Autonomous Code Assistant Agent Using LLM, FlowScript, and Multithreaded Job System

1. JobSystemAPI Overview

1.1 JobSystemAPI Functions

Job System Components

- **Constructor (`JobSystemAPI()`)**
    - Initializes the `JobSystemAPI` instance.
    - Creates worker threads based on the system's hardware concurrency.
- **Destructor (`~JobSystemAPI()`)**
    - Destroys the `JobSystemAPI` instance and cleans up resources.
- **Start (`void Start()`)**
    - Initializes the job system if it is not already initialized.
- **Stop (`void Stop()`)**
    - Completes any finished jobs if the job system is initialized.
- **Destroy (`void Destroy()`)**
    - Completes all finished jobs, destroys the job system, and sets the system to `nullptr` to prevent double destruction.
- **CreateJob (`nlohmann::json CreateJob(const char *jobName, nlohmann::json &input)`)**
    - Creates a job with the specified name and input, returning a JSON object with job details.
- **JobStatus (`nlohmann::json JobStatus(std::string &jobID)`)**
    - Retrieves the status of a job based on its ID, returning a JSON object with the job status (e.g., 'never seen', 'queued', 'running', 'completed', 'retired', 'unknown').
- **FinishJob (`nlohmann::json FinishJob(std::string &jobID)`)**
    - Finishes a job based on its ID and returns its output as a JSON object.
- **GetJobTypes (`nlohmann::json GetJobTypes()`)**
    - Retrieves the types of jobs available in the job system, returning them as a JSON object.
- **QueueJob (`void QueueJob(int jobId)`)**
    - Queues a job in the job system based on its ID.

- **RegisterJob** (`void RegisterJob(const char *jobName, std::function<Job *()> jobFactory)`)
    - Registers a new job type in the job system, allowing it to be created and managed by the system.
- **SetDependency** (`void SetDependency(const char *dependentJobName, const char *dependencyJobName)`)
    - Sets a dependency between two jobs, ensuring that one job (dependent) does not start until another job (dependency) is completed.

1.2 Available Jobs within the JobSystem

1.2.1 CustomJob

The `Job` class represents a single job that can be executed by the Job System. It has properties such as `jobID`, `jobStatus`, and `jobChannels`. The important methods of the `Job` class include:

- `Execute()`: Validates and processes the input JSON to extract necessary data, such as a command to be executed. Executes a command in the system's shell using a pipe. This command is specified in the input JSON under the "command" key. The method captures both standard output and standard error of the executed command.
- `JobCompleteCallback()`: Outputs the job's unique ID and its execution results, formatted as a JSON string, to the console.

1.3.2 CompileJob

The `CompileJob` class is a specific type of job that performs compilation of source code. It has a `compileJobOutput` property that stores the output of the compilation process. The important methods of the `CompileJob` class include:

- `Execute()`: Validates and processes the input JSON to extract necessary data, such as a command to be executed. Executes a command in the system's shell using a pipe. This command is specified in the input JSON under the "command" key. Performs the compilation process. The method captures both standard output and standard error of the executed command.
- `JobCompleteCallback()`: Outputs the job's unique ID and its execution results, formatted as a JSON string, to the console.

1.3.3 ParsingJob

The `ParsingJob` class is a specific type of job that performs parsing of the output from the compilation job. It has a `compileJobOutput` property that stores the output of the compilation job, as well as a `parsedErrors` property that stores the parsed error information. The important methods of the `ParsingJob` class include:

- `Execute()`: Performs the parsing process. Utilizes regular expressions to identify and extract different types of errors from the output, including linker text errors, linker errors, and compiler errors. Converts the parsed error information into a JSON array, where each error is represented as a JSON object containing detailed information.
- `JobCompleteCallback()`: Outputs the job's unique ID and its execution results, formatted as a JSON string, to the console.

1.3.4 OutputJob

The `OutputJob` class is a specific type of job that generates JSON output from the parsed error information. It has an `errorInfoVector` property that stores the parsed error information, as well as the necessary mutexes for thread safety. The important methods of the `OutputJob` class include:

- `Execute()`: Generates the JSON output. Opens a JSON file in append mode to add new error information. Adds each error entry to a JSON array, organized by the filepath. In the case of linker errors, these are grouped separately.
- `JobCompleteCallback()`: Outputs the job's unique ID and the JSON-formatted output to the console for verification.

1.3.5 FlowScriptParserJob

The `FlowScriptParserJob` is designed to parse and process FlowScript, a domain-specific language for defining and managing workflows in the system. This job translates the FlowScript into a structured format that can be easily utilized by other components of the system.The important methods of the `FlowScriptParserJob` class include:

- `Execute()`: Initializes a map of job factories to create different types of jobs (like **`CustomJob`**, **`ParsingJob`**, and **`OutputJob`**) based on FlowScript definitions.
  - Tokenizes the FlowScript input using a regular expression-based approach. This step breaks the script into meaningful tokens that can be further processed.
  - Parses the tokenized data to construct a graph representation of the FlowScript. Each node in the graph corresponds to a part of the FlowScript, such as a job or a data element, and contains relevant properties like type, dependencies, and input data.
  - Handles tokenization and parsing errors with appropriate error messages and exits the job if an error occurs.
  - Converts the parsed graph into a JSON format for easier handling and visualization. This JSON output includes details about each node, such as its type, dependencies, and any associated data.

- `JobCompleteCallback()`: Stores the JSON output in the job system or outputs an error message if the job system instance is not available.

2. FlowScript Language Integration

2.1 Introduction to FlowScript

FlowScript is a domain-specific language (DSL) designed to streamline the creation and management of complex workflows, particularly in the context of job execution, data manipulation, and process control. Leveraging a syntax that extends the DOT language, FlowScript is tailored for intuitive design, visualization, and execution of workflows, supporting conditional logic, parallelism, and dynamic data handling.

2.2 Language Design Goals

The primary goal of FlowScript is to provide a clear and expressive DSL for defining workflows in a way that is both human-readable and easily interpreted by machines. Key objectives include:

- **Intuitive Syntax:** Easy to write and understand, even for users with limited programming experience.
- **Flexibility:** Capable of expressing a wide range of workflow patterns, including conditional flows and parallel processing.
- **Visualization:** Easily translatable to visual representations, enhancing understanding and debugging of workflows.

