

Kendall Boesch  
48231304  
CS 3341  
9 December 2023

## Final: Autonomous Code Assistant Agent

<b>Automated System Overview.....</b>	<b>3</b>
<b>Error Inputs.....</b>	<b>3</b>
<b>Job Types.....</b>	<b>4</b>
Job Input Structure.....	4
Custom Job: Compile Job.....	5
Custom Job: Error Parse Job.....	5
Custom Job: Rest Job.....	5
Custom Job: Script Write Job.....	6
Custom Job: Code Repair Job.....	7
<b>System Runthrough.....</b>	<b>7</b>
<b>Prompt Development Journal.....</b>	<b>10</b>
C++ Compile Error Fix LLM Prompt.....	10
Prompt 1.....	10
Prompt 2.....	11
Prompt 3.....	12
Prompt 4.....	13
Prompt 5.....	15
Prompt 6.....	16
Prompt 7.....	18
Prompt 8:.....	19
Prompt 9.....	20
Prompt 11:.....	22
Flowscript Generation Prompt.....	24
Prompt 1.....	24
Prompt 2.....	27
Prompt 3.....	28
Prompt 4.....	31
Prompt 5.....	33
Prompt 6.....	36
Prompt 7.....	38
Prompt 8.....	41
Prompt 9.....	43
Prompt 10.....	46
Prompt 11.....	48
Prompt 12.....	51

Prompt 13.....	54
<b>Final LLM Prompts.....</b>	<b>57</b>
Flowscript Generation Prompt.....	57
Final Prompt.....	57
Prompt Methodology.....	59
C++ Code Repair Prompt.....	61
Final Prompt.....	61
Prompt Methodology.....	62

# Automated System Overview

**Overall:** This system takes in the description of a process flow from a text file and communicates with an LLM via rest calls, prompting the LLM to generate the Flowscript code for the described process. Once a script is produced, the system uses a Flowscript Interpreter to perform lexical and syntactical analysis on the LLM generated script. If the analysis does not produce errors, the interpreter then executes the process outlined in the Flowscript code.

**Usage:** While this system can be easily adapted to support a variety of jobs and process flows, the main.cpp of the system is configured to build and execute the Flowscript code for a particular process. In this process, a compile job for compiling a C++ file is created and executed. If the compile attempt produces compile errors, those errors are passed to an errorparse job which extracts the error message and other relevant debugging details and writes the information to a file “errors.json”. Once the error parsing is complete, a rest job is created and executed. The rest job sends the error information to the LLM and prompts it to return the error objects with the resolved source code. The updated error objects are written to fixedErrors.json, and the job completes. From there, a code repair job is created and executed. The code repair uses the updated code in the fixedErrors.json file to fix the incorrect code in the cpp file the errors occurred in. Once the code repair is complete, the compile job is refired. If the code compiles, the process is successful and it completes. If the new cpp code does not compile, the process repeats until the LLM produces code that allows the C++ file to compile.

**LLM Model:** Mistral Instruct

## Output Files:

1. Errors.json – holds the errors extracted by the errorparse job
2. FixedErrors.json – holds the error objects update with resolved code and a description of the resolution both generated by the LLM in a errorsolve rest job
3. Code/Flowscript/script.md – holds the LLM generated Flowscript code for the process described in Data/LLMFlowscriptPrompt.txt
4. Resolved C++ code – the updated C++ file that holds the code being repaired

# Error Inputs

To make integrating all of the components of this lab and the next one more seamless, I decided to keep the error object format the same when passing it into the LLM.

## Error Object Input Template:

```
{'colNum': #, 'errorMessage': 'error: ...', 'file': 'Code/toCompile/file.cpp', 'lineNum': #, 'resDescr': '', 'src': '...', 'srcResolved': ''}
```

**syntaxError.cpp**

```
{'colNum': 10, 'errorMessage': 'error: expected unqualified-id', 'file': 'Code/toCompile/syntaxError.cpp',  
'lineNum': 4, 'resDescr': '', 'src': '    int = 0;', 'srcResolved': ''}
```

#### **demoerror.cpp**

```
{'colNum': 40, 'errorMessage': "error: use of undeclared identifier 'endl'; did you mean 'std::endl'?", 'file':  
'Code/toCompile/demoerror.cpp', 'lineNum': 13, 'resDescr': '', 'src': '    std::cout << "in demoerror.cpp" <<  
endl;', 'srcResolved': ''} {'colNum': 13, 'errorMessage': "error: expected ';' after return statement", 'file':  
'Code/toCompile/demoerror.cpp', 'lineNum': 15, 'resDescr': '', 'src': '    return 0', 'srcResolved': ''}
```

#### **typeerror.cpp**

```
{'colNum': 11, 'errorMessage': "error: assigning to 'char' from incompatible type 'std::string' (aka  
'basic_string<char>')", 'file': 'Code/toCompile/typeerror.cpp', 'lineNum': 8, 'resDescr': '', 'src': '    chr = str;  
' , 'srcResolved': ''}
```

## **Job Types**

### **Job Input Structure**

All jobs in the job system take json objects as inputs. The json input structure for all jobs is as follows:

```
Json input{  
    {"identifier", ""},  
    {"inputId", ""},  
    {"inputData", ""},  
    {"output", ""},  
    {"success", ""}  
};
```

The identifier member of the input holds the string identifier for the job type being created (for example “compile” or “errorparse”). The inputId member holds the unique identifier the jobsystem stores the input underneath. This is helpful when a job that is executed in the middle of a process requires user input. The value in the inputId member is also used as the UI job label value in the Flowscript language. The inputData member hold the data specific to that job. For example, the inputData for a compile job would be the target in the makefile of the code to be compiled. The output member is where the jobs return values are stored, and usually serves as the input data for any dependent jobs. Finally, the success member of the job input structure is used to indicate whether or not the job’s execution was successful. This member is also what the executional dependencies are dependent on. For example, with a compile job, if the code being compiled produced errors, the compile was unsuccessful and the compile job’s success member is set to “false”. If the code compiled without errors, the compile job’s success member is set to true. Both the output and the success member of the json input should be empty at job creation.

## Custom Job: Compile Job

**Identifier:** “compile”

**Description:** To compile different files of source code in the job system, I created a compile job. As discussed above, the input data for a compile job is the target in the makefile to compile a specific file. In its execute function, the compile job attempts to execute the make target passed and writes the compilation output to a buffer, which every update of is appended to a string called output. If the compile fails, the compile errors are written to the output variable and the success member of the input is set to false. If the compilation was successful, the input’s success member is set to true. The makefile target for a given cpp file must be the name of the file without the .cpp extension. For example, the makefile target to compile “demoerror.cpp” would be “demoerror”

**Example input:**

```
Json compileJob
{
    {"identifier", "compile"},
    {"inputId", "demo"},
    {"inputData", "demoerror"},
    {"output", ""},
    {"success", ""}
};
```

## Custom Job: Error Parse Job

**Identifier:** errorparse

**Description:** The errorparse job takes in the results of a failed compile job as input data. The job then uses regex to extract the error message, source code, line and column number, and file name for each error in the input data. Next, the extracted for each error is placed in an error object, which is pushed onto the back of a vector. Once all the errors have been extracted and parsed, the job then opens up the errors.json file to check if there was any previously written json. If this is the case, the previously written json is read in and saved for appending, and the json file is cleared. Next, the job will iterate through each file and their nested error arrays, appending each one to the saved previous json, or a new json writer if there was no previous content in errors.json. Finally, the contents of the json writer is dumped to the errors.json, the success member for the errorparse job input is set to true, and the output is set to “errors.json”.

