

Final Lab Report

Prompt Utilized for generating FlowScript Pipeline

The first prompt I tried was only describing the task I wanted to perform since I assumed that maybe the LLM already knows the Dot programming language structure:

```
Write FlowScript, a new programming language based on the Dot programming language to execute three jobs in the following order: compile, parse_file, and output_to_file.
```

Mistra Instruct Output:

Here is an example of how you could write FlowScript code that executes these three jobs in the specified order:

```
flowscript
// Compile job
var compiledCode = compile("path/to/your/code.js");
// Parse file job
var parsedData = parse_file(compiledCode, "path/to/input/file.json");
// Output to file job
output_to_file(parsedData, "path/to/output/file.json");
```

In this example, the `compile()` function takes a path to your code as an argument and returns the compiled JavaScript code. The `parse_file()` function takes the compiled code and a file path as arguments and parses the input file into a data structure. Finally, the `output_to_file()` function takes the parsed data and a file path as arguments and writes the output to a new file at that location.

However, looking at the output it produced it was noticeable that the LLM didn't know about it (it is possible that wasn't trained on this topic). Therefore, I added a single shot training to this output to see if the LLM could identify it better:

```
Write FlowScript, a new programming language based on the Dot programming language to execute three jobs in the following order: compile, parse_file, and output_to_file.
```

```
Example FlowScript to execute three jobs in the following order: job1, job2, and job3:
```

```
digraph jobs {
    input -> job1 -> job2 -> job3 -> output
}
```

Mistra Instruct Output:

Here's an example of how you could write a FlowScript program that executes three jobs in the order specified:

```
javascript
const compile = require('compile');
const parse_file = require('parse_file');
const output_to_file = require('output_to_file');
// Compile the code
compile('code.js', 'compiled_code.js')
    .then(() => {
        // Parse the compiled code
        parse_file('compiled_code.js')
            .then((parsedCode) => {
                // Output the parsed code to a file
                output_to_file(parsedCode, 'output.txt');
            });
    })
```

Still, the LLM couldn't correctly resolve the task. Also, it added an explanation (which is not required for the task I am trying to accomplish), so I added a sentence stating that I only want the FlowScript code in the output response:

Write FlowScript, a new programming language based on the Dot programming language to execute three jobs in the following order: compile, parse_file, and output_to_file.

Example FlowScript to execute three jobs in the following order: job1, job2, and job3:

```
digraph jobs {
    input -> job1 -> job2 -> job3 -> output
}
```

Don't provide an explanation, only write the FlowScript.

Mistra Instruct Output:

```
flowscript
input -> job1 -> job2 -> job3 -> output
```

This was a cleaner solution, however, it didn't solve the task, instead, it solved the example. Therefore I changed the ordering and the wording for the main prompt and the example to see if this would clarify the task for the LLM:

Assume FlowScript is a new programming language based on the Dot programming language. An example of FlowScript code to execute three jobs in order: "job1", "job2", and "job3", is the following :

```
digraph jobs {
    input -> job1 -> job2 -> job3 -> output
}
```

Now, write FlowScript code to execute three jobs in the following order: "compile", "parse_file", and "output_to_file".
Don't provide an explanation, only write the FlowScript.

Mistra Instruct Output:

```
flowscript
digraph jobs {
    input -> compile -> parse_file -> output_to_file -> output
}
```

This output was the one expected, however, it included the word "flowscript" before the code, which could break the FlowScript code. Therefore, I simply added a last sentence indicating I didn't want the "flowscript" to be included in the code.

Assume FlowScript is a new programming language based on the Dot programming language. An example of FlowScript code to execute three jobs in order: "job1", "job2", and "job3", is the following :

```
digraph jobs {
    input -> job1 -> job2 -> job3 -> output
}
```

Now, write FlowScript code to execute three jobs in the following order: "compile", "parse_file", and "output_to_file".
Don't provide an explanation, only write the FlowScript. Don't include the word "flowscript" in the code.

Mistra Instruct Final Output:

```
digraph jobs {  
    input -> compile -> parse_file -> output_to_file -> output  
}
```

This prompt produced a really convincing output of a working FlowScript code, therefore, I kept it for the lab.

Prompt Utilized for fixing C++ Code

As shown in the last lab report, I found a prompt that produced a really convincing output. However, some helpful hints to solve this new task for this lab were missing in the JSON output:

The task is to fix the following C++ code with the given error message and provide a description of the resolution of the error in JSON format.