2.3 Language Elements

FlowScript's grammar encompasses several key elements:

- **Data Nodes (Type 0):** Represented as nodes with `shape="circle"`. These nodes hold data objects, often in JSON format, which are used as inputs for jobs.

  Example: `compileJobData[data="{'command', 'clang++ -g -std=c++14 ./Code/automated/*.cpp -o auto_out'}"];`

- **Job Nodes (Type 1):** Defined with `shape="box"`. These nodes represent executable jobs within the workflow.

  Example: `compileJob;`

- **Status Nodes (Type 2):** Indicated by `shape="diamond"`. These nodes represent the status or outcome of a job and can be used for conditional branching.

Example: `compileJobStatus;`

- **Dependencies:** Represented by the `>` operator, indicating the flow of execution and data passing from one node to another.

  Example: `compileJob -> compileJobData;`

3. LLM Integration and FlowScript Implementation

3.1 Overview of LLM and FlowScript Integration

The integration of GPT-3.5, an advanced language model developed by OpenAI, into the system plays a pivotal role in the autonomous generation of FlowScript. FlowScript is a domain-specific language extended from DOT language, primarily used for defining and orchestrating the workflow of various jobs within a custom C++ multithreaded job system. This integration is achieved through a two-step process:

1. **Custom Job Creation for FlowScript Generation:**
   A custom job (`flowscriptGenJob`) is registered and queued in the job system. This job is designed to execute a Node.js script (`flowScriptGen.js`) via the command line, passing the file path as an argument.
2. **Node.js Script for GPT-3.5 Interaction:**
   The Node.js script interacts with the GPT-3.5 model, providing it with a prompt to generate valid FlowScript based on predefined rules and structure. The script ensures that the generated FlowScript adheres to the specific node structure and ordering rules of FlowScript, encompassing different types of nodes (Circle, Box, Diamond) and their interconnections.

3.2 FlowScript Generation Process

- **Main Application Integration (`main.cpp`):**
  The main application parses command line arguments to extract the file path. It then constructs a command string to execute the Node.js script with the file path as a parameter. The job for FlowScript generation is registered, created, and queued in the job system. A delay is introduced to allow time for the FlowScript file (`flowscript.dot`) to be created.
- **Node.js Script (`flowScriptGen.js`):**
  The script employs the OpenAI Node.js library to facilitate the interaction with the GPT-3.5 model. It constructs a detailed prompt that instructs the model to generate a FlowScript based on a given input format. The script handles the entire process of sending the prompt to the model, receiving the response, and writing the generated

FlowScript to a file (`flowscript.dot`).
Key components of the script include:

- **Prompt Generation:** The script dynamically generates a prompt that instructs GPT-3.5 to create a valid FlowScript for a given set of job nodes. This prompt includes rules and examples for the model to follow.
- **OpenAI API Interaction:** The script utilizes the OpenAI API to send the generated prompt to GPT-3.5 and receive the generated FlowScript.
- **File Writing:** The FlowScript response from GPT-3.5 is written to a DOT file (`flowscript.dot`), which is then utilized by the job system.

4. Prompts, Training, and Parsing

4.1 Documentation of LLM Prompts

Initially, I prompted ChatGPT-4 to help me create a prompt for the GPT-3.5 Turbo model.

Initial Prompt to ChatGPT-4 to help create a prompt for GPT-3.5 Turbo:

```I have created a domain specific language called FlowScript, and it is an extension of the DOT programming language. Essentially, FlowScript allows me to define the flow and execution of jobs within a custom C++ multithreaded job system. It follows a very specific set of rules such as node structure and ordering. There are three type of nodes that can be created in FlowScript. The first node type has the shape "circle", and it is used to represent data nodes within the job system. Essentially this is the input for the second type of node. The second type of node is represented with the shape "box". These are the job nodes. These are custom job types that first get registered, then created, and then queued within the job system. The last type of node is the node with shape "diamond" which represents status nodes. These nodes help dictate the flow of execution based on associated label for each status type. All examples of FlowScript require that the file starts off with "digraph" and an opening bracket "{". Then my parser requires that the specific type of nodes are declared. Finally after the status nodes are declared, I create a dependency graph which connects nodes with the "->" operator. Each line represents a different correction. Currently my job system is capable of handling three job types: "compileJob", "compileParseJob", and "parseOutputJob". I will provide you with sample FlowScript that currently works for my job system. I need you to help me create a prompt that will allow me to query an LLM in order to have it generate this exact type of FlowScript for me in this format.

FlowScript:

digraph {
{
node[shape="circle"];

compileJobData[data="{'command', 'clang++ -g -std=c++14 ./Code/automated/*.cpp -o auto_out'}"];
}
{
node[shape="box"];
compileJob;
compileParseJob;
parseOutputJob;
}
{
node[shape="diamond"];
compileJobStatus;
compileParseJobStatus;
}

compileJob -> compileJobData;
compileJobStatus -> compileJob;
compileParseJob -> compileJobStatus;
parseOutputJob -> compileJobStatus;
compileParseJobStatus -> compileParseJob;
parseOutputJob -> compileParseJobStatus;
}```

The 1st Prompt Suggested by ChatGPT-4:

```I have a custom domain-specific language called FlowScript, which is an extension of the DOT programming language. It's designed to define the flow and execution of jobs within a custom C++ multithreaded job system. FlowScript has a very specific set of rules, including node structure and ordering, and supports three types of nodes:

- Circle Nodes: Represent data nodes (inputs for job nodes).
- Box Nodes: Represent job nodes (custom jobs that are registered, created, and queued).
- Diamond Nodes: Represent status nodes (control flow based on associated status labels).

A FlowScript file starts with 'digraph {' and requires specific declarations for each type of node. The nodes are then connected in a dependency graph using the '->' operator.