**Example Input:** Because errorparse jobs are dependent jobs that need the output from a compile job, there is no need to create a json input object for the job.

## Custom Job: Rest Job

**Identifier:** rest

**Description:** To communicate with the LLM via a REST interface, I utilized my previously existing compile job (CustomJob registered to the job system in main.cpp). What differentiates a restjob from a

compile job is the contents of the “inputData” attribute. With a standard compile job, the “inputData” value is either the compile command or make target to the compile command of a specific .cpp file, with a restjob, the “inputData” data member should be “pyrest” (the target to run the Python code to communicate with the LLM) followed by ARGS=”. The first item in the args should be either “flowgen” indicating you are trying to fire a job to generate Flowscript code for a process, or “errorsolve”, indicating you want the LLM to generate the resolved code to the compile errors passed. The second argument is the IP address or domain of the LLM to be communicated with. Finally, the third argument is the file for input. In the case of a Flowscript generation rest job, this would be the path to the file that holds the description of the process to generate Flowscript code for (Data/LLMFlowscriptPrompt.txt) and in the case of a code repair rest job, this would be the path to the file that holds the compile errors to be resolved (errors.json). In RestJob.py, the openai library is used to interface with the LLM via GPT4ALL on localhost. The program argument at index 1 is read into a variable “instruction”, the program argument at index 2 is read into openai.api\_base, and the argument at index 3 is passed into a variable promptFile. Because the LLM is running locally on with GPT4all, the model variable is set to NA and the openai.api\_key is set to “not needed for a local LLM”

If the instruction is set to flowgen, the LLM produces the Flowscript code for the requested process and prints it to the file Code/Flowscript/script.md, and prints the file path. If the instruction is set to errorsolve, the resolved error objects are printed to fixedErrors.json, and the file path is printed. Once the work in RestJob.py is completed, the rest job’s execution function extracts the printed file path from the output of RestJob.py and saves it to the output member of the json input object.

#### Example Input:

```
Json restJob
{
    {"identifier", "rest"},
    {"inputId", "errorfix"},
    {"inputData", "pyrest ARGS=\"errorsolve http://localhost:4891/v1
errors.json\""},
    {"output", ""},
    {"success", ""}
};
```

## Custom Job: Script Write Job

**Identifier:** scriptwrite

**Description:** Once a rest job has executed, a scriptwrite job is executed to ensure the LLM generated Flowscript will be properly read. The scriptwrite job takes in the path to the LLM generated Flowscript file. This job checks to ensure the LLM did not encase its response in “`” and removes any empty lines because if the first line is empty the Flowscript Interpreter will throw a syntax error. After reading in the script, the scriptwrite job empties the script.md file and writes the script to it, omitting any of the lines of code described above. Once the script write is complete, the success member of the json input is set to true and the output member is set to the path to the file the process’ Flowscript code is written in.

#### Example Input:

```
Json scriptWriteJob
```

```
{
    {"identifier", "scriptwrite"},
    {"inputId", "flowwrite"},
    {"inputData", "Code/Flowscript/script.md"},
    {"output", ""},
    {"success", ""}
};
```

## Custom Job: Code Repair Job

**Identifier:** coderepair

**Description:** The input for a coderepair job is the path to the file with the LLM generated code fixes are written. This job utilizes the resolved errors generated in the errorsolve rest job to update the files. For each error entry in fixedErrors.json, this job will open the .cpp file and replace the lines of code that are producing errors with the resolved code generated by the rest job. If the srcResolved member of the error object that is supposed to hold the solution is empty, the coderepair job will re-enter the old code back into the file.

**Example Input:**

```
Json repairJob
{
    {"identifier", "coderepair"},
    {"inputId", "repair"},
    {"inputData", "fixedErrors.json"},
    {"output", ""},
    {"success", ""}
};
```

## System Runthrough

**Video Demonstration:** <https://youtu.be/koW0mxXWc9c>

- In main.cpp, all job types are registered and any job inputs are loaded to the job system (see previous section on job types)
- Still in main.cpp, the system creates a rest job to generate the Flowscript code for the execution process outlined in Data/LLMFlowscriptPrompt.txt. (See section on final prompts)
  - Example prompt: Create job A with input "demoerror"
    - If job A is successful, the process ends and no other jobs should be created or executed
    - If A fails , create an errorparse job, job B
    - After job B finishes, create job C with input "restjob"
    - After job C finishes, create a coderepair job, job D
    - If job D is successful, job A executes again

if Job D fails, the process ends and no other jobs are created

- The system will continue to repeat the rest call until a valid script is returned. The script is written to Code/Flowscript/script.md.

- LLM generated script: digraph exampleflow

```
{
  subgraph clusterExample
  {
    start[shape=none];
    a[label=demoerror];
    b[label=errorparse];
    c[label=restjob];
    d[label=coderepair];
    end[shape=point];
    start->a;
    a->b[label=false];
    a->end[label=true];
    b->c;
    c->d;
    d->end[label=false];
    d->a[label=true];
  }
}
```

- 
- Once a valid script is produced, an instance of FlowscriptInterpreter is created and a call to interpret the LLM generated Flowscript is made.
- The interpreter performs lexical and syntactical analysis on the LLM generated Flowscript. If no errors are produced during analysis, the interpreter will build the workflow and begin to execute the jobs.

- First, the system creates a compile job to compile the code in the file demoerror.txt

- File contents: 

```
int main()
{
    std::cout << "in demoerror.cpp" << endl;
    return 0
}
```

- When the compile fails, the system will create an error parse job to isolate and extract each compile error along with any other relevant debugging information, outputting it to errors.json

- Errors.json contents for demoerror.cpp: {  
"Code/toCompile/demoerror.cpp": [  
 {  
 "colNum": 40,  
 "errorMessage": "error: use of undeclared identifier  
'endl'; did you mean 'std::endl'?",  
 "file": "Code/toCompile/demoerror.cpp",



```

        "lineNum": 13,
        "nextLine": "    return 0",
        "previousLine": "{",
        "resDescr": "",
        "src": "    std::cout << \"in demoerror.cpp\" <<
endl;",
        "srcResolved": ""
    },
    {
        "colNum": 13,
        "errorMessage": "error: expected ';' after return
statement",
        "file": "Code/toCompile/demoerror.cpp",
        "lineNum": 14,
        "nextLine": "}",
        "previousLine": "    std::cout << \"in
demoerror.cpp\" << endl;",
        "resDescr": "",
        "src": "    return 0",
        "srcResolved": ""
    }
]
}

```

- Once the errorparse job is complete and the compile errors are written to errors.json, the job system creates and executes another rest job. This rest job communicates with the LLM, sending it the contents of errors.json, and prompting it to provide the resolved C++ code for each error. (See section on final prompts).
- In the code repair rest job each error object in errors.json is rewritten to a file called fixedErrors.json, where the “srcResolved” member is populated by the resolved C++ code and the “resDescr” member holds a description of the error resolution.

