

CUSTOM AI SHELL

Gaurav Kapildeo Prasad

23BCE0449

Code: <https://github.com/gaurav-k-prasad/shell>

Abstract –

This project presents the development of a **Custom AI-Integrated Unix Shell** implemented in the C programming language. The shell combines traditional command-line functionality with an intelligent assistant capable of understanding natural language and automating command execution. Core shell features include command parsing, pipelining, command chaining, background job control, I/O redirection, and process group management.

The integrated **AI system** enhances usability by interpreting user intent through natural language prompts such as “*ai explain about loop in shell scripting*” or “*ai create a new rust project in my desktop*”. It analyses recent command history, operating system information, and command context to generate meaningful responses, explanations, or executable command sequences. The AI component incorporates feedback mechanisms by detecting command errors, logging them temporarily, and re-executing commands with contextual adjustments. To ensure safety, it also establishes guardrails and provides users with explanatory warnings before executing potentially unsafe operations.

This combination of traditional shell mechanisms and intelligent automation bridges the gap between manual command execution and AI-driven system interaction, demonstrating how artificial intelligence can enhance usability, safety, and adaptability in command-line environments.

Introduction –

A shell serves as a bridge between the user and the operating system, allowing execution of commands, management of processes, and interaction with system resources through a textual interface. Traditional Unix shells like Bash provide powerful control but often require users to remember exact command syntax and handle errors manually. This project aims to advance the conventional shell by integrating AI-driven assistance directly into its architecture.

The developed shell retains all essential Unix functionalities — including pipelines, command chaining, background jobs, terminal control, and signal management — while introducing an AI subsystem that interprets user intent through natural language. When users issue AI-

prefixed commands, the system analyses the last five executed commands, operating system context, and recent errors to automatically generate appropriate responses, explanations, or executable actions. For example, users can request the AI to *explain shell scripting concepts* or *perform a system operation* using descriptive phrases rather than explicit syntax.

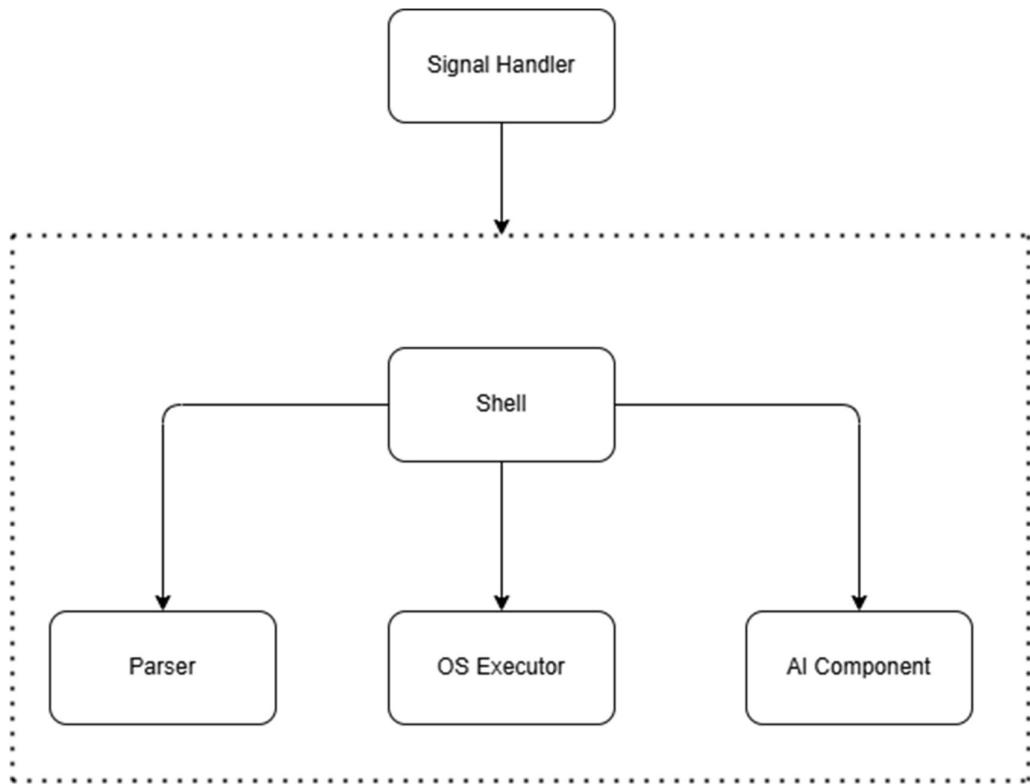
Moreover, the AI component continuously monitors execution results. If a command fails, it records the error in a temporary file and reattempts execution with improved context awareness. Before running commands, it also applies safety guardrails and provides users with warning messages or explanations about the implications of each operation.