Here's an example of FlowScript for a job system that handles 'XJob', 'YJob', and 'ZJob':

digraph {
{
node[shape="circle"];
XJobData[data="{'command', 'clang++ -g -std=c++14 ./Code/automated/*.cpp -o auto_out'}"];
}
{
```

node[shape="box"];
XJob;
YJob;
ZJob;
}
{
node[shape="diamond"];
XJobStatus;
YJobStatus;
}

XJob -> XJobData;
XJobStatus -> XJob;
YJob -> XJobStatus;
ZJob -> XJobStatus;
YJobStatus -> YJob;
ZJob -> YJobStatus;
}

Based on these rules and the example, can you generate a FlowScript for a job system handling 'XJob' ("compileJob"), 'YJob' ("compileParseJob"), and 'ZJob' ("parseOutputJob"), ensuring to maintain the structure and node types as described?```

Response from ChatGPT-3.5 Turbo based on 1st Prompt

```Certainly! Here's a FlowScript for a job system that handles 'XJob' ("compileJob"), 'YJob' ("compileParseJob"), and 'ZJob' ("parseOutputJob") while maintaining the structure and node types as described:

```
digraph {
  {
    node[shape="circle"];
    XJobData[data="{'command', 'clang++ -g -std=c++14 ./Code/automated/*.cpp -o auto_out'}"];
  }
  {
    node[shape="box"];
    compileJob;
    compileParseJob;
    parseOutputJob;
  }
  {
    node[shape="diamond"];
    compileJobStatus;
    compileParseJobStatus;
  }
```

compileJob -> XJobData;
compileJobStatus -> compileJob;
compileParseJob -> compileJobStatus;
parseOutputJob -> compileJobStatus;
compileParseJobStatus -> compileParseJob;
parseOutputJob -> compileParseJobStatus;
}

This FlowScript represents a job system with 'compileJob' as 'XJob', 'compileParseJob' as 'YJob', and 'parseOutputJob' as 'ZJob', following the structure and node types you described.```

Attempting a Second Prompt

- Attempting to use a single shot prompt to ensure that the output is only the FlowScript with no accompanying text.

```I have a custom domain-specific language called FlowScript, which is an extension of the DOT programming language. It's designed to define the flow and execution of jobs within a custom C++ multithreaded job system. FlowScript has a very specific set of rules, including node structure and ordering, and supports three types of nodes:

- Circle Nodes: Represent data nodes (inputs for job nodes).
- Box Nodes: Represent job nodes (custom jobs that are registered, created, and queued).
- Diamond Nodes: Represent status nodes (control flow based on associated status labels).

A FlowScript file starts with 'digraph {' and requires specific declarations for each type of node. The nodes are then connected in a dependency graph using the '->' operator.

Here's an example of FlowScript for a job system that handles 'XJob', 'YJob', and 'ZJob':

Example Input:
'XJob', 'YJob', 'ZJob'

Example Output:
digraph {
{
node[shape="circle"];
XJobData[data="{'command', 'clang++ -g -std=c++14 ./Code/automated/.cpp -o auto_out'}"];*
}
{
node[shape="box"];
XJob;
YJob;
ZJob;

}
{
node[shape="diamond"];
XJobStatus;
YJobStatus;
}

XJob -> XJobData;
XJobStatus -> XJob;
YJob -> XJobStatus;
ZJob -> XJobStatus;
YJobStatus -> YJob;
ZJob -> YJobStatus;
}

End Example

Your Task:
'compileJob', 'compileParseJob', 'parseOutputJob'```

Response from ChatGPT-3.5 for the 2nd Prompt

```To create FlowScript for the given jobs 'compileJob', 'compileParseJob', and 'parseOutputJob', you can follow a similar structure as the example you provided. Here's the FlowScript for these jobs:

```flowscript
digraph {
  {
    node[shape="circle"];
    CompileJobData[data="{'inputFiles': ['file1.cpp', 'file2.cpp'], 'outputFile': 'compiled.out'}"];
    CompileParseJobData[data="{'inputFile': 'compiled.out', 'parsedFile': 'parsed.txt'}"];
    ParseOutputJobData[data="{'inputFile': 'parsed.txt'}"];
  }
  {
    node[shape="box"];
    CompileJob;
    CompileParseJob;
    ParseOutputJob;
  }
  {
    node[shape="diamond"];
    CompileJobStatus;
    CompileParseJobStatus;
```
```

```
    ParseOutputJobStatus;
  }

  CompileJob -> CompileJobData;
  CompileParseJob -> CompileParseJobData;
  ParseOutputJob -> ParseOutputJobData;
  CompileJobStatus -> CompileJob;
  CompileParseJobStatus -> CompileParseJob;
  ParseOutputJobStatus -> ParseOutputJob;

  CompileParseJob -> CompileJobStatus;
  ParseOutputJob -> CompileParseJobStatus;
}
```

In this FlowScript, I've defined three jobs ('CompileJob', 'CompileParseJob', and 'ParseOutputJob') along with their associated data nodes and status nodes. The dependencies between these nodes are also specified using the '->' operator, following the specific rules and structure of FlowScript.```

3rd Prompt

- Unfortunately, the 2nd prompt did not output how I wanted it to, so I am modifying how I am asking for the model to output.

```*I have a custom domain-specific language called FlowScript, which is an extension of the DOT programming language. It's designed to define the flow and execution of jobs within a custom C++ multithreaded job system. FlowScript has a very specific set of rules, including node structure and ordering, and supports three types of nodes:*

*- Circle Nodes: Represent data nodes (inputs for job nodes).*
*- Box Nodes: Represent job nodes (custom jobs that are registered, created, and queued).*
*- Diamond Nodes: Represent status nodes (control flow based on associated status labels).*

*A FlowScript file starts with 'digraph {' and requires specific declarations for each type of node. The nodes are then connected in a dependency graph using the '->' operator.*

*Your task is to generate a FlowScript for a job system handling 'XJob' ("compileJob"), 'YJob' ("compileParseJob"), and 'ZJob' ("parseOutputJob") based on these rules and the example. Ensure to maintain the structure and node types as described and return only the generated FlowScript with no accompanying explanation.*

