt"
#Windows log file
Linux_st_file = r"/home/james/OpenPLC_v3/webserver/st_files/program_to_check.st"
#ST file on OpenPLC Server
First_Prompt_File =
r"C:\Users\james\OneDrive\Desktop\Capstone_Project_Work\First_Prompt_V3.txt"
#First Prompt File
Gemini_Output_File =
r"C:\Users\james\OneDrive\Desktop\Capstone_Project_Work\Gemini_API_Raw_Output_V3.txt"
#Raw AI Output
Windows_st_file = r"C:\Users\james\OneDrive\Desktop\Capstone_Project_Work\Extracted_Code.st"
#Iterative ST file

'''
Suggestions
- Create different versions of the log file
- Save final ST code as a different file
- Need to send the final corrected file
'''

#-----------------------------------------------------------------------------------------------
----------------------------------------------------
def DataLogger(Update):
    print(Update)
    Logs = open(Windows_log_file, 'a')
    Logs.write("\n{}".format(Update))
    Logs.close()

#-----------------------------------------------------------------------------------------------
----------------------------------------------------
```

```
def ssh_login(hostname, username, password):
    try:
        # Create an SSH client instance
        client = paramiko.SSHClient()

        # Automatically add the server's host key to the local HostKeys object
        client.set_missing_host_key_policy(paramiko.AutoAddPolicy())

        # Connect to the SSH server
        client.connect(hostname, username=username, password=password)

        String = "Successfully connected to " + hostname
        DataLogger(String)

        #Return SSH client instance to main function
        return client

    except Exception as e:
        print("Error logging in SSH:", e)
        exit(1)

#Python Function for Interacting with Command Line
def run_command(command, activity):
    global client

    #Display activity
    DataLogger("Running: " + command)

    try:
        # Execute the command in the terminal
        DataLogger("Sent command")
        stdin, stdout, stderr = client.exec_command(command)

        print("Waiting for response\n")
        #Capture the output
        output = stdout.read()
        error = stderr.read()

        # Check if there's any error
        if error:
            String = "Error occured when " + activity + ":\n" + error.decode()
            DataLogger(String)
        else:
            String = "Output:" + output.decode()
            DataLogger(String)
        return output.decode(), error.decode()
    except Exception as e:
        print("An error occurred:", e)
        abort(1)

#Controlled shutdown function
def abort(mode):
    global client
    scp.close()
    client.close()
    exit(mode)

#-----------------------------------------------------------------------------------------------
---

def Convert_ChatGPT_2_ST(source, destination):

    #Declares flags
    Flag = 0
    WaitOneLine = 0

    try:
```

```
        with open(source) as input:
            with open(destination, "w") as output:

                #Scan files line by line
                DataLogger("Writing code to .st file")
                for line in input:
                    #Identifies which parts of output contain structured text code
                    if ((line.rstrip() == "```structured-text") or (line.rstrip() ==
"```structured text")):  #Start of structured text
                        DataLogger("Structure code begins at line " + line)
                        Flag = 1
                        WaitOneLine = 1
                    elif line.rstrip() == "```":
                        DataLogger("Structured code ends at line " + line)
                        Flag = not Flag

                    #Copies relevant sections into structured text file
                    if (Flag and not WaitOneLine):
                        output.write(line)
                    elif (Flag == 1 and WaitOneLine == 1):
                        DataLogger("Begin")
                        WaitOneLine = 0

                input.close()
                output.close()

    except Exception as e:
        print("Error formatting st file:", e)
        abort(1)

#**********************************************************************************************
*******************************************************************************

#Function for error handling
def ClassifyErrorAndFeedback(Error, Chat_Session):
    global Gemini_Output_File, Windows_st_file

    #Extract code from file
    try:
        with open(Windows_st_file, 'r') as file:
            code = file.read()
    except FileNotFoundError:
        print("File not found error, aborting...")
        abort(1)

    #---List of Error Messages I am looking for---
    DataLogger("Determinining response to model")

    #Error message caused when LLM improperly formats ST code block in response and my code
can't read it
    CannotReadPromptError =  """mv: cannot stat 'Config0.c': No such file or directory
    mv: cannot stat 'Config0.h': No such file or directory
    mv: cannot stat 'Res0.c': No such file or directory"""

    #Error message caused when code is ineligible to OpenPLC
    IneglibleCodeError = "Parsing failed because of too many consecutive syntax errors.
Bailing out!"

    #--Check which type of error message was gotten back--
    if Error is CannotReadPromptError:
        ModelFeedback = "Can you please reformat how the structured text response is enclosed
the response?"
    elif Error is IneglibleCodeError:
        ModelFeedback = "Can you please try again?"
    else:
        ModelFeedback = "Attempted to run the code but got error messages: \n{} \nPlease
revise the following code to fix the errors: {}\n".format(Error, code)
```

```python
    #Send Feedback to Model and Get New Response
    DataLogger("Feedback to send to the model is ... " + ModelFeedback)
    New_Response = Chat_Session.send_message(ModelFeedback).text
    DataLogger("New Response from Model: \n" + New_Response)

    DataLogger("\n------------------------------------------------")
    #Write new response to file
    Output = open(Gemini_Output_File, 'w')
    Output.write(New_Response)
    Output.close()

    return

#-----------------------------------------------------------------------------------------------
--------