By integrating AI reasoning with traditional process management, this shell not only deepens understanding of system-level programming concepts but also demonstrates how intelligent automation can simplify complex command-line tasks and make Unix environments more interactive, safe, and adaptive.

System Requirements –

Component	Description
Operating System	Linux / Unix-based system (Ubuntu, Fedora, Debian, etc.)
Compiler	GCC (GNU Compiler Collection) version 9.0 or above
Build Tool	“make” utility for compilation and automation
Libraries Used	Standard C libraries (stdio.h, stdlib.h, string.h)
	POSIX libraries (unistd.h, fcntl.h, signal.h, sys/wait.h, sys/types.h)
	Terminal control (termios.h)
AI Integration Dependencies	Lightweight text-processing and system introspection modules implemented in C (no external AI frameworks required) Python – 3.7+ and standard library
Shell Execution Environment	Any POSIX-compliant terminal supporting I/O redirection and job control
Error Logging	Temporary file system access (e.g., Shell directory itself for storing logs and error traces)

System Design –



2. Architectural Components

a. Command Parser

- Responsible for reading user input and tokenizing commands.
- Recognizes operators such as |, >, <, &&, ||, ;, and &.
- Builds a structured command tree or list for the executor.
- Differentiates between standard commands and AI-prefixed commands (ai ...).

b. Command Executor

- Executes parsed commands using fork() and execvp().

- Handles I/O redirection via dup2() and pipes for inter-process communication.
- Supports **pipelining, conditional execution, and command chaining**.
- Manages foreground and background job execution.

c. Job Control Module

- Maintains process groups and tracks background jobs.
- Handles signals (SIGCHLD, SIGINT) to pause, resume, or terminate processes.
- Ensures terminal control is correctly assigned using tcsetpgrp().

d. Terminal Handling Module

- Uses the termios library to enable **raw mode input** when reading commands.
- Provides custom line editing features (handling backspace, arrow keys, Ctrl+C, Ctrl+D).
- Restores canonical mode after each input cycle to maintain terminal stability.

e. AI Integration Layer

- Activated when the user enters commands beginning with the ai prefix.
- Analyzes:
 - The last five executed commands
 - Operating system information (e.g., current directory, environment variables)
 - Command history and recent errors (from temporary log files)
- Generates context-aware actions such as:
 - Explaining shell scripting concepts (ai explain about loop in shell scripting)
 - Constructing valid commands (ai create a new directory in parent directory)
 - Providing command usage hints or safer alternatives
- Includes **guardrail logic** that:
 - Simulates and validates commands before execution
 - Displays safety warnings and explanations
 - Logs execution attempts and outcomes

f. Error Handling and Feedback System

- Creates temporary log files (e.g., /tmp/gshell_error.log) to record failed commands.
- When re-executing, uses previous command context, error output, and OS information to refine execution or suggest corrections.

- Provides readable error explanations to assist users in understanding command failures.
-

System Workflow

Step 1: Input Handling

- Shell enters raw mode and captures user input.
- If the command starts with ai, control is passed to the AI Integration Layer; otherwise, it goes to the standard parser.

Step 2: Parsing and Validation

- Input is tokenized and checked for syntax errors.
- Redirections, pipes, and background flags are identified.

Step 3: Execution

- The Command Executor forks processes, sets up pipes, applies redirection, and executes programs.