The fixed code should be contained in the JSON output.

The input will be formatted in JSON.

The note message will help a lot in order to figure out the error.

The line and column numbers are also provided in the JSON object.

The source C++ code is provided and it is contained in the "code" label of the JSON object provided.

The error message is in the "error" label of the JSON object provided.

The note message is in the "note" label of the JSON object provided, if any.

Example:

```
{"./Project1/Dog.cpp": [{"code": "    ptr2 = &val;\n\n    std::cout << \"Hello World!!!\" << std::endl\n    return 0;\n\n", "column": 47, "error": "expected ';' before 'return'", "file": "./compilecode/Project1/hello_world.cpp", "line": 11}]}
```

Response:

```
{"code": "    ptr2 = &val;\n\n    std::cout << \"Hello World!!!\" << std::endl;\n    return 0;\n\n", "resolution": "The error was caused by a missing semicolon in line 11, column 47. As a result, a semicolon was added after the std::endl statement in order to solve the compiler error."}
```

End of Example

Here is the task:

```
{"/Project1/Dog.cpp": [{"code": "{\n    int woof = 1\n}\n", "column": 1, "error":  
"expected ',' or ';' before '}' token", "file":  
"./compilecode/Project1/Dog.cpp", "line": 8}]}
```

Mistral Instruct Output:

```
Here is the fixed code in JSON format:\n{\n    \"/Project1/Dog.cpp\": {\n  
\"code\": \"{ \\n\\ntint woof = 1; \\n}\\n\", \n        \"resolution\": \"The error was  
caused by a missing semicolon after the line containing only an integer value (woof).  
As a result, the necessary semicolons were added before and after the line in order to  
solve the compiler error.\\n    }\\n}
```

To accomplish the new task for this lab, I added more specifications to the prompt, like providing a line number and the file name in the output. Also, I specified in the single shot training example that the output should be a JSON list this time instead of a single object. Lastly, I removed some non-important redundant instructions from the previous prompt, like the "note" label specifications since I noticed that the LLM already knows how to deal with it and also avoid more confusion in general:

```
The task is to fix the following C++ code errors with the given error messages and  
provide a description of the resolution of the error in JSON format.  
The fixed codes should be contained in the JSON output, as well as the line numbers,  
and the resolutions.  
Include the file names in the JSON output.  
The input will be formatted in JSON.  
The line and column numbers are also provided in the JSON object.  
The source C++ code is provided and it is contained in the "code" label of the JSON  
object provided.
```

Example:

```
{"/Project1/Dog.cpp": [{"code": "    ptr2 = &val;\n\n    std::cout << \"Hello  
World!!!\" << std::endl\n    return 0;\n}\n", "column": 47, "error": "expected ';' before 'return'", "file": ".\\compilecode/Project1/hello_world.cpp", "line": 11}]}
```

Response:

```
[{"fixed_code": "    ptr2 = &val;\n\n    std::cout << \"Hello World!!!\" <<  
std::endl;\n    return 0;\n}\n", "line": 11, "resolution": "The error was caused by a  
missing semicolon in line 11, column 47. As a result, a semicolon was added after the  
std::endl statement in order to solve the compiler error.", "file_name":  
"./Project1/Dog.cpp"},]
```

End of Example

Here is the task:

```
{ "../Data/compilecode/Project1/Dog.cpp": [{"code": "{\n    int woof = 1\n}\n\n", "column": 1, "error": "expected ',', ' or ';' before '}' token", "file": "../Data/compilecode/Project1/Dog.cpp", "line": 8}], "../Data/compilecode/Project1/Dog.h": [{"code": "{\npublic:\n    void Dog();\n    void bark();\n};\n", "column": 5, "error": "return type specification for constructor invalid", "file": "../Data/compilecode/Project1/Dog.h", "line": 7}], "../Data/compilecode/Project1/hello_world.cpp": [{"code": "#pragma once\n#include <iostream>\n\n", "column": 9, "file": "../Data/compilecode/Project1/hello_world.cpp", "line": 1, "warning": "#pragma once in main file"}, {"code": "    char *ptr2;\n    ptr1 = &val;\n    ptr2 = &val;\n\n    std::cout << \"Hello World!!!\" << std::endl;\n", "column": 12, "error": "cannot convert 'float*' to 'char*' in assignment", "file": "../Data/compilecode/Project1/hello_world.cpp", "line": 9}]]
```