#Extract A Specific Line from a Text File
def extract_a_line_from_a_txt_file(file_path, line_number):
    try:
        with open(file_path, 'r') as file:
            lines = file.readlines()
            if 1 <= line_number <= len(lines):
                desired_line = lines[line_number - 1]  # Adjust for 0-based indexing
                return desired_line.strip()  # Remove leading/trailing whitespace
            else:
                return f"Line {line_number} does not exist in the file."
    except FileNotFoundError:
        return f"File '{file_path}' not found."
        abort(1)

#-----------------------------------------------------------------------------------------------
--------

#Open Dialoge with AI Model
def Setup_Gemini(GOOGLE_API_KEY):
    global Gemini_Output_File, First_Prompt_File

    #Step 1: Move API Key
    genai.configure(api_key=GOOGLE_API_KEY)

    #Step 2: Get Model
    Gemini_Model = genai.GenerativeModel('gemini-pro')
    Gemini_convo = Gemini_Model.start_chat(history=[])

    #Step 3: Get Initial Prompt
    DataLogger("Sending first prompt to LLM model")

    prompt = None
    with open(First_Prompt_File, 'r') as Prompt1file:
        prompt = Prompt1file.read()
        Prompt1file.close()

    #Step 4: Test Response
    Gemini_reply = Gemini_convo.send_message(prompt)
    Gemini_response = Gemini_reply.text
    DataLogger(f"Resonse from model \n: {Gemini_response}")

    #Step 5: Extract Output to File
    DataLogger("Writing LLM response to file")
    Logs = open(Gemini_Output_File, 'w')
    Logs.write(Gemini_response)
    Logs.close()
    return Gemini_Model, Gemini_response, Gemini_convo

#***********************************************************************************************
*********************************************************************************
```

```python
#Main python fuunction for initiating command sequence
#Note: note: only have one instance of shell per command, so commands must be run in one shot
def main_control_sequence(Chat_session):
    global Gemini_Output_File, Windows_st_file, Windows_log_file, Linux_st_file
    global scp

    ActiveLoop = True
    counter = 0
    while ActiveLoop is True:

        #FILENAMES AND FILEPATHS
        #Windows_ChatGPT_Output =
r"C:\Users\james\OneDrive\Desktop\Capstone_Project_Work\Example_of_working_code.txt"
        #Windows_ChatGPT_Output =
r"C:\Users\james\OneDrive\Desktop\Capstone_Project_Work\V3_Chat_GPT_Raw_Output.txt"

        #PHASE 1 -- EXTRACT STRUCTURED TEXT FROM CHATGPT OUTPUT
        Convert_ChatGPT_2_ST(Gemini_Output_File, Windows_st_file)

        #PHASE 2 -- UPLOAD FILE TO OPENPLC RUNTIME WEBSERVER
        scp.put(Windows_st_file, recursive=True, remote_path=Linux_st_file)

        #PHASE 3 -- COMPILE ST FILE USING OPENPLC RUNTIME
        Activity = 'running complier'
        Command = 'cd OpenPLC_v3/webserver/scripts/; bash compile_program.sh
program_to_check.st >> /home/james/log.txt'
        Output, Error = run_command(Command, Activity)

        #Delay
        #time.sleep(1)

        #PHASE 4 -- RECORD ERROR MESSAGES OR OUTPUT TO FILE
        print("Writing complier feedback to file")
        ErrorFlag = 0
        if Error:
            DataLogger("Error Detected: \n {}".format(Error))
            print("Sending to error function")
            ClassifyErrorAndFeedback(Error, Chat_session)
            ErrorFlag = 1
        else:
            DataLogger("Code run successfully: \n" + Output)
            SuccessFeedback = "Thank you"
            Reply = Chat_session.send_message(SuccessFeedback).text
            print(Reply)

        #PHASE 5 -- RECORD ST FILE TO NEW FILE
        counter += 1
        if(counter >= 3):
            ActiveLoop = False

    #Send success message to model
    DataLogger("-------------------\n")

    #PHASE 6: Preform post-loop actions
    print("Sending feedback to LLM (WIP)")


    print("Operation Successful")
    return

#-----------------------------------------------------------------------------------------------
---------------------------------------------------------

#Program Entry
if __name__ == '__main__':
```

```python
#Get all needed creds
hostname = "JamesOMEN"
username = extract_a_line_from_a_txt_file(Keychain, 6)
password = extract_a_line_from_a_txt_file(Keychain, 9)
GOOGLE_API_KEY = extract_a_line_from_a_txt_file(Keychain, 3)

#Logs date and time when program is run
timecode = datetime.now()
timestring = timecode.strftime("%d/%m/%Y %H:%M:%S")   # dd/mm/YY H:M:S
Message1 = "Program started at {}\n\n".format(timestring)
print(Message1)

#Begins Log File
Logs = open(Windows_log_file, 'w')
Logs.write(Message1)
Logs.close()

#Starts a convo with LLM Model
Model, First_Response, Active_Session = Setup_Gemini(GOOGLE_API_KEY)

# Call the function to initiate an SSH login & establish clients
client = ssh_login(hostname, username, password)
scp = SCPClient(client.get_transport())

# Start main control sequence
main_control_sequence(Active_Session)


# Close the SSH connection
abort(0)
```