*Example Input:*
*'XJob', 'YJob', 'ZJob'*

*Example Output:*
```

digraph {
{
node[shape="circle"];
XJobData[data="{'command', 'clang++ -g -std=c++14 ./Code/automated/.cpp -o auto_out'}"];*
}
{
node[shape="box"];
XJob;
YJob;
ZJob;
}
{
node[shape="diamond"];
XJobStatus;
YJobStatus;
}

XJob -> XJobData;
XJobStatus -> XJob;
YJob -> XJobStatus;
ZJob -> XJobStatus;
YJobStatus -> YJob;
ZJob -> YJobStatus;
}

End Example

Your Task:
'compileJob', 'compileParseJob', 'parseOutputJob'```

Response from ChatGPT-3.5 for the 3rd Prompt:

```Here is the FlowScript generated for 'compileJob', 'compileParseJob', and 'parseOutputJob' based on the provided rules and structure:

```plaintext
digraph {
{
node[shape="circle"];
compileJobData[data="{'command', 'clang++ -g -std=c++14 ./Code/automated/.cpp -o auto_out'}"];*
}
{
node[shape="box"];
compileJob;

compileParseJob;
parseOutputJob;
}
{
node[shape="diamond"];
compileJobStatus;
compileParseJobStatus;
}

compileJob -> compileJobData;
compileJobStatus -> compileJob;
compileParseJob -> compileJobStatus;
parseOutputJob -> compileJobStatus;
compileParseJobStatus -> compileParseJob;
parseOutputJob -> compileParseJobStatus;
}

This FlowScript defines the structure and connections for 'compileJob', 'compileParseJob', and 'parseOutputJob' in the specified format.```

4th Prompt

- The 3rd prompt was unsuccessful, so I am trying a 4th prompt with more explicit language.
- Adjusting the guidelines to try and ensure that it returns nothing besides the FlowScript.
- Making sure FlowScript also returns desired data input for custom job.

```Do not include any explanation and return an answer without code block. I have a custom domain-specific language called FlowScript, which is an extension of the DOT programming language. It's designed to define the flow and execution of jobs within a custom C++ multithreaded job system. FlowScript has a very specific set of rules, including node structure and ordering, and supports three types of nodes:

- Circle Nodes: Represent data nodes (inputs for job nodes).
- Box Nodes: Represent job nodes (custom jobs that are registered, created, and queued).
- Diamond Nodes: Represent status nodes (control flow based on associated status labels).

A FlowScript file starts with 'digraph {' and requires specific declarations for each type of node. The nodes are then connected in a dependency graph using the '->' operator.

Your task is to generate a FlowScript for a job system handling 'XJob' ("compileJob"), 'YJob' ("compileParseJob"), and 'ZJob' ("parseOutputJob") based on these rules and the example. Ensure to maintain the structure and node types as described and return only the generated FlowScript without code block.

Example Input:
'XJob', 'XJobData: data={'command', 'clang++ -g -std=c++14 ./Code/automated/.cpp -o auto_out'}','YJob', 'ZJob'*

Example Output:
```
digraph {
{
node[shape="circle"];
XJobData[data="{'command', 'clang++ -g -std=c++14 ./Code/automated/*.cpp -o auto_out'}"];
}
{
node[shape="box"];
XJob;
YJob;
ZJob;
}
{
node[shape="diamond"];
XJobStatus;
YJobStatus;
}

XJob -> XJobData;
XJobStatus -> XJob;
YJob -> XJobStatus;
ZJob -> XJobStatus;
YJobStatus -> YJob;
ZJob -> YJobStatus;
}
```

End Example

Your Task:
'compileJob', 'compileJobData: data={'command', 'clang++ -g -std=c++14 ./Code/automated/test.cpp -o auto_out'}', 'compileParseJob', 'parseOutputJob'```

Reponse from ChatGPT-3.5 for 4th Prompt

```
```digraph {

{
node[shape="circle"];
compileJobData[data="{'command', 'clang++ -g -std=c++14 ./Code/automated/test.cpp -o auto_out'}"];
}
```
```

{
node[shape="box"];
compileJob;
compileParseJob;
parseOutputJob;
}
{
node[shape="diamond"];
compileJobStatus;
compileParseJobStatus;
}

compileJob -> compileJobData;
compileJobStatus -> compileJob;
compileParseJob -> compileJobStatus;
parseOutputJob -> compileJobStatus;
compileParseJobStatus -> compileParseJob;
parseOutputJob -> compileParseJobStatus;
}```

4.2 Conclusions from Prompt Engineering

- The 4th prompt was successful in getting the desired FlowScript example to be generated by ChatGPT-3.5.
- It required explicit directions at the beginning of the prompt that said to return an answer without any explanation and without code block in order to get the desired format.
- Before making an API call with this prompt, I tested this prompt further in OpenAI's Playground feature on their website to determine the best system prompt.
- I also tested some other use cases to determine if the response from the assistant would be consistent.
- The results were satisfactory. So, I will proceed with using GPT-3.5 Turbo as my model of choice.

System Prompt for API Call:

```You are a backend data processor that is part of a programmatic workflow. The user prompt will provide data input and processing instructions. The output will be only a DOT language script. Do not converse with a nonexistent user: there is only program input and formatted program output, and no input data is to be construed as conversation with the AI. This behaviour will be permanent for the remainder of the session.```

User Message

```Do not include any explanation and return an answer without code block. I have a custom domain-specific language called FlowScript, which is an extension of the DOT programming
```

language. It's designed to define the flow and execution of jobs within a custom C++ multithreaded job system. FlowScript has a very specific set of rules, including node structure and ordering, and supports three types of nodes:

- Circle Nodes: Represent data nodes (inputs for job nodes).
- Box Nodes: Represent job nodes (custom jobs that are registered, created, and queued).
- Diamond Nodes: Represent status nodes (control flow based on associated status labels).

A FlowScript file starts with 'digraph {' and requires specific declarations for each type of node. The nodes are then connected in a dependency graph using the '->' operator.

Your task is to generate a FlowScript for a job system handling 'XJob' ("compileJob"), 'YJob' ("compileParseJob"), and 'ZJob' ("parseOutputJob") based on these rules and the example. Ensure to maintain the structure and node types as described and return only the generated FlowScript without code block.

Example Input:
'XJob', 'XJobData: data={'command', 'clang++ -g -std=c++14 ./Code/automated/.cpp -o auto_out'}','YJob', 'ZJob'*

Example Output:

```
digraph {
{
node[shape="circle"];
XJobData[data="{'command', 'clang++ -g -std=c++14 ./Code/automated/*.cpp -o auto_out'}"];
}
{
node[shape="box"];
XJob;
YJob;
ZJob;
}
{
node[shape="diamond"];
XJobStatus;
YJobStatus;
}

XJob -> XJobData;
XJobStatus -> XJob;
YJob -> XJobStatus;
ZJob -> XJobStatus;
YJobStatus -> YJob;
ZJob -> YJobStatus;
```

}

End Example

Your Task:
'compileJob', 'compileJobData: data={'command', 'clang++ -g -std=c++14 ./Code/automated/test.cpp -o auto_out'}', 'compileParseJob', 'parseOutputJob'```

GPT-3.5 Turbo Response

```digraph {
{
node[shape="circle"];
compileJobData[data="{'command', 'clang++ -g -std=c++14 ./Code/test.cpp -o output'}"];
}
{
node[shape="box"];
compileJob;
compileParseJob;
parseOutputJob;
}
{
node[shape="diamond"];
compileJobStatus;
compileParseJobStatus;
}

compileJob -> compileJobData;
compileJobStatus -> compileJob;
compileParseJob -> compileJobStatus;
parseOutputJob -> compileJobStatus;
compileParseJobStatus -> compileParseJob;
parseOutputJob -> compileParseJobStatus;
}```

## Further Testing:

**User Message**

```'flowScriptParserJob', 'flowScriptGenJob', 'flowScriptOutputJob'```

GPT-3.5 Turbo Response

~~~
```digraph {
{
node[shape="circle"];
flowScriptParserJobData[data="{'command', './parse_script.py input.txt output.txt'}"];
}
{
node[shape="box"];
flowScriptParserJob;
flowScriptGenJob;
flowScriptOutputJob;
}
{
node[shape="diamond"];
flowScriptParserJobStatus;
flowScriptGenJobStatus;
}

flowScriptParserJob -> flowScriptParserJobData;
flowScriptParserJobStatus -> flowScriptParserJob;
flowScriptGenJob -> flowScriptParserJobStatus;
flowScriptOutputJob -> flowScriptParserJobStatus;
flowScriptGenJobStatus -> flowScriptGenJob;
flowScriptOutputJob -> flowScriptGenJobStatus;
}```
~~~

5. Correcting Code with LLM

This section documents the methodology of utilizing Language Model (LLM) for generating code change suggestions and applying these corrections to source files. The system employs a series of Node.js scripts and intermediate files, combined with polling methods, to ensure seamless and sequential job completion.

5.1 Use of Intermediary Files for Message Passing

- **Error Report File:** The system first generates an `error_report.json` file containing compiler errors and warnings. This file is used as a source of information for the LLM to understand the errors in the code.
- **FlowScript File (`flowscript.dot`):** This file is generated by the LLM and contains the FlowScript code. It is parsed by the system to determine the workflow of jobs, such as compilation, parsing, and output generation.
- **Corrected Code File (`corrected_code.json`):** Generated by the `gptCall.js` script, this file contains JSON-formatted corrections suggested by the LLM based on the errors present in the `error_report.json`.

- **Code Change Descriptions (`code_change_descriptions.txt`):** The `codeCorrection.js` script creates this file, detailing the specific changes made to the source code, facilitating easy tracking and review of modifications.

5.2 Polling Methods for Job Completion

- **Job Registration and Queuing:** The system first registers and queues various jobs from the parsed FlowScript. This is achieved through a function `registerAndQueueJobs`, which dynamically handles the creation and dependency setting of these jobs based on the FlowScript structure.
- **Checking for Compilation Errors:** The system employs a function `hasCompilationErrors` to check whether the `error_report.json` contains compilation errors. This determines if further action, such as calling the LLM for suggestions, is necessary.
- **Polling for File Updates:** To ensure that jobs have completed their tasks, the system implements a polling mechanism. It regularly checks if certain files (`corrected_code.json`, `correction_history.json`, `code_change_descriptions.txt`) have been updated, indicating that the corresponding jobs have finished processing. This method includes a timeout feature to prevent indefinite waiting.

5.3 LLM Integration for Code Correction

- **Generating LLM Prompts (`gptCall.js`):** This script reads the `error_report.json` and formulates a prompt for the LLM. It instructs the LLM to analyze the errors and suggest corrections in a structured JSON format. The script then sends this prompt to the OpenAI GPT-3.5 model and retrieves the suggested corrections.
- **Applying Code Corrections (`codeCorrection.js`):** Upon receiving the corrected code in JSON format, this script applies the suggested changes to the actual source files. It reads the corrections and edits the source code files accordingly, also updating the `code_change_descriptions.txt` file with descriptions of the changes.

5.4 Sequential Execution and Error Handling

The system's design ensures sequential execution of jobs and effective error handling. After parsing the FlowScript, it enters a loop to process the jobs, constantly checking for compilation errors and updating the source code based on LLM suggestions. In case of errors during script execution or file handling, the system outputs appropriate error messages and halts further processing if necessary.

6. Complete System Run-Through Autonomous Code Assistant Agent

6.1 Initializing Autonomous Code Assistant Agent

The initialization of the Autonomous Code Assistant Agent involves setting up the environment and launching the main application. Here's a step-by-step guide:

6.1.1 Environment Setup

1. **Add `.env` File:** A `.env` file is created to store environment variables, including the OpenAI API Key. This key is essential for the system to interact with the GPT-3.5 model.
2. **Make Compile Command:** The system uses a makefile to compile the C++ code. The `make compile` command compiles the necessary source files and prepares the agent for execution.
3. **NPM Command:** `npm install` will install necessary libraries, particularly those required for Node.js scripts.
4. **Arguments for Main Function:** The main function of the application is executed with arguments that specify the path to the source files to be processed. For example: `./app ./Code/automated/geometry.cpp`.

6.1.2 Registration of FlowScript Generation Job

The system first generates a prompt for the GPT-3.5 model to create FlowScript. This prompt includes instructions and rules specific to the FlowScript language and the desired job execution flow.