Mistra Instruct Output:

```
{
  "fixed_code": "{\n    int woof = 1;\n}",
  "column": 2,
  "resolution": "The error was caused by a missing semicolon in line 8, column 1. As a result, a semicolon was added after the integer value of 'woof' to solve the compiler error.",
  "file_name": "../Data/compilecode/Project1/Dog.cpp",
  "line": 9
},
{
  "fixed_code": "{\npublic:\n    void Dog();\n    void bark();\n};\n",
  "column": 8,
  "resolution": "The error was caused by an invalid return type specification for the constructor in line 7. As a result, the 'void' keyword before the function names were removed to fix the compiler error.",
  "file_name": "../Data/compilecode/Project1/Dog.h",
  "line": 8
},
{
  "fixed_code": "{\n#pragma once\n#include <iostream>\n",
  "column": 9,
```

```

"resolution": "The error was caused by a warning message in line 1 of the file
'hello_world.cpp'. As a result, the '#pragma once' directive before the include
statement was removed to fix the compiler error.",
"file\_name": "../Data/compilecode/Project1/hello_world.cpp",
"line": 2
},
{
"fixed\_code": "    char *ptr2;\n    ptr1 = &val;\n    ptr2 =
static_cast<char*>(ptr1);\n\n    std::cout << \"Hello World!!!\" << std::endl;\n",
"column": 9,
"resolution": "The error was caused by a type mismatch in the assignment of 'ptr2' to
'val'. As a result, a static cast operator was used to convert the pointer value from
'float*' to 'char*' before assigning it to 'ptr2', which fixed the compiler error.",
"file\_name": "../Data/compilecode/Project1/hello_world.cpp",
"line": 9
}

```

This prompt produced a good result with the correct format and fixes, however, the resulting JSON format is not valid since it is missing the square brackets ([]) to make it an array, and to be able to iterate over it, it is needed. Therefore I just tweaked the sentence where I specify how I want my output to be and also changed the square brackets in the example to correct the formatting:

The task is to fix the following C++ code errors with the given error messages and provide a description of the resolution of the error in JSON format.

The output JSON must be an array of the fixed codes, as well as the line numbers, and the resolutions.

Include the file names in the JSON output.

The input will be formatted in JSON.

The line and column numbers are also provided in the JSON object.

The source C++ code is provided and it is contained in the "code" label of the JSON object provided.

Example:

```

{"/Project1/Dog.cpp": [{"code": "    ptr2 = &val;\n\n    std::cout << \"Hello
World!!!\" << std::endl\n    return 0;\n}\n", "column": 47, "error": "expected ';'
before 'return'", "file": "../compilecode/Project1/hello_world.cpp", "line": 11}]}

```

Response:

```
[{"fixed_code": "    ptr2 = &val;\n\n    std::cout << \"Hello World!!!\" <<\nstd::endl;\n    return 0;\n}\n", "line": 11, "resolution": "The error was caused by a\nmissing semicolon in line 11, column 47. As a result, a semicolon was added after the\nstd::endl statement in order to solve the compiler error.", "file_name":\n\"./Project1/Dog.cpp"}]
```

End of Example

Here is the task:

```
{"../Data/compilecode/Project1/Dog.cpp":[{"code":"\n    int woof =\n1\n}\n\n", "column":1, "error":"expected ',' or ';' before '}'\ntoken", "file":"../Data/compilecode/Project1/Dog.cpp", "line":8}], "../Data/compilecode/P\nroject1/Dog.h":[{"code":"\npublic:\n    void Dog();\n    void\nbark();\n};\n", "column":5, "error":"return type specification for constructor\ninvalid", "file":"../Data/compilecode/Project1/Dog.h", "line":7}], "../Data/compilecode/P\nroject1/hello_world.cpp":[{"code":"#pragma once\n#include\n<iostream>\n\n", "column":9, "file":"../Data/compilecode/Project1/hello_world.cpp", "line\n":1, "warning":"#pragma once in main file"}, {"code":"    char *ptr2;\n    ptr1 =\n&val;\n    ptr2 = &val;\n\n    std::cout << \"Hello World!!!\" <<\nstd::endl;\n", "column":12, "error":"cannot convert 'float*' to 'char*' in\nassignment", "file":"../Data/compilecode/Project1/hello_world.cpp", "line":9}]}
```