Example contents of fixedErrors.json: {

```

"Code/toCompile/demoerror.cpp": [
{
"colNum": 40,
"errorMessage": "error: use of undeclared identifier
'endl'; did you mean 'std::endl'?",
"file": "Code/toCompile/demoerror.cpp",
"lineNum": 13,
"nextLine": "    return 0",
"previousLine": "{",
"resDescr": "Replace the use of endl with std::endl.",
"srcResolved": "    std::cout << \"in demoerror.cpp\" <<
std::endl;"
},
{

```

```

"colNum": 13,
"errorMessage": "error: expected ';' after return
statement",
"file": "Code/toCompile/demoerror.cpp",
"lineNum": 14,
"nextLine": "",
"previousLine": "    std::cout << \"in demoerror.cpp\" <<
std::endl;",
"resDescr": "Add a semicolon after the return statement.",
"srcResolved": "    return 0;"
}
]
}

```

- After producing the resolved code in the rest job, the job system creates and executes a coderepair job. The code repair job iterates through the fixedErrors.json file and opens each C++ source code file. In each file, the job replaces the line of code causing compile errors with the resolved code provided by the LLM and written to fixedErrors.json. If the “srcResolved” member of the error object does not hold any value, the coderepair job will leave the original code, so when the code is recompiled, the LLM will be re-prompted to provide a resolution.
  - A coderepair job only fails if there is a file read error. In this case, the entire process exits and is completed unsuccessfully.
  - Example fixed demoerror.cpp:
 

```

int main()
{
    std::cout << "in demoerror.cpp" << std::endl;
    return 0;
}

```
- Once the code repair job has completed, the job system will try to execute the original compile job with the fixed code. If the updated code compiles, the process completes with success. If the resolved code continues to produce compile errors, the entire process repeats until the C++ code in the original compile job compiles successfully.

## Prompt Development Journal

### C++ Compile Error Fix LLM Prompt

#### Prompt 1

This is the format of an error object:\n {\n{'colNum': #}, \n {'errorMessage': ""},\n{'file': ""},\n{'lineNum': #},\n{'resDescr': ""},\n{'src': ""},\n{'srcResolved': ""}\n}\n\nFor each error object given, provide the resolved C++ code in the 'srcResolved' member, and provide a description of

the fix in the 'resDescr' member. Leave the incorrect code in the 'src' member. \nReturn only the updated error object, do not prompt your response.

**Errors Passed:** demoerror.cpp

### Response

```
{'colNum': 40, 'errorMessage': "error: use of undeclared identifier 'endl'; did you mean 'std::endl'?", 'file': 'Code/toCompile/demoerror.cpp', 'lineNum': 13, 'nextLine': ' return 0', 'previousLine': '{', 'resDescr': 'Added the namespace for the endl identifier.', 'src': ' std::cout << "in demoerror.cpp" << endl;', 'srcResolved': ' std::cout << "in demoerror.cpp" << std::endl;}'
```

```
{'colNum': 13, 'errorMessage': "error: expected ';' after return statement", 'file': 'Code/toCompile/demoerror.cpp', 'lineNum': 14, 'nextLine': '}', 'previousLine': ' std::cout << "in demoerror.cpp" << std::endl;', 'resDescr': 'Added a semicolon at the end of the return statement.', 'src': ' return 0', 'srcResolved': ' return 0;}'
```

**Reasoning:** This was my final prompt for lab 4

**Feedback:** This is the error format I want, now I need them listed under their files in json format

### Prompt 2

This is the format of an error object:\n {\n{'colNum' : #}, \n {'errorMessage': ""},\n{'file': ""},\n{'lineNum': #},\n{'resDescr': ""},\n{'src': ""},\n{'srcResolved': ""}\n}\n\nFor each error object given, provide the resolved C++ code in the 'srcResolved' member, and provide a description of the fix in the 'resDescr' member. Leave the incorrect code in the 'src' member. \nYour response should be in json format and have an entry for each file, with a nested list of the updated error objects. Return only the json do not prompt your response.

**Errors Passed:** demoerror.cpp

### Response: {

```
"Code/toCompile/demoerror.cpp": [  
{  
  "colNum": 40,  
  "errorMessage": "error: use of undeclared identifier 'endl'; did you mean 'std::endl'?",  
  "file": "Code/toCompile/demoerror.cpp",  
  "lineNum": 13,  
  "resDescr": "Resolved undeclared identifier 'endl' by using 'std::endl'",  
  "src": " std::cout << "in demoerror.cpp" << endl;",  
  "srcResolved": " std::cout << "in demoerror.cpp" << std::endl;"
```

```

    },
    {
      "colNum": 13,
      "errorMessage": "error: expected ';' after return statement",
      "file": "Code/toCompile/demoerror.cpp",
      "lineNum": 14,
      "resDescr": "Added missing semicolon after the return statement",
      "src": " return 0",
      "srcResolved": " return 0;"
    }
  ]
}

```

**Reasoning:** I added the part in blue to get the LLM to produce the output in the same format as the errors.json file

**Feedback:** This is the output I want, but after running multiple tests, the response is inconsistent

### Prompt 3

**Prompt:** This is the format of an error object:

```

{
  {'colNum' : #},
  {'errorMessage': ""},
  {'file': ""},
  {'lineNum': #},
  {'nextLine': ""},
  {'previousLine': ""},
  {'resDescr': ""},
  {'src': ""},
  {'srcResolved': ""}
}

```

For each error object given, provide the resolved C++ code in the 'srcResolved' member, and provide a description of the fix in the 'resDescr' member.

Leave the incorrect code in the 'src' member.

'srcResolved' should only contain valid c++ code.

Your response should be in json format and have an entry [for the file path](#), with a nested list of the updated error objects. Do not prompt your response.

**Errors Passed:** demoerror.cpp

**Response:** {  
 "file": ["Code/toCompile/demoerror.cpp"],  
 "errors": [  
 {  
 "colNum": 40,  
 "errorMessage": "error: use of undeclared identifier 'endl'; did you mean 'std::endl'",  
 "file": "Code/toCompile/demoerror.cpp",  
 "lineNum": 13,  
 "nextLine": " return 0",  
 "previousLine": "{",  
 "resDescr": "",  
 "srcResolved": ""  
 },  
 {  
 "colNum": 13,  
 "errorMessage": "error: expected ';' after return statement",  
 "file": "Code/toCompile/demoerror.cpp",  
 "lineNum": 14,  
 "nextLine": "",  
 "previousLine": " std::cout << \"in demoerror.cpp\" << endl;",  
 "resDescr": "",  
 "srcResolved": ""  
 }  
 ]  
}

**Reasoning:** Instead of using the actual file path, the LLM would occasionally return “file”. I was hoping being more specific about wanting the path & not the word file would help get more consistent results.