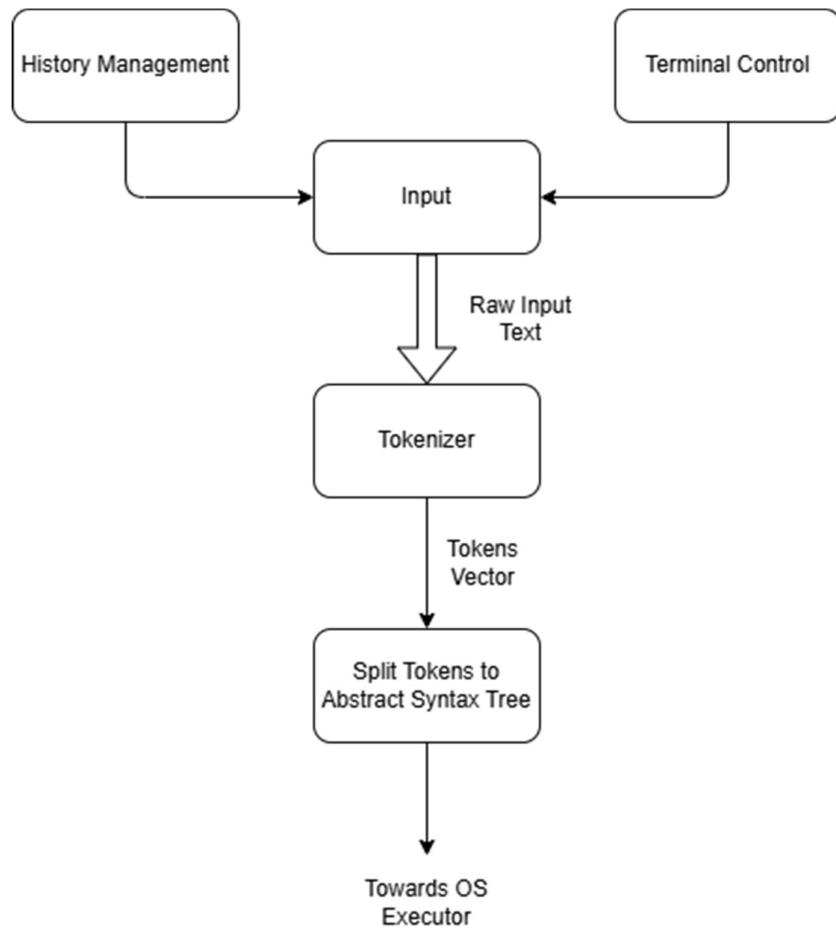
Step 4: AI Processing (if applicable)

- The AI module analyzes context and generates one of the following:
 - An explanation or set of instructions for the user.
 - An executable command sequence.
- Before execution, safety checks and guardrails are applied.