Mistra Instruct Final Output:

```
{"../Dog.cpp":[{"fixed_code":"\n    int woof = 1;\n}\n\n", "column":1, "resolution":"The\nerror was caused by a missing semicolon in line 8, column 1. As a result, a semicolon\nwas added after the int statement to solve the compiler error.",\n"file_name":"../Dog.cpp"}], "../Dog.h":[{"fixed_code":"\npublic:\n    void Dog();\n\nvoid bark();\n};\n", "column":5, "resolution":"The error was caused by an invalid return\ntype specification for the constructor in line 7, column 5. As a result, the\nconstructor declaration was updated to include the 'void' keyword before the function\nname.", "file_name":"../Dog.h"}], "../hello_world.cpp":[{"fixed_code":"#pragma\nonce\n#include <iostream>\n\n", "column":9, "resolution":"The error was caused by a\nwarning message in line 1, column 9 of the file ./hello_world.cpp. As a result, the\n#pragma directive was removed to resolve the warning.",\n"file_name":"../hello_world.cpp"}]}
```

This last prompt produced a neat format for the error correction. However, this LLM (Mistra Instruct) was not consistent with the structure of the output JSON, therefore, I had to switch to a smarter LLM, in this case, ChatGPT3.5. I provided the same prompt as above and the result was really good and consistent every time, which is perfect for the task I am trying to accomplish:

The task is to fix the following C++ code errors with the given error messages and provide a description of the resolution of the error in JSON format.

The output JSON must be an array of the fixed codes, as well as the line numbers, and the resolutions.

Include the file names in the JSON output.

The input will be formatted in JSON.

The line and column numbers are also provided in the JSON object.

The source C++ code is provided and it is contained in the "code" label of the JSON object provided.

Example:

```
{"/Project1/Dog.cpp": [{"code": "    ptr2 = &val;\n\n    std::cout << \"Hello World!!!\" << std::endl\n    return 0;\n\n", "column": 47, "error": "expected ';' before 'return'", "file": "../compilecode/Project1/hello_world.cpp", "line": 11}]}
```

Response:

```
[{"fixed_code": "    ptr2 = &val;\n\n    std::cout << \"Hello World!!!\" << std::endl;\n    return 0;\n\n", "line": 11, "resolution": "The error was caused by a missing semicolon in line 11, column 47. As a result, a semicolon was added after the std::endl statement in order to solve the compiler error.", "file_name": "../Project1/Dog.cpp"}]
```

End of Example

Here is the task:

```
{"/Data/compilecode/Project1/Dog.cpp": [{"code": "\n    int woof = 1\n}\n\n", "column": 1, "error": "expected ',' or ';' before '}' token", "file": "../Data/compilecode/Project1/Dog.cpp", "line": 8}], "/Data/compilecode/Project1/Dog.h": [{"code": "\npublic:\n    void Dog();\n    void bark();\n};\n", "column": 5, "error": "return type specification for constructor invalid", "file": "../Data/compilecode/Project1/Dog.h", "line": 7}], "/Data/compilecode/Project1/hello_world.cpp": [{"code": "#pragma once\n#include <iostream>\n\n", "column": 9, "file": "../Data/compilecode/Project1/hello_world.cpp", "line": 1, "warning": "#pragma once in main file"}, {"code": "    char *ptr2;\n    ptr1 = &val;\n    ptr2 = &val;\n\n    std::cout << \"Hello World!!!\" << std::endl;\n", "column": 12, "error": "cannot convert 'float*' to 'char*' in assignment", "file": "../Data/compilecode/Project1/hello_world.cpp", "line": 9}]}
```

ChatGPT3.5 Output:

```
```json
[
 {
 "fixed_code": "{\n int woof = 1;\n}\n",
 "line": 8,
 "resolution": "The error was caused by a missing semicolon in line 8, column 1. A semicolon was added after the 'int woof = 1' statement to resolve the issue.",
 "file_name": "../Data/compilecode/Project1/Dog.cpp"
 },
 {
 "fixed_code": "{\npublic:\n Dog();\n void bark();\n};\n",
 "line": 7,
 "resolution": "The error was due to an invalid return type specification for the constructor in line 7, column 5. The 'void' return type was removed from the constructor declaration to fix the issue.",
 "file_name": "../Data/compilecode/Project1/Dog.h"
 },
 {
 "fixed_code": "#pragma once\n#include <iostream>\n\n",
 "line": 1,
 "resolution": "#pragma once directive is used in the main file, and no action is required. This is a warning and does not affect the compilation.",
 "file_name": "../Data/compilecode/Project1/hello_world.cpp"
 },
 {
 "fixed_code": " char *ptr2;\n ptr1 = &val;\n ptr2 = (char*)&val;\n\nstd::cout << \"Hello World!!!\" << std::endl;\n",
 "line": 9,
 "resolution": "The error was caused by attempting to assign a float pointer to a char pointer in line 9, column 12. To fix the issue, a type cast (char*) was added to the assignment of 'ptr2' to match the types.",
 "file_name": "../Data/compilecode/Project1/hello_world.cpp"
 }
]
```
```