**Feedback:** not proper JSON format. I’m going to pass in the LLM output format as well

## Prompt 4

**Prompt:** This is the format of an error object:

```
{
  {'colNum' : #},
  {'errorMessage': ''},
  {'file': ''},
  {'lineNum': #},
  {'nextLine': ''},
  {'previousLine': ''},
  {'resDescr': ''},
  {'src': ''},
  {'srcResolved': ''}
```

```
}
```

For each error object given, provide the resolved C++ code in the 'srcResolved' member, and provide a description of the fix in the 'resDescr' member.

Leave the incorrect code in the 'src' member.

'srcResolved' should only contain valid c++ code.

Your response should be in the following format:

```
{
"path/to/file.cpp": [
{
{'colNum': #},
{'errorMessage': ""},
{'file': ""},
{'lineNum': #},
{'nextLine': ""},
{'previousLine': ""},
{'resDescr': ""},
{'src': ""},
{'srcResolved': ""}
},
{
{'colNum': #},
{'errorMessage': ""},
{'file': ""},
{'lineNum': #},
{'nextLine': ""},
{'previousLine': ""},
{'resDescr': ""},
{'src': ""},
{'srcResolved': ""}
}
]
}
```

Do not prompt your response.

**Errors passed:** demoerror.cpp

**Response:** {

```
"Code/toCompile/demoerror.cpp": [
{
{'colNum': 40, 'errorMessage': "error: use of undeclared identifier 'endl'; did you mean 'std::endl'?", 'file':
'Code/toCompile/demoerror.cpp', 'lineNum': 13, 'nextLine': '    return 0', 'previousLine': '{', 'resDescr':
"Replace the use of endl with std::endl in line 13.", 'src': '    std::cout << "in demoerror.cpp" << std::endl;',
'srcResolved': ""},
{
```

```
{'colNum' : 13, 'errorMessage': "error: expected ';' after return statement", 'file':
'Code/toCompile/demoerror.cpp', 'lineNum': 14, 'nextLine': ", 'previousLine': '    std::cout << "in
demoerror.cpp" << std::endl;', 'resDescr': "Add a semicolon after the return statement in line 14.", 'src': '
return 0;', 'srcResolved': ""}
]
}
```

**Reasoning:** I passed the output format I wanted to the LLM, so it would return results in valid json format

**Feedback:** invalid curly brace use – i passed output format incorrectly

## Prompt 5

**Prompt:** This is the format of an error object:

```
{
{'colNum' : #},
{'errorMessage': ""},
{'file': ""},
{'lineNum': #},
{'nextLine':""},
{'previousLine':""},
{'resDescr': ""},
{'src': ""},
{'srcResolved': ""}
}
```

For each error object given, provide the resolved C++ code in the 'srcResolved' member, and provide a description of the fix in the 'resDescr' member.

Leave the incorrect code in the 'src' member.

'srcResolved' should only contain valid c++ code.

Your response should be in the following format:

```
{
"path/to/file.cpp": [
{'colNum' : #,
'errorMessage': "",
'file': "",
'lineNum': #,
'nextLine': "",
'previousLine': "",
'resDescr': "",
'src': "",
'srcResolved': ""
},
{'colNum' : #,
'errorMessage': "",
'file': "",
```

```

'lineNum': #,
'nextLine':",
'previousLine':",
'resDescr': ",
'src': ",
'srcResolved': "
}
]
}

```

Do not prompt your response.

**Errors Passed:** demoerror.cpp

**Response :**

```

{
"path/to/file.cpp": [
{'colNum' : 40,
'errorMessage': "error: use of undeclared identifier 'endl'; did you mean 'std::endl'?", 'file':
'Code/to/Compile/demoerror.cpp', 'lineNum': 13, 'nextLine': '    return 0', 'previousLine': '{', 'resDescr':
'Replace 'endl' with 'std::endl'', 'src': '    std::cout << "in demoerror.cpp" << endl;', 'srcResolved': ""},
{'colNum' : 13, 'errorMessage': "error: expected ';' after return statement", 'file':
'Code/to/Compile/demoerror.cpp', 'lineNum': 14, 'nextLine': '    std::cout << "in
demoerror.cpp" << endl;', 'resDescr': "", 'src': '    return 0', 'srcResolved': ""}
]
}

```

**Reasoning:** LLM was including incorrect curly braces, so I removed them from encasing each error object attribute

**Feedback:** LLM did not change the “path/to/file.cpp” placeholder to the actual file name

## Prompt 6

**Prompt 6:** This is the format of an error object:

```

{
{'colNum' : #},
{'errorMessage': ""},
{'file': ""},
{'lineNum': #},
{'nextLine':""},
{'previousLine':""},
{'resDescr': ""},
{'src': ""},
{'srcResolved': ""}

```



```
}
```

For each error object given, provide the resolved C++ code in the 'srcResolved' member, and provide a description of the fix in the 'resDescr' member.

Leave the incorrect code in the 'src' member.

'srcResolved' should only contain valid c++ code.

Your response should be in the following format:

```
{
  "path/to/file.cpp": [
    {'colNum' : #,
     'errorMessage': ",
     'file': ",
     'lineNum': #,
     'nextLine':",
     'previousLine':",
     'resDescr': ",
     'src': ",
     'srcResolved': "
    }
  ],
  {'colNum' : #,
   'errorMessage': ",
   'file': ",
   'lineNum': #,
   'nextLine':",
   'previousLine':",
   'resDescr': ",
   'src': ",
   'srcResolved': "
  }
]
```

Replace "path/to/file.cpp" with the 'file' memeber in the error object.

Do not prompt your response.

**Errors Passed:** demoerror.cpp

**Response:**

```
{
  "Code/toCompile/demoerror.cpp": [
    {'colNum' : 40,
     'errorMessage': "error: use of undeclared identifier 'endl'; did you mean 'std::endl'?", 'file':
     'Code/toCompile/demoerror.cpp', 'lineNum': 13, 'nextLine': '    return 0', 'previousLine': '{', 'resDescr':
     "Replace the use of endl with std::endl to fix this error.", 'src': '    std::cout << "in demoerror.cpp" <<
     endl;', 'srcResolved': "}
```

```
{'colNum' : 13, 'errorMessage': "error: expected ';' after return statement", 'file':
'Code/toCompile/demoerror.cpp', 'lineNum': 14, 'nextLine': ", 'previousLine': '    std::cout << "in
demoerror.cpp" << endl;', 'resDescr': "", 'src': '    return 0', 'srcResolved': "}
]
```

**Reasoning:** Giving explicit instruction to change the placeholder file path

**Feedback:** Is not being properly written to json

## Prompt 7

**Prompt:** This is the format of an error object:

```
{
{'colNum' : #},
{'errorMessage': ""},
{'file': ""},
{'lineNum': #},
{'nextLine': ""},
{'previousLine': ""},
{'resDescr': ""},
{'src': ""},
{'srcResolved': ""}
}
```

For each error object given, provide the resolved C++ code in the 'srcResolved' member, and provide a description of the fix in the 'resDescr' member.

Leave the incorrect code in the 'src' member.

'srcResolved' should only contain valid c++ code.