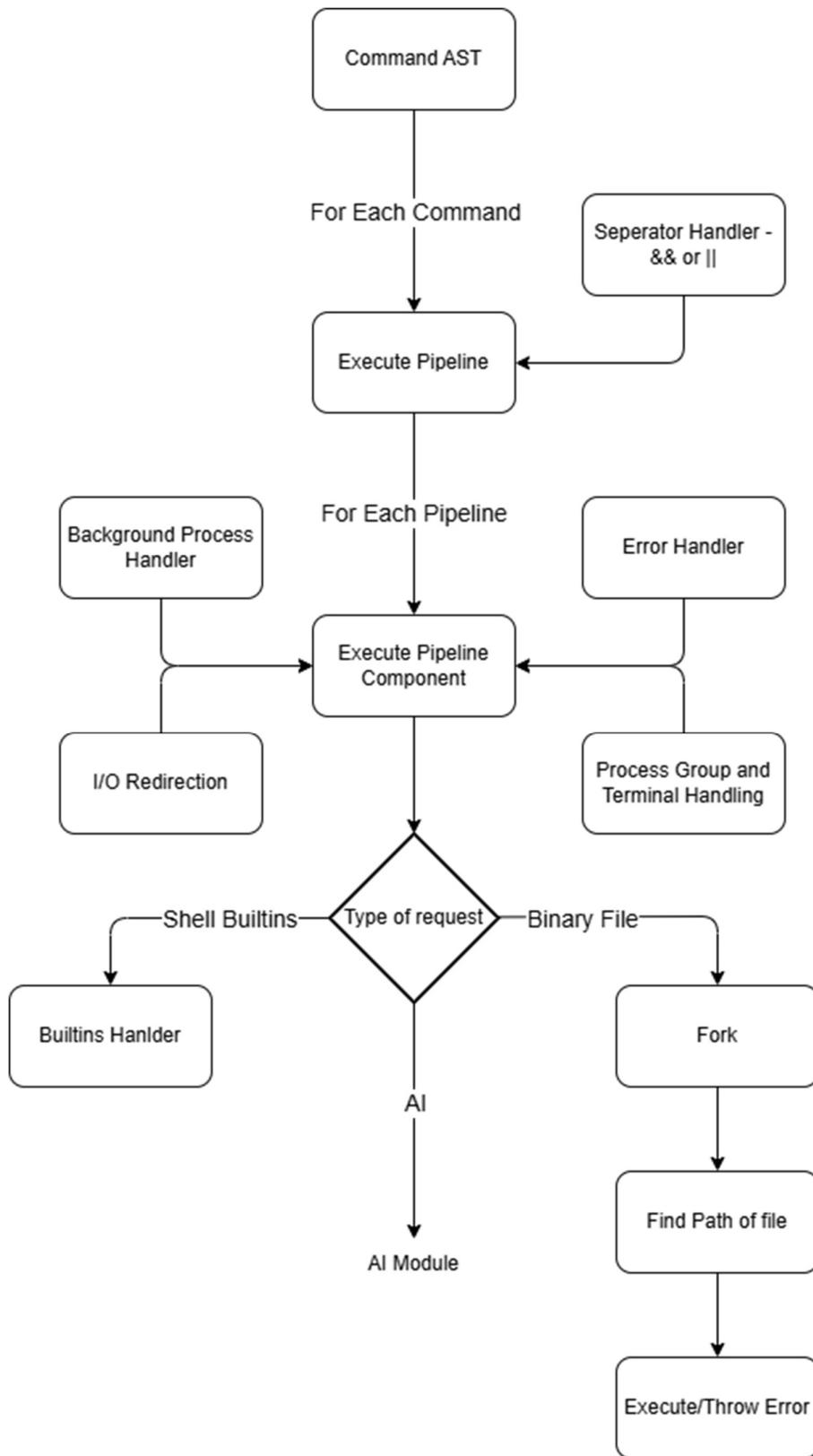
Step 5: Error and Feedback Loop

- If a command fails, the shell logs the error.
- The AI layer reads this context and optionally re-executes or explains the issue.