I saved this prompt to use it when I call the LLM from the program and now called the ChatGPT API instead of the local LLM Minstra Instruct.

The Compile Job

The Compile Job is implemented to execute a Makefile command that compiles one or more C++ files, where the compilation output is JSON formatted. It returns a JSON object containing two strings: the compile output (In JSON format) and the project name. This Job gets created when loading all the jobs into the FlowScript interpreter, right after the LLM finishes generating the FlowScript compilation pipeline.

Example input:

```
make project1
```

Where the Makefile command is the following:

```
.SILENT:
project1:
    -g++ -fdiagnostics-format=json -g -std=c++14 ../Data/compilecode/Project1/*.cpp -o
    ../Data/compilecode/Project1/auto_out1
```

Example output:

```
{
  "file_name": "output_project1.json",
  "output": "[{\n\"kind\": \"error\", \"message\": \"return type specification for\nconstructor invalid\", \"children\": [], \"column-origin\": 1, \"locations\": [{\n\"caret\":\n{\n\"file\": \"../Data/compilecode/Project1/Dog.h\", \"line\": 7, \"display-column\": 5,\n\"byte-column\": 5, \"column\": 5}, \"finish\": {\n\"file\":\n\"../Data/compilecode/Project1/Dog.h\", \"line\": 7, \"display-column\": 8, \"byte-column\":\n8, \"column\": 8}}], \"escape-source\": false}, {\n\"kind\": \"error\", \"message\":\n\"expected ', ' or ';' before '}' token\", \"children\": [], \"column-origin\": 1,\n\"locations\": [{\n\"caret\": {\n\"file\": \"../Data/compilecode/Project1/Dog.cpp\", \"line\":\n8, \"display-column\": 1, \"byte-column\": 1, \"column\": 1}}], \"escape-source\":\nfalse}]\n[{\n\"kind\": \"error\", \"message\": \"cannot convert 'float*' to 'char*' in\nassignment\", \"children\": [], \"column-origin\": 1, \"locations\": [{\n\"caret\": {\n\"file\":\n\"../Data/compilecode/Project1/hello_world.cpp\", \"line\": 9, \"display-column\": 12,\n\"byte-column\": 12, \"column\": 12}, \"finish\": {\n\"file\":\n\"../Data/compilecode/Project1/hello_world.cpp\", \"line\": 9, \"display-column\": 15,\n\"byte-column\": 15, \"column\": 15}, \"label\": \"float*\"}], \"escape-source\": false}]\n"]
}
```

The Parse Job

The Parse Job is implemented to parse the whole compilation output error (formatted in JSON) into errors for each file. It returns a JSON that contains the error for each file (formatted in JSON) and the project name. To make this work, this Job calls a separate local function called “generateJson”, which gets a split raw compilation error JSON and reorganizes the content into a global JSON as shown below. The “generateJson” function also is in charge of reading the source file and extracting the line of the error plus two lines above and below. This Job gets created when loading all the jobs into the FlowScript interpreter, right after the LLM finishes generating the FlowScript compilation pipeline.