Your response should be in the following format:

```
{
"path/to/file.cpp": [
{'colNum' : #},
{'errorMessage': ""},
{'file': ""},
{'lineNum': #},
{'nextLine': ""},
{'previousLine': ""},
{'resDescr': ""},
{'src': ""},
{'srcResolved': ""}
],{'colNum' : #},
{'errorMessage': ""},
{'file': ""},
{'lineNum': #},
{'nextLine': ""},
{'previousLine': ""},
{'resDescr': ""},
```

```
{'src': ""},
{'srcResolved': ""}
]
```

Replace "path/to/file.cpp" with the 'file' member in the error object.

Do not prompt your response.

**Errors Passed:** demoerror.json

**Response:** {

```
"Code/toCompile/demoerror.cpp": [
  {'colNum': 40, 'errorMessage': "error: use of undeclared identifier 'endl'; did you mean 'std::endl'?", 'file':
'Code/toCompile/demoerror.cpp', 'lineNum': 13, 'nextLine': '    return 0', 'previousLine': '{', 'resDescr':
'Replace the use of endl with std::endl.", 'src': '    std::cout << "in demoerror.cpp" << std::endl;',
'srcResolved': ""},
  {'colNum': 13, 'errorMessage': "error: expected ';' after return statement", 'file':
'Code/toCompile/demoerror.cpp', 'lineNum': 14, 'nextLine': "", 'previousLine': '    std::cout << "in
demoerror.cpp" << std::endl;', 'resDescr': "Add a semicolon after the return statement.", 'src': '    return 0;',
'srcResolved': ""}
]
```

**Reasoning:** Trying to produce proper json output

**Feedback:** Need error object attributes in double quotes not single quotes

## Prompt 8:

**Prompt:** This is the format of an error object:

```
{
  {'colNum': #},
  {'errorMessage': ""},
  {'file': ""},
  {'lineNum': #},
  {'nextLine': ""},
  {'previousLine': ""},
  {'resDescr': ""},
  {'src': ""},
  {'srcResolved': ""}
}
```

For each error object given, provide the resolved C++ code in the 'srcResolved' member, and provide a description of the fix in the 'resDescr' member.

Leave the incorrect code in the 'src' member.

'srcResolved' should only contain valid c++ code.

Your response should be in the following format:

```
{
  "path/to/file.cpp": [
    {"colNum": #},
    {"errorMessage": ""},
```

```

{"file": ""},
{"lineNum": #},
{"nextLine":""},
{"previousLine":""},
{"resDescr": ""},
{"src": ""},
{"srcResolved": ""}
,{"colNum" : #},
{"errorMessage": ""},
{"file": ""},
{"lineNum": #},
{"nextLine":""},
{"previousLine":""},
{"resDescr": ""},
{"src": ""},
{"srcResolved": ""}
]
}

```

Replace "path/to/file.cpp" with the 'file' memeber in the error object.

Do not prompt your response.

**Errors Passed:** errors.json

**Response:** {

```

"Code/toCompile/demoerror.cpp": [
{"colNum" : 40, "errorMessage": "error: use of undeclared identifier 'endl'; did you mean 'std::endl'",
"file": "Code/toCompile/demoerror.cpp", "lineNum": 13, "nextLine": "    return 0", "previousLine": "{",
"resDescr": "", "src": "    std::cout << \"in demoerror.cpp\" << endl;\", \"srcResolved\": \"\"},
{"colNum" : 13, "errorMessage": "error: expected ';' after return statement", "file":
"Code/toCompile/demoerror.cpp", "lineNum": 14, "nextLine": "", "previousLine": "    std::cout << \"in
demoerror.cpp\" << endl;\", \"resDescr\": \"\", \"src\": \"    return 0\", \"srcResolved\": \"\"}
]

```

**Reasoning:** Error object attributes need to be in double quotes for json reading

**Feedback:** LLM leaving some error attributes empty - going to try switching order of instructions

## Prompt 9

**Prompt:** This is the format of an error object:

```

{
{'colNum' : #},
{'errorMessage': ''},
{'file': ''},
{'lineNum': #},
{'nextLine': ''},
{'previousLine': ''},
{'resDescr': ''},

```

```
{'src': ""},
{'srcResolved': ""}
}
```

Your response should be in the following valid json format:

```
{
"path/to/file.cpp": [
{"colNum": #},
{"errorMessage": ""},
{"file": ""},
{"lineNum": #},
{"nextLine": ""},
{"previousLine": ""},
{"resDescr": ""},
{"src": ""},
{"srcResolved": ""}
,{"colNum": #},
{"errorMessage": ""},
{"file": ""},
{"lineNum": #},
{"nextLine": ""},
{"previousLine": ""},
{"resDescr": ""},
{"src": ""},
{"srcResolved": ""}
]
}
```

For each error object given, provide the resolved C++ code in the 'srcResolved' member, and provide a description of the fix in the 'resDescr' member.

Leave the incorrect code in the 'src' member.

**'srcResolved' should only contain valid c++ code. Replace "path/to/file.cpp" with the 'file' memeber in the error object.**

Do not prompt your response.

**Errors Passed:** demoerror.cpp

**Response:** {

```
"Code/toCompile/demoerror.cpp": [
{
"colNum": 40,
"errorMessage": "error: use of undeclared identifier 'endl'; did you mean 'std::endl'",
"file": "Code/toCompile/demoerror.cpp",
"lineNum": 13,
"nextLine": "    return 0",
"previousLine": "{",
"resDescr": "Replace the use of endl with std::endl.",
"srcResolved": "std::cout << \"in demoerror.cpp\" << std::endl;"
```

```

    },
    {
      "colNum": 13,
      "errorMessage": "error: expected ';' after return statement",
      "file": "Code/toCompile/demoerror.cpp",
      "lineNum": 14,
      "nextLine": "",
      "previousLine": "    std::cout << \"in demoerror.cpp\" << std::endl;",
      "resDescr": "Add a semicolon after the return statement.",
      "srcResolved": "return 0;"
    }
  ]
}

```

**Reasoning:** Changed the orders of instructions so the actual task for the LLM comes after the output format description

**Feedback:** This is the output I was looking for

**Errors Passed:** onesimpleerror.cpp

```

Response: {
  "path/to/oneerrorsimple.cpp": [
    {
      "colNum": 13,
      "errorMessage": "error: expected ';' after return statement",
      "file": "Code/toCompile/oneerrorsimple.cpp",
      "lineNum": 5,
      "nextLine": "",
      "previousLine": "    std::cout << \"in oneerrorsimple.cpp\" << std::endl;",
      "resDescr": "Add a semicolon after the return statement.",
      "src": "    return 0",
      "srcResolved": "    return 0;"
    }
  ]
}

```

**Reasoning:** Because the prompt output format is an example where there are multiple errors, I wanted to test with a file that only produced one error to ensure the LLM still produced the proper format.