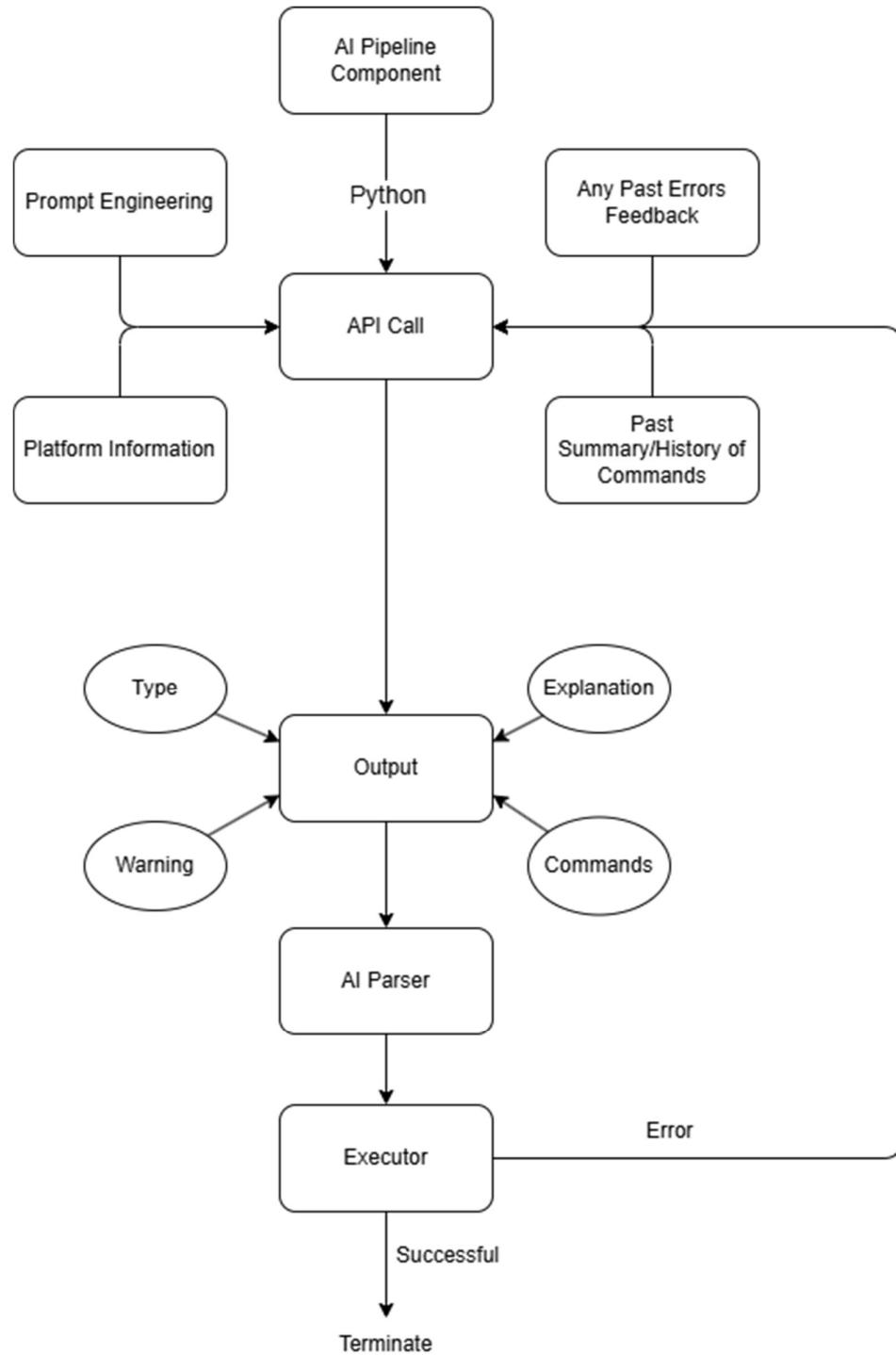
Parser –



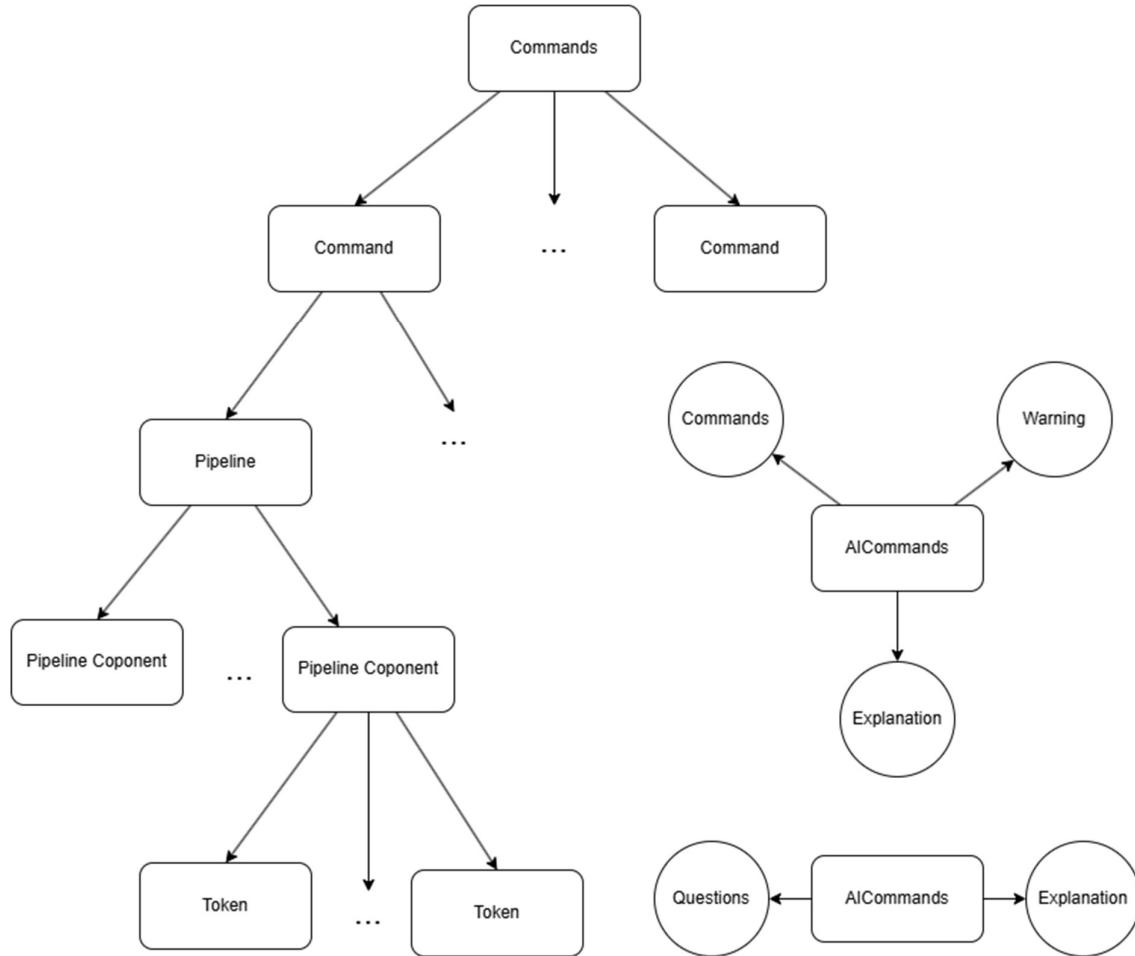
Executor –



AI -



Data Structures –



Output and Results –

```
@gaurav [/mnt/c/Users/gaura/Desktop/GitRepo/Shell]>
$ ai create a new rust project in desktop
AI never changes the state of current shell

-----Attempt 1/4-----

Answer these questions to get accurate results:
- What would you like to name your new Rust project?

Explanation: To create a new Rust project, I need a name for it.
Answer:
name it gaurav-rust-project
-----Attempt 2/4-----

Commands:
- cargo new ../../gaurav-rust-project

Explanation: Creates a new Rust project named 'gaurav-rust-project' in the Desktop directory.

Warning:

Do you want to execute the commands ([y]/n): y
cargo new ../../gaurav-rust-project

Creating binary (application) 'gaurav-rust-project' package
note: see more 'Cargo.toml' keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html
```

```
@gaurav [/mnt/c/Users/gaura/Desktop/GitRepo/Shell]>
$ ls && echo "hello world"; ls | grep gshell > write.txt; cat write.txt
AI      README.md  bugs.md   gshell           gshellAIHistoryInfo.txt  gshellAIPlatformInfo.txt  headers  structs  todo.md
Makefile __try    diagrams  gshellAIErrorInfo.txt  gshellAIOutputInfo.txt  gshellAITemp.sh     src      test.md
hello world
gshell
gshellAIErrorInfo.txt
gshellAIHistoryInfo.txt
gshellAIOutputInfo.txt
gshellAIPlatformInfo.txt
gshellAITemp.sh
```

```
@gaurav [/mnt/c/Users/gaura/Desktop/GitRepo/Shell]>
$ sleep 5 && pwd &
Process Group: 2856
@gaurav [/mnt/c/Users/gaura/Desktop/GitRepo/Shell]
$ /mnt/c/users/gaura/Desktop/GitRepo/Shell
Background process exit: 0
$ █
```

```
@gaurav [/mnt/c/Users/gaura/Desktop/GitRepo/Shell]>