Example input:

```
{
  "file_name": "output_project1.json",
  "output": "[{\\"kind\\": \\"error\\", \\"message\\": \\"return type specification for
constructor invalid\\", \\"children\\": [], \\"column-origin\\": 1, \\"locations\\": [{\\"caret\\":
{\\"file\\": \\"../Data/compilecode/Project1/Dog.h\\", \\"line\\": 7, \\"display-column\\": 5,
\\"byte-column\\": 5, \\"column\\": 5}, \\"finish\\": {\\"file\\":
\\"../Data/compilecode/Project1/Dog.h\\", \\"line\\": 7, \\"display-column\\": 8, \\"byte-column\\":
8, \\"column\\": 8}}], \\"escape-source\\": false}, {\\"kind\\": \\"error\\", \\"message\\":
\\"expected ', ' or ';' before ')' token\\", \\"children\\": [], \\"column-origin\\": 1,
\\"locations\\": [{\\"caret\\": {\\"file\\": \\"../Data/compilecode/Project1/Dog.cpp\\", \\"line\\":
8, \\"display-column\\": 1, \\"byte-column\\": 1, \\"column\\": 1}}], \\"escape-source\\":
false}]\n[{\\"kind\\": \\"error\\", \\"message\\": \\"cannot convert 'float*' to 'char*' in
assignment\\", \\"children\\": [], \\"column-origin\\": 1, \\"locations\\": [{\\"caret\\": {\\"file\\":
\\"../Data/compilecode/Project1/hello_world.cpp\\", \\"line\\": 9, \\"display-column\\": 12,
\\"byte-column\\": 12, \\"column\\": 12}, \\"finish\\": {\\"file\\":
\\"../Data/compilecode/Project1/hello_world.cpp\\", \\"line\\": 9, \\"display-column\\": 15,
\\"byte-column\\": 15, \\"column\\": 15}, \\"label\\": \\"float*\\"}], \\"escape-source\\": false}]\n"
}
```

Example of split raw JSON passed into the “generateJson” function:

```
[
  {
    "kind": "error",
    "message": "return type specification for constructor invalid",
    "children": [

    ],
    "column-origin": 1,
    "locations": [
```

```

{
  "caret": {
    "file": "../Data/compilecode/Project1/Dog.h",
    "line": 7,
    "display-column": 5,
    "byte-column": 5,
    "column": 5
  },
  "finish": {
    "file": "../Data/compilecode/Project1/Dog.h",
    "line": 7,
    "display-column": 8,
    "byte-column": 8,
    "column": 8
  }
},
],
"escape-source": false
},
{
  "kind": "error",
  "message": "expected ',' or ';' before '}' token",
  "children": [

],
"column-origin": 1,
"locations": [
  {
    "caret": {
      "file": "../Data/compilecode/Project1/Dog.cpp",
      "line": 8,
      "display-column": 1,
      "byte-column": 1,
      "column": 1
    }
  }
],
"escape-source": false
}
]

```

Example return of the “generateJson” function:

```

{
  "../Data/compilecode/Project1/Dog.cpp": [
    {
      "code": "{\n    int woof = 1\n}\n\n",
      "column": 1,
      "error": "expected ',' or ';' before '}' token",
      "file": "../Data/compilecode/Project1/Dog.cpp",
      "line": 8
    }
  ]
}

```

```

    }
  ],
  "../Data/compilecode/Project1/Dog.h": [
    {
      "code": "{\npublic:\n    void Dog();\n    void bark();\n};\n",
      "column": 5,
      "error": "return type specification for constructor invalid",
      "file": "../Data/compilecode/Project1/Dog.h",
      "line": 7
    }
  ]
}

```

Example output:

```

{
  "content": {
    "../Data/compilecode/Project1/Dog.cpp": [
      {
        "code": "{\n    int woof = 1\n}\n\n",
        "column": 1,
        "error": "expected ',' or ';' before '}' token",
        "file": "../Data/compilecode/Project1/Dog.cpp",
        "line": 8
      }
    ],
    "../Data/compilecode/Project1/Dog.h": [
      {
        "code": "{\npublic:\n    void Dog();\n    void bark();\n};\n",
        "column": 5,
        "error": "return type specification for constructor invalid",
        "file": "../Data/compilecode/Project1/Dog.h",
        "line": 7
      }
    ],
    "../Data/compilecode/Project1/hello_world.cpp": [
      {
        "code": "    char *ptr2;\n    ptr1 = &val;\n    ptr2 = &val;\n\n    std::cout <<
        \"Hello World!!!\" << std::endl;\n",
        "column": 12,
        "error": "cannot convert 'float*' to 'char*' in assignment",
        "file": "../Data/compilecode/Project1/hello_world.cpp",
        "line": 9
      }
    ]
  },
  "file_name": "output_project1.json"
}

```

The Output To File Job

The Output To File Job is implemented to simply output content to an output file. In the first case, the content provided is the FlowScript pipeline generated by the LLM, and in the second case, the parsed error JSON object. This Job needs to clean any JSON provided first to output it to the JSON file since otherwise, the JSON format would be invalid. It returns a message that indicates that the Job is completed. This Job gets created when outputting the FlowScript pipeline to its respective file and when loading all the jobs into the FlowScript interpreter, right after the LLM finishes generating the FlowScript compilation pipeline.