**Feedback:** File path is inconsistent, sometimes it is correct and others it is not.

## Prompt 11:

**Prompt:** This is the format of an error object:

```

{
  {'colNum': #},
  {'errorMessage': ""},
  {'file': ""},
  {'lineNum': #},

```

```
{'nextLine':"},
{'previousLine':"},
{'resDescr': ""},
{'src': ""},
{'srcResolved': ""}
}
```

Your response should be in the following valid json format:

```
{
  "file.cpp": [
    {"colNum" : #,
    "errorMessage": "",
    "file": "",
    "lineNum": #,
    "nextLine": "",
    "previousLine": "",
    "resDescr": "",
    "src": "",
    "srcResolved": ""}
    ,{"colNum" : #,
    "errorMessage": "",
    "file": "",
    "lineNum": #,
    "nextLine": "",
    "previousLine": "",
    "resDescr": "",
    "src": "",
    "srcResolved": ""}
  ]
}
```

For each error object given, provide the resolved C++ code in the 'srcResolved' member, and provide a description of the fix in the 'resDescr' member.

Leave the incorrect code in the 'src' member.

'srcResolved' should only contain valid c++ code. Replace "file.cpp" with the 'file' memeber in the error object.

Do not prompt your response.

**Errors Passed:** onesimpleerror.cpp

```
Response: {
  "file.cpp": [
    {
      "colNum" : 13,
      "errorMessage": "error: expected ';' after return statement",
      "file": "Code/toCompile/oneerrorsimple.cpp",
      "lineNum": 5,
      "nextLine": ""},

```

```

"previousLine": "    std::cout << \"in oneerrorsimple.cpp\" << std::endl;\",
"resDescr": "Add a semicolon after the return statement.",
"srcResolved": "return 0;\"
}
]
}

```

**Reasoning:** Changed so the LLM would input the actual cpp file

**Feedback:** After battling with this for a while, I decided instead of trying to develop a prompt that produced consistent results 100% of the time, I would implement a check in the coderepair job execution function that will check if the file name in the error object and the file name the error object is nested under are the same.

## Flowscript Generation Prompt

### Prompt 1

**Prompt:** Flowscript is a simple custom DSL for building process flows. FFlowscript specifics are provided below:

Keywords: "digraph", "subgraph", "label", "shape", "true", "false", "point", "none"

Job Types: "compile", "errorparse", "rest", "coderepair", "show", "test"

Lexical Tokens:

1. KEYWORD - any of the words in the keywords list OR the job types list
2. IDENTIFIER - names for the jobs
3. OPERATOR - "=" or "->"
  - (a) = operator is used to assign values to "label" & "shape"
  - (b) -> operator is used to indicate the job before the operator must finish before the job after the operator can execute
4. BRACKET - "[" or "]"
5. BLOCK - "{" or "}"
6. TERMINATOR - ";\n"
  - (a) every line except for BLOCK\_TITLES and BLOCK\_BOUNDS must be concluded with a TERMINATOR

Valid Syntax:

1. BLOCK\_TITLE : {KEYWORD, IDENTIFIER}
2. DEPENDENT\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
3. UI\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, IDENTIFIER, BRACKET, TERMINATOR}
4. END\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
5. EXECUTIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, TERMINATOR}



6. CONDIITONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR }

7. BLOCK\_BOUND : {BLOCK}

8. START\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}

Here is example flow script for a process that creates a UI job with the input "demo". if the job fails, an errorparse job is created and run before the process completes. If the test job does not fail, a 'show' job is run and then the process completes:

```Flowscript

digraph exampleflow

```
{
subgraph clusterExample
{
start[shape=none];
a[label=demo];
b[label=errorparse];
c[label=show]
end[shape=point];
a->b[label=false];
a->c[label=true];
c->end;
b->end;
}
}
```

```

This example follows the following syntactical format:

<BLOCK\_TITLE>

<BLOCK\_BOUND>

<BLOCK\_TITLE>

<BLOCK\_BOUND>

<START\_POINT>

<UI\_JOB>

<DEPENDENT\_JOB>

<DEPENDENT\_JOB>

<END\_POINT>

<CONDIITONAL\_DEPENDENCY>

<CONDIITONAL\_DEPENDENCY>

<EXECUTIONAL\_DEPENDENCY>

<EXECUTIONAL\_DEPENDENCY>

Create a Flowscript script that creates a job with input "demoerror". If demoerror job fails, an errorparse job is created and run, then a job with the input "restjob" is run, then the process exits. If the compile job does not fail, the process exits.

Do not include anything other than the Flowscript script in your response

**Response:** ```Flowscript

digraph exampleflow

{

subgraph clusterExample

{

start[shape=none];

a[label=demoerror];

b[label=restjob];

c[label=compile];

d[label=errorparse];

e[label=end]

f[label="false"]

g[label="true"];

h[label="not failed"]

i[label="failed"]

j[label="executed"]

k[label="not executed"]

l[label="completed"]

m[label="incomplete"]

}

a->b[label=false];

a->c[label=true];

c->d[label=h];

d->e[label=i];

f->g;

g->h;

h->j;

j->k;

m->l;

l->end;

}

```

**Reasoning:** For the base of this prompt, I thought it was important to explain the specifics of the Flowscript language. I prompted the LLM with the list of keywords, job types, and valid operators. Additionally, I passed in a list of the Flowscript lexical tokens and explained what each token was, what values they can hold, and any basic rules about the specific token. Next, I provided the list of syntactically valid combinations of these lexical tokens. Following that, I explain in English, what the example Flowscript code is doing before providing the example, and then the syntactical format of the example. Finally, I instruct the LLM to produce the Flowscript code for the specific process I want to execute, and tell it to return only the Flowscript code. Overall, I thought it was important to just include the very basic rules of Flowscript because the example I provided can serve to imply other rules, thus the LLM won't have to read in extra information.

**Feedback:** This response is very far from the correct Flowscript code for the process I requested.

## Prompt 2

**Prompt:** Flowscript is a simple custom DSL for building process flows. FLOWscript specifics are provided below:

Keywords: "digraph", "subgraph", "label", "shape", "true", "false", "point", "none"

Job Types: "compile", "errorparse", "rest", "coderepair", "show", "test"

Lexical Tokens:

1. KEYWORD - any of the words in the keywords list OR the job types list
2. IDENTIFIER - names for the jobs
3. OPERATOR - "=" or "->"
  - (a) = operator is used to assign values to "label" & "shape"
  - (b) -> operator is used to indicate the job before the operator must finish before the job after the operator can execute
4. BRACKET - "[" or "]"
5. BLOCK - "{" or "}"
6. TERMINATOR - ";"

(a) every line except for BLOCK\_TITLES and BLOCK\_BOUNDS must be concluded with a TERMINATOR

Valid Syntax:

1. BLOCK\_TITLE : {KEYWORD, IDENTIFIER}
2. DEPENDENT\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
3. UI\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, IDENTIFIER, BRACKET, TERMINATOR}
4. END\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
  - (a) required to indicate process end point
5. EXECUTIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, TERMINATOR}
6. CONDITIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR }
  - (a) keyword 1 must be label
  - (b) keyword 2 must be true or false
7. BLOCK\_BOUND : {BLOCK}
8. START\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
  - (a) required to indicate the first job in the process

Here is example flow script for a process that creates a UI job with the input "demo". if the job fails ("false"), an errorparse job is created and run before the process completes. If the test job does not fail ('true'), a 'show' job is run and then the process completes:

**Flowscript**

```
digraph exampleflow
{
  subgraph clusterExample
  {
    start[shape=none];
```