$ ai create a python code and make a ml regression model. Take data from sklearn and execute and run inference on it. name the file test.py
AI never changes the state of current shell

-----Attempt 1/4-----

Commands:
- python3 <<'EOF' > test.py
- import pandas as pd
- from sklearn.datasets import load_diabetes
- from sklearn.model_selection import train_test_split
- from sklearn.linear_model import LinearRegression
- from sklearn.metrics import mean_squared_error, r2_score
-
- # Load the diabetes dataset as an example
- diabetes = load_diabetes()
- X = pd.DataFrame(diabetes.data, columns=diabetes.feature_names)
- y = pd.Series(diabetes.target)
-
- # Split data into training and testing sets
- X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
-
- # Initialize and train a Linear Regression model
- model = LinearRegression()
- model.fit(X_train, y_train)
-
- # Make predictions (inference) on the test set
- y_pred = model.predict(X_test)
-
- # Evaluate the model
- mse = mean_squared_error(y_test, y_pred)
- r2 = r2_score(y_test, y_pred)
-
- print("\n--- ML Regression Model Inference Results ---")
- print(f"Mean Squared Error: {mse:.2f}")
- print(f"R-squared: {r2:.2f}")
-
- # Example of single inference
- if not X_test.empty:
-     sample_input = X_test.iloc[[0]] # Take the first test sample
-     sample_prediction = model.predict(sample_input)
-     print(f"\nPrediction for first test sample (input: {sample_input.values[0]}): {sample_prediction[0]:.2f}")
-     print(f"Actual value for first test sample: {y_test.iloc[0]:.2f}")
- else:
-     print("\nNo test data available for single sample inference.")
- EOF
- python3 test.py
```

```
Explanation: Creates a Python script 'test.py' for an ML regression model using scikit-learn's diabetes dataset, trains it, and runs inference, then executes the script.