Example input:

```
{
  "content": {
    "../Data/compilecode/Project1/Dog.cpp": [
      {
        "code": "{\n    int woof = 1\n}\n\n",
        "column": 1,
        "error": "expected ',' or ';' before '}' token",
        "file": "../Data/compilecode/Project1/Dog.cpp",
        "line": 8
      }
    ],
    "../Data/compilecode/Project1/Dog.h": [
      {
        "code": "{\npublic:\n    void Dog();\n    void bark();\n};\n",
        "column": 5,
        "error": "return type specification for constructor invalid",
        "file": "../Data/compilecode/Project1/Dog.h",
        "line": 7
      }
    ],
    "../Data/compilecode/Project1/hello_world.cpp": [
      {
        "code": "    char *ptr2;\n    ptr1 = &val;\n    ptr2 = &val;\n\n    std::cout <<
        \"Hello World!!!\" << std::endl;\n",
        "column": 12,
        "error": "cannot convert 'float*' to 'char*' in assignment",
        "file": "../Data/compilecode/Project1/hello_world.cpp",
        "line": 9
      }
    ]
  },
  "file_name": "output_project1.json"
}
```

Example output to the JSON file (output_project1.json):

```
{
  "../Data/compilecode/Project1/Dog.cpp": [
    {
      "code": "{\n    int woof = 1\n}\n\n",
      "column": 1,
      "error": "expected ',' or ';' before '}' token",
      "file": "../Data/compilecode/Project1/Dog.cpp",
      "line": 8
    }
  ],
  "../Data/compilecode/Project1/Dog.h": [
    {
      "code": "{\npublic:\n    void Dog();\n    void bark();\n};\n",
      "column": 5,
      "error": "return type specification for constructor invalid",
      "file": "../Data/compilecode/Project1/Dog.h",
      "line": 7
    }
  ],
  "../Data/compilecode/Project1/hello_world.cpp": [
    {
      "code": "    char *ptr2;\n    ptr1 = &val;\n    ptr2 = &val;\n\n    std::cout <<
\"Hello World!!!\" << std::endl;\n",
      "column": 12,
      "error": "cannot convert 'float*' to 'char*' in assignment",
      "file": "../Data/compilecode/Project1/hello_world.cpp",
      "line": 9
    }
  ]
}
```

Example output:

```
Done!
```

The LLM Call Job

The LLM Call Job is implemented to call any LLM API by a Python script that handles the request. An IP, prompt, model name, and an optional API key are the parameters needed to execute this job. It returns just the LLM response as a string. This Job gets created when calling the LLM to generate the FlowScript pipeline and when calling the LLM to generate a fix for the C++ source code.

Example input:

```
{
  "ip": "http://localhost:4891/v1/chat/completions",
  "prompt": "[GENERATE FLOWSCRIPT PROMPT HERE]",
  "model": "mistral-7b-instruct-v0.1.Q4_0"
}
```

Raw LLM JSON response:

```
{
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "message": {
        "content": "digraph jobs {\n    input -> compile -> parse_file -> output_to_file ->\noutput\n}\n",
        "role": "assistant"
      },
      "references": []
    }
  ],
  "created": 1702145132,
  "id": "foobarbaz",
  "model": "Mistral Instruct",
  "object": "text_completion",
  "usage": {
    "completion_tokens": 11,
    "prompt_tokens": 143,
    "total_tokens": 154
  }
}
```

Example output:

```
digraph jobs {
  input -> compile -> parse_file -> output_to_file -> output
}
```

It is worth noting that all Jobs get created by just calling the “createJob” with the function name and a pointer to that function, thanks to the capabilities of the Job System library. Also, there are some other functions like “cleanJson” and openFile were implemented to reduce repeated code and abstract tasks concerning JSON formatting. These two functions ensure that JSON objects passed as strings do not break when parsed back.