Sample output:
./app ./Code/automated/geometry.cpp
Registering FlowScript Generation job

Creating FlowScript Generation Job: {
  "dependencies": [],
  "jobId": 0,
  "status": "Job created"
}
Queuing flowscriptGenJob:

6.1.3 Creation and Queuing of FlowScript Generation Job

The GPT-3.5 model, upon receiving the prompt, generates the FlowScript, which is then captured by the system and stored in the file (`flowscript.dot`). This FlowScript dictates the sequence of jobs and their dependencies, tailored to the specific requirements of the code in question.
`flowscript.dot` contents:

```
digraph {
{
node[shape="circle"];
```

```
compileJobData[data="{'command', 'clang++ -g -std=c++14 ./Code/automated/geometry.cpp
-o auto_out'}"];
}
{
node[shape="box"];
compileJob;
compileParseJob;
parseOutputJob;
}
{
node[shape="diamond"];
compileJobStatus;
compileParseJobStatus;
}

compileJob -> compileJobData;
compileJobStatus -> compileJob;
compileParseJob -> compileJobStatus;
parseOutputJob -> compileJobStatus;
compileParseJobStatus -> compileParseJob;
parseOutputJob -> compileParseJobStatus;
}
```

6.2 FlowScript Parser Job Output and Explanation

Once the FlowScript is generated, the system proceeds to parse it:

1. **Parsing FlowScript:** A custom job (`flowscriptJob`) is created to parse the generated FlowScript. This job reads the FlowScript from the file (`flowscript.dot`) and translates it into a structured format (JSON) that the system can understand and act upon.

Sample output:

```
…
Creating FlowScript Parse Job: {
   "dependencies": [],
   "jobId": 1,
   "status": "Job created"
}
Queuing FlowScript Parse Job with ID: 1
...
Finishing Job 1 with result: {
   "message": "Job #1 finished successfully",
   "status": "success"
```

}

Enqueuing Jobs from FlowScript Graph!
…

Upon completion of the FlowScript parsing job, the system proceeds with the following steps:

1. **Execution of Compiled Jobs:** The system dynamically registers and queues various jobs such as `compileJob`, `compileParseJob`, and `parseOutputJob`, which are essential for processing the source code. These jobs are created with unique IDs and their dependencies are set accordingly.
    a. **Sample Output:**

        Registered job: compileJob
        Created job: compileJob with ID: 2
        Set dependency for compileJob on compileJobData
        Queued job: compileJob
        ...
        Opening Error Report JSON file: ./Data/error_report.json

        Compilation errors present.
        …

2. **Polling for Job Completion and Error Checking:** The system checks for compilation errors by opening the `error_report.json` file. If errors are present, it registers and queues additional jobs (`gptCallJob` and `codeCorrectionJob`) for error correction. The system also continuously polls for the completion of jobs, particularly the `gptCallJob`. This is evident from the repeated output lines indicating polling status. A timeout mechanism is implemented to prevent indefinite waiting.
    a. **Sample Output:**

        Polling for gptCall job completion...
        ...
        Timeout reached while waiting for gptCallJob.
        Queuing codeCorrection Job:
        …
        Polling for codeCorrection job completion...
        code_change_descriptions.txt file has been modified, breaking out of loop
        …

3. **Applying Code Corrections:** Once the `gptCallJob` completes, the system queues the `codeCorrectionJob`. The system again employs polling to confirm the completion of this job.

4. **Final Check for Compilation Errors:** After applying corrections, the system performs another check for compilation errors. The absence of errors indicates the successful correction of the code. If there is a linker error, the program also ends gracefully and notifies the user of a linker error.
    a. **Sample Output:**

        …
        Opening Error Report JSON file: ./Data/error_report.json

        No compilation errors (file is empty or contains 'null').
        ...

6.3 Execution Output and Job Results

The system's output provides detailed information about each step and the results of various jobs:

- **FlowScript Generation Output:** The FlowScript generated by the system is outputted in the DOT format and written to the `flowscript.dot` file.
- **Compilation and Parsing Job Results:** These jobs output details about the compilation process, including error descriptions and file paths.
- **GPT-3.5 Job Results:** The `gptCallJob` successfully generates corrections based on the compilation errors and outputs them in JSON format.
- **Code Correction Job Results:** This job applies the suggested corrections to the source code and outputs a confirmation message.

6.3.1 Agent Produced Report and Code

During the execution of the `gptCallJob`, the agent produces several key outputs which are critical for the code correction process:

6.3.2 `corrected_code.json` File Output:

The `gptCall` job results in the creation of the `corrected_code.json` file. This file contains the suggested corrections for the source code, formatted as JSON objects. Each object includes details such as the file path, line number, column number, a description of the code change, and the corrected code snippet.

Example Output from `corrected_code.json`:


```
{
  "./Code/automated/geometry.cpp": [
    {
      "lineNumber": 5,
      "columnNumber": 39,
      "codeChangeDescription": "Add a semicolon at the end of the statement.",
```

```
            "correctedCodeSnippet": [
                "...",
                "}"
            ]
        },
        ...
    ]
}
```

6.3.3 `correction_history.json` File Output:

In addition to the corrected code, the `gptCall` job also produces the `correction_history.json` file. This file logs the history of corrections suggested by the system, including the specific error signature and the corresponding suggestion.

Example Output from `correction_history.json`:

```
[
        {
        "errorSignature": "./Code/automated/geometry.cpp:5:39",
        "suggestion": "Add a semicolon at the end of the statement."
        },
        ...
]
```

6.3.4 `code_change_descriptions.txt` File Output:

Finally, the `codeCorrection` job compiles the `code_change_descriptions.txt` file, which provides a textual summary of the changes applied to the code. This file is an essential record for tracking the modifications made during the correction process.

Example Output from `code_change_descriptions.txt`:

File: ./Code/automated/geometry.cpp, Line: 5, Column: 39, Description: Add a semicolon at the end of the statement.

...

6.4 Demonstrating Compile System Output

To demonstrate the Compile System within the Autonomous Code Assistant Agent including feedback from the LLM, let's consider an example where we have a source code file that needs to be compiled and parsed for errors. We will use the Job System to perform these tasks.