```

```
Warning:
```

```
Do you want to execute the commands ([y]/n):
```

```
File "/mnt/c/Users/gaura/Desktop/GitRepo/Shell/test.py", line 2
    --- ML Regression Model Inference Results ---
    ^^^^^^^^^^
SyntaxError: invalid syntax
----Attempt 2/4-----
```

```
Commands:
```

```
- cat > /mnt/c/Users/gaura/Desktop/GitRepo/Shell/test.py << 'EOF'
- import pandas as pd
- from sklearn.model_selection import train_test_split
- from sklearn.linear_model import LinearRegression
- from sklearn.metrics import mean_squared_error, r2_score
- from sklearn.datasets import load_diabetes
-
- # Load the Diabetes dataset from sklearn
- data = load_diabetes()
- X = pd.DataFrame(data.data, columns=data.feature_names)
- y = pd.Series(data.target)
-
- # Split data into training and testing sets
- X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
-
- # Initialize the Linear Regression model
- model = LinearRegression()
-
- # Train the model
- model.fit(X_train, y_train)
-
- # Make predictions on the test set (inference)
- y_pred = model.predict(X_test)
-
- # Evaluate the model
- mse = mean_squared_error(y_test, y_pred)
- r2 = r2_score(y_test, y_pred)
-
- print("---- ML Regression Model Inference Results ----")
- print(f"Mean Squared Error: {mse:.2f}")
- print(f"R-squared: {r2:.2f}")
- print("-----")
- EOF
- python /mnt/c/Users/gaura/Desktop/GitRepo/Shell/test.py
```

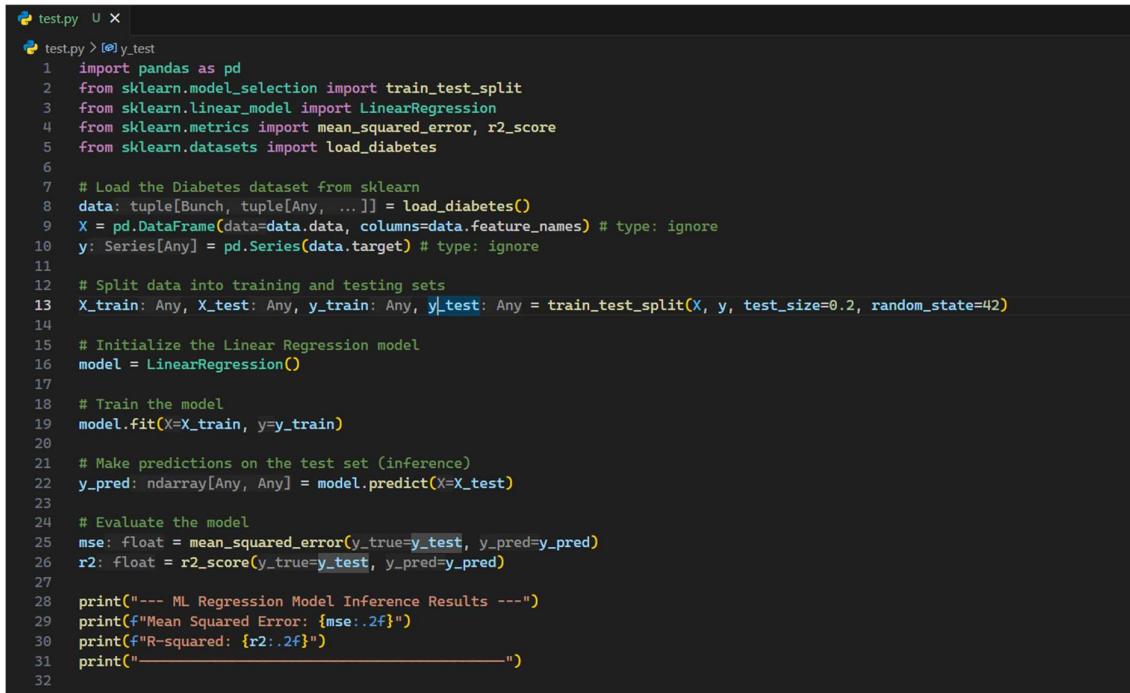
```
Explanation: Creates a Python script 'test.py' to build and run inference on a Linear Regression model using the scikit-learn diabetes dataset, then executes it.
```

```
Warning:
```

```
python /mnt/c/Users/gaura/Desktop/GitRepo/Shell/test.py
```

```
--- ML Regression Model Inference Results ---
Mean Squared Error: 2900.19
R-squared: 0.45-----
```