Discussion of Example Ran

As seen in the example run of the [video recording](#), there is a project with errors on three different files: hello_world.cpp, Dog.h, and Dog.h.

hello_world.cpp

```
// #pragma once
#include <iostream>

int main()
{
    float *ptr1, val = 3.14;
    char *ptr2;
    ptr1 = &val;
    ptr2 = &val; // error: a value of type "float *" cannot be assigned to an entity of type "char *"

    std::cout << "Hello World!!!" << std::endl;
    return 0;
}
```

Dog.h

```
#pragma once
#include <iostream>

class Dog
{
public:
    void Dog(); // error: member function with the same name as its class must be a constructor
    void bark();
};
```

Dog.cpp

```
#include "Dog.h"

Dog::Dog() {}

void Dog::bark()
{
    int woof = 1 // error: expected a ;
}
```

After the program is executed, it asks to input a makefile command to run. In this case, is “project1” which is the Makefile command that compiles the C++ source code contained in the “compilecode” folder of the “Data” folder. Then, it is seen that the FlowScript Job Starts running, which is the one that generates the FlowScript pipeline of compilation. This takes about a minute since this is a call made using the Call LLM Job to the local GPT4All LLM, using the previously discussed prompt to generate FlowScript. This LLM uses the CPU to process the prompt, therefore explaining why it takes a relatively long time. After the correct FlowScript is provided, the Agent then runs the Output To File Job to write the FlowScript generated to a file called “compiling_pipeline.dot”. Then, it runs the FlowScript file to start the pipeline to compile, parse files, and output the errors. After determining that the project contains errors, it calls the LLM through the Call LLM Job again, this time using ChatGPT3.5 (Faster and Smarter) to fix the errors. Using the previously discussed prompt to fix C++ code, the LLM returns possible fixes for the three different files as seen in the video along with an explanation of the error and fix:

```
File Name: ../Data/compilecode/Project1/Dog.cpp
```

```
Line number: 8
```

```
Resolution explanation: The error was caused by a missing semicolon in line 8, column 1. As a result, a semicolon was added after the declaration statement in order to solve the compiler error.
```

```
Fixed code:
```

```
{
    int woof = 1;
}
```

```
=====
```

```
File Name: ../Data/compilecode/Project1/Dog.h
```

```
Line number: 7
```

```
Resolution explanation: The error was caused by an invalid return type specification for the constructor in line 7, column 5. Constructors do not have return types, so the return type specification was removed to solve the compiler error.
```

```
Fixed code:
```

```
{
public:
```

```

    void Dog();
    void bark();
};

```

=====

File Name: ../Data/compilecode/Project1/hello_world.cpp

Line number: 9

Resolution explanation: The error was caused by trying to assign a float pointer (&val) to a char pointer (ptr2) in line 9, column 12. To solve the error, a type cast (char *) was added before the assignment to convert the float pointer to a char pointer.

Fixed code:

```

    char *ptr2;
    ptr1 = &val;
    ptr2 = (char *)&val;

    std::cout << "Hello World!!!" << std::cout;

```

However, it is noticed that one of the fixes the LLM provided did not resolved the issue:

```

{
public:
    void Dog(); // still error: member function with the same name as its class must be a
    constructor
    void bark();
};

```

Therefore, when the Agent checks for errors again after executing the FlowScript pipeline, it detects that there is still one error to correct, therefore, it calls the LLM through the Call LLM Job again to fix this error. The response is the following:

File Name: ../Data/compilecode/Project1/Dog.h

Line number: 7

Resolution explanation: The error was caused by specifying a return type for the constructor in line 7. Constructors do not have a return type and should be declared without one. As a result, the return type specification was removed from the constructor declaration in order to fix the error.

Fixed code:

```

{

```

```
public:
    Dog();
    void bark();
};
```

Now, the error has been solved correctly and after running the FlowScript pipeline again, it is seen that all the errors have been solved.

Final Overall Description of the Agent and its Capabilities

The Autonomous Agent created for this lab is capable of fixing C++ code automatically thanks to the power of AI and LLMs. This Agent is not only able to fix errors in one source code but multiple files. By simply providing a Makefile that compiles C++ code, the Agent can detect, fix, and explain the errors to the user without the need to touch the code manually while providing a nice and intuitive interface. However, some limitations of the Agent are noticeable since it depends on the power of LLMs, therefore, there may be some occasions where the Agent will not correctly solve the errors. It is expected that these models will get better and better over time, which will make them more capable of correctly resolving errors that they could not solve before.

In that order, the Agent comes built-in with a lot of error-catching capabilities. For example, it can detect connection errors to LLMs, poorly formatted JSON objects passed, FlowScript files with errors, FlowScript files that did not produce any outputs, and API key file presence (needed to call ChatGPT 3.5). After every error (except the API file and LLM connection), the Agent is smart enough to try again until it works, which makes it very robust.