6.4.1 Example Code Demonstrating Error Output

Here is an example code that demonstrates the usage of the Compile System:

The following code will produce multiple errors:

```
#include <iostream>

double areaCircle(double radius)

{

   double area = 3.14 *radius *radius double diff = sub(radius, 5.0);

   return area

}



int main(void)

{

   double radius = 21.0;

   double area = areaCircle(radus);



   st::cout << "The area of circle with radius " << radius << " is: " << area <<
std::endl;

   return 0;

}
```

6.4.2 Raw Errors and Warnings

The following is the raw error output:

```
clang++ -g -std=c++14 ./Code/automated/geometry.cpp -o auto_out
./Code/automated/geometry.cpp:5:39: error: expected ';' at end of declaration
        double area = 3.14 *radius *radius double diff = sub(radius, 5.0);
                                  ^
                                  ;
./Code/automated/geometry.cpp:5:54: error: use of undeclared identifier 'sub'
```

```
        double area = 3.14 *radius *radius double diff = sub(radius, 5.0);
                                          ^
./Code/automated/geometry.cpp:6:16: error: expected ';' after return statement
        return area
        ^
        ;
./Code/automated/geometry.cpp:12:30: error: use of undeclared identifier 'radus'; did you mean 'radius'?
        double area = areaCircle(radus);
                      ^~~~~
                      radius
./Code/automated/geometry.cpp:11:12: note: 'radius' declared here
        double radius = 21.0;
        ^
./Code/automated/geometry.cpp:14:5: error: use of undeclared identifier 'st'
        st::cout << "The area of circle with radius " << radius << " is: " << area << std::endl;
        ^
5 errors generated.
```

6.4.3 JSON Output Job Output to `error_report.json`

The JSON output job within this implementation of the Job System provides information about the errors and warnings that occur during the compilation process of some C++ code that is executed from a command within the makefile.

```
{
   "./Code/automated/geometry.cpp": [
       {
           "lineNumber": 5,
           "columnNumber": 39,
           "errorDescription": "expected ';' at end of declaration",
           "codeSnippet": [
               "double areaCircle(double radius)",
               "{",
               "    double area = 3.14 *radius *radius double diff = sub(radius, 5.0);",
```

```
                "    return area",
                "}"
            ]
        },
        {
            "lineNumber": 5,
            "columnNumber": 54,
            "errorDescription": "use of undeclared identifier 'sub'",
            "codeSnippet": [
                "double areaCircle(double radius)",
                "{",
                "    double area = 3.14 *radius *radius double diff = sub(radius,
5.0);",
                "    return area",
                "}"
            ]
        },
        {
            "lineNumber": 6,
            "columnNumber": 16,
            "errorDescription": "expected ';' after return statement",
            "codeSnippet": [
                "{",
                "    double area = 3.14 *radius *radius double diff = sub(radius,
5.0);",
```

```
                "    return area",
                "}",
                ""
            ]
        },
        {
            "lineNumber": 12,
            "columnNumber": 30,
            "errorDescription": "use of undeclared identifier 'radus'; did you mean 'radius'?",
            "codeSnippet": [
                "{",
                "    double radius = 21.0;",
                "    double area = areaCircle(radus);",
                "",
                "    st::cout << \"The area of circle with radius \" << radius << \" is: \" << area << std::endl;"
            ]
        },
        {
            "lineNumber": 14,
            "columnNumber": 5,
            "errorDescription": "use of undeclared identifier 'st'",
            "codeSnippet": [
                "    double area = areaCircle(radus);",
```

```
                "",

                "    st::cout << \"The area of circle with radius \" << radius << \"
is: \" << area << std::endl;",

                "    return 0;",

                "}"

            ]

        }

    ]

}
```

- Each file object contains an array of JSON objects related to the individual errors and warnings.
- The line number of the error is parsed and outputted as “lineNumber”.
- The column number of the error is parsed and outputted as “columnNumber”.
- The error message itself is parsed and outputted as “errorDescription”.
- The code snippet, containing a max of two lines before and after the error, is also outputted as “codeSnippet”.

The user is presented with enough information to determine where the error occurs within their source files, as well as the description of the compilation error.

6.4.4 Corrected Code File Output

```
{
   "./Code/automated/geometry.cpp": [
       {
           "lineNumber": 5,
           "columnNumber": 39,
           "codeChangeDescription": "Add a semicolon at the end of the declaration.",
           "correctedCodeSnippet": [
               "double areaCircle(double radius)",
               "{",
               "    double area = 3.14 *radius *radius; double diff = sub(radius,
5.0);",
               "    return area",
               "}"
           ]
       },
       {
```

```
            "lineNumber": 5,
            "columnNumber": 54,
            "codeChangeDescription": "Declare 'sub' function or include its definition.",
            "correctedCodeSnippet": [
                "double areaCircle(double radius)",
                "{",
                "    double area = 3.14 *radius *radius; double diff = sub(radius, 5.0);",
                "    return area",
                "}"
            ]
        },
        {
            "lineNumber": 6,
            "columnNumber": 16,
            "codeChangeDescription": "Add a semicolon after the return statement.",
            "correctedCodeSnippet": [
                "{",
                "    double area = 3.14 *radius *radius; double diff = sub(radius, 5.0);",
                "    return area;",
                "}",
                ""
            ]
        },
        {
            "lineNumber": 12,
            "columnNumber": 30,
            "codeChangeDescription": "Correct the spelling of 'radus' to 'radius'.",
            "correctedCodeSnippet": [
                "{",
                "    double radius = 21.0;",
                "    double area = areaCircle(radius);",
                "",
                "    std::cout << \"The area of circle with radius \" << radius << \" is: \" << area << std::endl;"
            ]
        },
        {
            "lineNumber": 14,
            "columnNumber": 5,
```

```
            "codeChangeDescription": "Use the correct namespace 'std' for 'cout'.",
            "correctedCodeSnippet": [
                "    double area = areaCircle(radius);",
                "",
                "    std::cout << \"The area of circle with radius \" << radius << \"
is: \" << area << std::endl;",
                "    return 0;",
                "}"
            ]
        }
    ]
}
```