```

a[label=demo];
b[label=errorparse];
c[label=show]
end[shape=point];
a->b[label=false];
a->c[label=true];
c->end;
b->end;
}
}
...

```

This example follows the following syntactical format:

```

<BLOCK_TITLE>
<BLOCK_BOUND>
<BLOCK_TITLE>
<BLOCK_BOUND>
<START_POINT>
<UI_JOB>
<DEPENDENT_JOB>
<DEPENDENT_JOB>
<END_POINT>
<CONDIITONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<EXECUTIONAL_DEPENDENCY>
<EXECUTIONAL_DEPENDENCY>

```

Create job A with input "demoerror"

If job A is successful, the process ends

If A fails , create an errorparse job, job B

After job B finishes, create job C with input "restjob"

Affter job C finishes, create a coderepair job, job D

if Job D fails, the process ends

If job D is successful, job A executes again

Do not include anything other than the Flowscript script in your response

**Response:** Thank you for providing the example flow script. It looks like it follows the syntax specified in the documentation. Is there anything else I can help with?

**Reasoning:** I added some of the syntax specific rules to the valid syntax list, so the LLM would know to put either true or false for a dependency label. I also changed the format of how I am describing the process to be created, so it is more clear to the LLM how many jobs should be created in the script.

**Feedback:** LLM did not understand I was asking it to do something.

### Prompt 3

**Prompt:** Flowscript is a simple custom DSL for building process flows. FLOWscript specifics are provided below:

Keywords: "digraph", "subgraph", "label", "shape", "true", "false", "point", "none"

Job Types: "compile", "errorparse", "rest", "coderepair", "show", "test"

Lexical Tokens:

1. KEYWORD - any of the words in the keywords list OR the job types list
2. IDENTIFIER - names for the jobs
3. OPERATOR - "=" or "->"
  - (a) = operator is used to assign values to "label" & "shape"
  - (b) -> operator is used to indicate the job before the operator must finish before the job after the operator can execute
4. BRACKET - "[" or "]"
5. BLOCK - "{" or "}"
6. TERMINATOR - ";"
  - (a) every line except for BLOCK\_TITLEs and BLOCK\_BOUNDS must be concluded with a TERMINATOR

Valid Syntax:

1. BLOCK\_TITLE : {KEYWORD, IDENTIFIER}
2. DEPENDENT\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
3. UI\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, IDENTIFIER, BRACKET, TERMINATOR}
4. END\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
  - (a) required to indicate process end point
5. EXECUTIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, TERMINATOR}
6. CONDITIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR }
  - (a) keyword 1 must be label
  - (b) keyword 2 must be true or false
7. BLOCK\_BOUND : {BLOCK}
8. START\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
  - (a) required to indicate the first job in the process

Here is example flow script for a process that creates a UI job with the input "demo". if the job fails ("false"), an errorparse job is created and run before the process completes. If the test job does not fail ('true'), a 'show' job is run and then the process completes:

...

digraph exampleflow

```
{
subgraph clusterExample
{
start[shape=none];
a[label=demo];
b[label=errorparse];
c[label=show]
```

```

end[shape=point];
a->b[label=false];
a->c[label=true];
c->end;
b->end;
}
}
'''

```

This example follows the following syntactical format:

```

<BLOCK_TITLE>
<BLOCK_BOUND>
<BLOCK_TITLE>
<BLOCK_BOUND>
<START_POINT>
<UI_JOB>
<DEPENDENT_JOB>
<DEPENDENT_JOB>
<END_POINT>
<CONDIITONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<EXECUTIONAL_DEPENDENCY>
<EXECUTIONAL_DEPENDENCY>

```

-----

Your task is to generate the Flowscript script for the following process flow:

Create job A with input "demoerror"

If job A is successful, the process ends

If A fails , create an errorparse job, job B

After job B finishes, create job C with input "restjob"

Affter job C finishes, create a coderepair job, job D

if Job D fails, the process ends

If job D is successful, job A executes again

Do not include anything other than the Flowscript script in your response

**Response:** ```

```

digraph exampleflow
{
subgraph clusterExample
{
start[shape=none];
a[label=demoerror];
b[label=errorparse];
c[label=restjob];
d[label=coderepair];
end[shape=point];
a->b[label=false];

```

```

b->c;
c->d[label=true];
d->start;
}
}
...

```

**Reasoning:** I added the explicit line about what the LLM's task is so it understands I am trying to get it to do something and not just providing it with information about Flowscript.

**Feedback:** I forgot to set the start node in the example I provided to the LLM.

## Prompt 4

**Prompt:** Flowscript is a simple custom DSL for building process flows. Flowscript specifics are provided below:

Keywords: "digraph", "subgraph", "label", "shape", "true", "false", "point", "none"

Job Types: "compile", "errorparse", "rest", "coderepair", "show", "test"

Lexical Tokens:

1. KEYWORD - any of the words in the keywords list OR the job types list
2. IDENTIFIER - names for the jobs
3. OPERATOR - "=" or "->"
  - (a) = operator is used to assign values to "label" & "shape"
  - (b) -> operator is used to indicate the job before the operator must finish before the job after the operator can execute
4. BRACKET - "[" or "]"
5. BLOCK - "{" or "}"
6. TERMINATOR - ";"
  - (a) every line except for BLOCK\_TITLES and BLOCK\_BOUNDS must be concluded with a TERMINATOR

Valid Syntax:

1. BLOCK\_TITLE : {KEYWORD, IDENTIFIER}
2. DEPENDENT\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
3. UI\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, IDENTIFIER, BRACKET, TERMINATOR}
4. END\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
  - (a) required to indicate process end point
5. EXECUTIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, TERMINATOR}
6. CONDITIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR }
  - (a) keyword 1 must be label
  - (b) keyword 2 must be true or false
7. BLOCK\_BOUND : {BLOCK}
8. START\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}

(a) required to indicate the first job in the process

Here is example flow script for a process that creates a UI job with the input "demo". if the job fails ("false"), an errorparse job is created and run before the process completes. If the test job does not fail ('true'), a 'show' job is run and then the process completes:

```
...  
digraph exampleflow  
{  
  subgraph clusterExample  
  {  
    start[shape=none];  
    a[label=demo];  
    b[label=errorparse];  
    c[label=show]  
    end[shape=point];  
    start->a  
    a->b[label=false];  
    a->c[label=true];  
    c->end;  
    b->end;  
  }  
}  
...
```

This example follows the following syntactical format:

```
<BLOCK_TITLE>  
<BLOCK_BOUND>  
<BLOCK_TITLE>  
<BLOCK_BOUND>  
<START_POINT>  
<UI_JOB>  
<DEPENDENT_JOB>  
<DEPENDENT_JOB>  
<END_POINT>  
<EXECUTIONAL_DEPENDENCY>  
<CONDIITONAL_DEPENDENCY>  
<CONDIITONAL_DEPENDENCY>  
<EXECUTIONAL_DEPENDENCY>  
<EXECUTIONAL_DEPENDENCY>
```

-----

Your task is to generate the Flowscript script for the following process flow:

Create job A with input "demoerror"

If job A is successful, the process ends

If A fails , create an errorparse job, job B

After job B finishes, create job C with input "restjob"

Affter job C finishes, create a coderepair job, job D



if Job D fails, the process ends

If job D is successful, job A executes again

Do not include anything other than the Flowscript script in your response

**Response:** ```

```
digraph exampleflow
{
  subgraph clusterExample
  {
    start[shape=none];
    a[label=demoerror];
    b[label=errorparse];
    c[label=restjob];
    d[label=coderepair];
    end[shape=point];
    start->a;
    a->b[label=false];
    b->c[label=true];
    c->d[label=false];
    d->end;
  }
}
```
```

**Reasoning:** I added the line to set the start node in the example Flowscript code because I forgot to include that initially.