Created by Shell -



```
test.py  U x
test.py > [e]y_test
 1 import pandas as pd
 2 from sklearn.model_selection import train_test_split
 3 from sklearn.linear_model import LinearRegression
 4 from sklearn.metrics import mean_squared_error, r2_score
 5 from sklearn.datasets import load_diabetes
 6
 7 # Load the Diabetes dataset from sklearn
 8 data: tuple[Bunch, tuple[Any, ...]] = load_diabetes()
 9 X = pd.DataFrame(data=data.data, columns=data.feature_names) # type: ignore
10 y: Series[Any] = pd.Series(data.target) # type: ignore
11
12 # Split data into training and testing sets
13 X_train: Any, X_test: Any, y_train: Any, y_test: Any = train_test_split(X, y, test_size=0.2, random_state=42)
14
15 # Initialize the Linear Regression model
16 model = LinearRegression()
17
18 # Train the model
19 model.fit(X=X_train, y=y_train)
20
21 # Make predictions on the test set (inference)
22 y_pred: ndarray[Any, Any] = model.predict(X=X_test)
23
24 # Evaluate the model
25 mse: float = mean_squared_error(y_true=y_test, y_pred=y_pred)
26 r2: float = r2_score(y_true=y_test, y_pred=y_pred)
27
28 print("---- ML Regression Model Inference Results ----")
29 print(f"Mean Squared Error: {mse:.2f}")
30 print(f"R-squared: {r2:.2f}")
31 print("-----")
32
```

Future Enhancements

- Command history and autocomplete
- Better error handling
- Support for scripting
- Windows compatibility

Conclusion –

This project successfully demonstrates the design and implementation of a custom Unix shell integrated with an AI-driven automation system, built entirely in C. Through this work, both traditional shell functionality and intelligent automation were combined to create a modern, context-aware command-line interface.

On the systems side, the shell provided complete support for pipelining, I/O redirection, command chaining, background processes, and job control, illustrating how low-level system calls such as fork(), execvp(), and pipe() interact with the operating system to manage processes and resources.

On the AI side, the integration of a natural-language command interpreter transformed the shell into an intelligent assistant capable of understanding user intent, analyzing execution history, and adapting to command failures. By using contextual information such as past commands, OS environment, and recent errors, the AI component enhanced usability, safety, and adaptability within the terminal environment.

Overall, the project provided deep insights into Unix process management, inter-process communication, terminal handling, and AI-assisted automation. It showcases how merging traditional system programming with

AI reasoning can make command-line systems more accessible, resilient, and intelligent — bridging the gap between manual control and autonomous command execution.

References –

Linux Manual - [man\(1\) - Linux manual page](#)

ChatGPT

StackOverflow

GeeksForGeeks