6.4.5 Code Correction History File Output

```
[
    {
        "errorSignature": "./Code/automated/geometry.cpp:5:39",
        "suggestion": "Add a semicolon at the end of the declaration."
    },
    {
        "errorSignature": "./Code/automated/geometry.cpp:5:54",
        "suggestion": "Declare 'sub' function or include its definition."
    },
    {
        "errorSignature": "./Code/automated/geometry.cpp:6:16",
        "suggestion": "Add a semicolon after the return statement."
    },
    {
        "errorSignature": "./Code/automated/geometry.cpp:12:30",
        "suggestion": "Correct the spelling of 'radus' to 'radius'."
    },
    {
        "errorSignature": "./Code/automated/geometry.cpp:14:5",
        "suggestion": "Use the correct namespace 'std' for 'cout'."
    }
]
```

6.4.6 Code Change Descriptions Text File Output

File: ./Code/automated/geometry.cpp, Line: 5, Column: 39, Description: Add a semicolon at the end of the declaration.
File: ./Code/automated/geometry.cpp, Line: 5, Column: 54, Description: Declare 'sub' function or include its definition.
File: ./Code/automated/geometry.cpp, Line: 6, Column: 16, Description: Add a semicolon after the return statement.
File: ./Code/automated/geometry.cpp, Line: 12, Column: 30, Description: Correct the spelling of 'radus' to 'radius'.
File: ./Code/automated/geometry.cpp, Line: 14, Column: 5, Description: Use the correct namespace 'std' for 'cout'.

Console output at this moment:

```
…
Opening Error Report JSON file: ./Data/error_report.json

Compilation errors present.
Creating FlowScript Parse Job: {
        "dependencies": [],
        "jobId": 7,
        "status": "Job created"
}
Queuing FlowScript Parse Job with ID: 7
…
```

6.4.7 Re-running the Compile FlowScript to Check if Errors Fixed

Output:

```
…
Enter job ID to finish a specific job:
7
Finishing Job 7 with result: {
        "message": "Job #7 finished successfully",
        "status": "success"
}

Enqueuing Jobs from FlowScript Graph!

Registered job: compileJob
Registered job: compileParseJob
Registered job: parseOutputJob
Created job: compileJob with ID: 8
Dependency job factory not found for job: compileJobData
```

Set dependency for compileJob on compileJobData
Queued job: compileJob
Created job: compileParseJob with ID: 9
Set dependency for compileParseJob on compileJob
Created job: parseOutputJob with ID: 10
Set dependency for parseOutputJob on compileJob
Set dependency for parseOutputJob on compileParseJob
Opening Error Report JSON file: ./Data/error_report.json

No compilation errors (file is empty or contains 'null').

No more compilation errors. Skipping gptCallJob.

Opening Error Report JSON file: ./Data/error_report.json

No compilation errors (file is empty or contains 'null').

Execution complete!
Destroying the Job System.
…

6.7 Final State and Cleanup

After the code correction process, the system performs a final check for compilation errors and proceeds with cleanup:

Example Output for Final Check and Cleanup:
Opening Error Report JSON file: ./Data/error_report.json

No compilation errors (file is empty or contains 'null').

Execution complete!
Destroying the Job System.
…

- This output confirms the successful completion of the code correction process and the absence of any further compilation errors, signifying the effective functioning of the Autonomous Code Assistant Agent.

6.8 Video Demonstration

A video demonstration is available in the *GitHub* repository under the **Report** section:
https://github.com/HuMIn-Game-Lab/final-autonomous-code-assistant-agent-vkatvell/tree/main/Report

7. Conclusion and Future Directions

7.1 Project Summary

The Autonomous Code Assistant Agent is a system that automates code compilation and debugging by integrating GPT-3.5 with a custom job system. Its ability to autonomously identify and correct code errors, manage dependencies, and process FlowScript showcases a potential way to reduce manual coding efforts. The system's efficient synchronization of various components through intermediary files and polling methods help with its effectiveness in automating certain complex programming workflows.

7.2 Future Work

Some possible areas that I am interested in taking this project include:

7.2.1 Direct Integration with JavaScript and Web Interface Development:

One of the future enhancements involves the direct integration of the C++ library within a JavaScript function. This integration would allow for more seamless interactions between the C++ and JavaScript components, potentially opening avenues for creating a user-friendly web interface. A web-based interface would make the system more accessible, providing users with an intuitive platform for managing and observing the code correction process.

7.2.2 Utilization of Web Workers:

To further improve the handling of asynchronous tasks and enhance the system's performance, the implementation of web workers is being considered. Web workers would allow for background processing of tasks without blocking the main execution thread, thus improving the efficiency and responsiveness of the system, especially in a web environment.

7.2.3 Updating Callback Mechanisms:

Replacing the current polling methods with more sophisticated callback mechanisms is another area of interest. This change would enhance the system's responsiveness and reduce the overhead associated with continuous polling. Callbacks would allow the system to react to events or job completions more dynamically, leading to a more efficient and streamlined process flow.

7.2.4 Moving Away from Intermediary Files:

The current reliance on intermediary files for process and job communication is a point of optimization. Future versions of the system may explore alternative communication methods, such as direct data passing or the use of shared memory. This shift would not only simplify the system's architecture but also potentially increase its performance by reducing the reliance on file I/O operations.

8. References

8.1 AI Code Generation & Debugging Tools

- *OpenAI GPT-3.5 Turbo*: Used for generating valid FlowScript and suggesting code changes for parsed compiler errors.
- *OpenAI ChatGPT*: Used for generating code snippets and providing programming-related assistance.
- *Google BARD*: Utilized for debugging and troubleshooting code issues.

8.2 Third-party Libraries

- *JSON for Modern C++*. Parsing Library: nlohmann/json. Available at: github.com/nlohmann/json.
- *OpenAI Node API Library*: For API calls to OpenAI's GPT-3.5 Turbo model. Available at: github.com/openai/openai-node.

8.3 Academic Resources

- *Job System (Dr. Clark)*: The project is built on top of the Job System introduced in CS5393 by Dr. Clark.