**Feedback:** Start node is now also set properly in the LLM generated Flowscript, but the process being output is incorrect. It's only allowing for one possible flow of execution.

## Prompt 5

**Prompt:** Flowscript is a simple custom DSL for building process flows. FLOWscript specifics are provided below:

Keywords: "digraph", "subgraph", "label", "shape", "true", "false", "point", "none"

Job Types: "compile", "errorparse", "rest", "coderepair", "show", "test"

Lexical Tokens:

1. KEYWORD - any of the words in the keywords list OR the job types list
2. IDENTIFIER - names for the jobs
3. OPERATOR - "=" or "->"
  - (a) = operator is used to assign values to "label" & "shape"
  - (b) -> operator is used to indicate the job before the operator must finish before the job after the operator can execute
4. BRACKET - "[" or "]"
5. BLOCK - "{" or "}"
6. TERMINATOR - ";;"
  - (a) every line except for BLOCK\_TITLES and BLOCK\_BOUNDS must be concluded with a TERMINATOR

Valid Syntax:

1. BLOCK\_TITLE : {KEYWORD, IDENTIFIER}
2. DEPENDENT\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
3. UI\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, IDENTIFIER, BRACKET, TERMINATOR}
4. END\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
  - (a) required to indicate process end point
5. EXECUTIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, TERMINATOR}
6. CONDITIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR }
  - (a) keyword 1 must be label
  - (b) keyword 2 must be true or false
7. BLOCK\_BOUND : {BLOCK}
8. START\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
  - (a) required to indicate the first job in the process

Here is example flow script for a process that creates a UI job with the input "demo". if the job fails ("false"), an errorparse job is created and run before the process completes. If the test job does not fail ('true'), a 'show' job is run and then the process completes:

...

digraph exampleflow

```
{
subgraph clusterExample
{
start[shape=none];
a[label=demo];
b[label=errorparse];
c[label=show]
end[shape=point];
start->a
a->b[label=false];
a->c[label=true];
c->end;
b->end;
}
}
...
```

This example follows the following syntactical format:

```
<BLOCK_TITLE>
<BLOCK_BOUND>
<BLOCK_TITLE>
<BLOCK_BOUND>
```

```

<START_POINT>
<UI_JOB>
<DEPENDENT_JOB>
<DEPENDENT_JOB>
<END_POINT>
<EXECUTIONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<EXECUTIONAL_DEPENDENCY>
<EXECUTIONAL_DEPENDENCY>

```

-----

Your task is to generate the Flowscript script for the following process flow:

Create job A with input "demoerror"

If job A is successful, the process ends **and no other jobs should be created or executed**

If A fails , create an errorparse job, job B

After job B finishes, create job C with input "restjob"

Affter job C finishes, create a coderepair job, job D

if Job D fails, the process ends

If job D is successful, job A executes again

Do not include anything other than the Flowscript script in your response

**Response:** ```

```

digraph exampleflow
{
subgraph clusterExample
{
start[shape=none];
a[label=demoerror];
b[label=errorparse];
c[label=restjob];
d[label=coderepair];
end[shape=point];
start->a;
a->b[label=false];
b->c[label=true];
c->d[label=false];
d->end;
}
}
```

```

**Reasoning:** I added the line about no other jobs being executed to indicate to the LLM that there should be multiple execution paths in the Flowscript.

**Feedback:** Produced the same one-track process flow.

## Prompt 6

**Prompt:** Flowscript is a simple custom DSL for building process flows. FLOWscript specifics are provided below:

Keywords: "digraph", "subgraph", "label", "shape", "true", "false", "point", "none"

Job Types: "compile", "errorparse", "rest", "coderepair", "show", "test"

Lexical Tokens:

1. KEYWORD - any of the words in the keywords list OR the job types list
2. IDENTIFIER - names for the jobs
3. OPERATOR - "=" or "->"
  - (a) = operator is used to assign values to "label" & "shape"
  - (b) -> operator is used to indicate the job before the operator must finish before the job after the operator can execute
4. BRACKET - "[" or "]"
5. BLOCK - "{" or "}"
6. TERMINATOR - ";;"

(a) every line except for BLOCK\_TITLES and BLOCK\_BOUNDS must be concluded with a TERMINATOR

Valid Syntax:

1. BLOCK\_TITLE : {KEYWORD, IDENTIFIER}
2. DEPENDENT\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
3. UI\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, IDENTIFIER, BRACKET, TERMINATOR}
4. END\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
  - (a) required to indicate process end point
5. EXECUTIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, TERMINATOR}
6. CONDITIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR }
  - (a) keyword 1 must be label
  - (b) keyword 2 must be true or false
7. BLOCK\_BOUND : {BLOCK}
8. START\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}

(a) required to indicate the first job in the process

In this example:

Job A is created with input "demo"

if Job A is successful ("true"), the process ends.

If job A fails ("false"), an errorparse job, job B is created.

After Job B finishes, a show job, job C is created.

After Job C finishes the process is complete

...

digraph exampleflow

{

```

subgraph clusterExample
{
start[shape=none];
a[label=demo];
b[label=errorparse];
c[label=show]
end[shape=point];
start->a
a->b[label=false];
a->end[label=true];
c->end;
b->c;
}
}
```

```

This example follows the following syntactical format:

```

<BLOCK_TITLE>
<BLOCK_BOUND>
<BLOCK_TITLE>
<BLOCK_BOUND>
<START_POINT>
<UI_JOB>
<DEPENDENT_JOB>
<DEPENDENT_JOB>
<END_POINT>
<EXECUTIONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<EXECUTIONAL_DEPENDENCY>
<EXECUTIONAL_DEPENDENCY>

```

-----

Your task is to generate the Flowscript script for the following process flow:

Create job A with input "demoerror"

If job A is successful, the process ends and no other jobs should be created or executed

If A fails , create an errorparse job, job B

After job B finishes, create job C with input "restjob"

Affter job C finishes, create a coderepair job, job D

if Job D fails, the process ends

If job D is successful, job A executes again

Do not include anything other than the Flowscript script in your response

**Response:** ```

```

digraph exampleflow
{
subgraph clusterExample

```

```

{
start[shape=none];
a[label=demoerror];
b[label=errorparse];
c[label=restjob];
d[label=coderepair];
end[shape=point];
start->a;
a->b[label=false];
b->c;
c->d;
d->end;
}
}
...

```

**Reasoning:** I changed the format of the Flowscript example explanation to match the format of the description of the process the LLM is being requested to write the script for.

**Feedback:** LLM script is only providing one option for conditional dependencies. Need both true & false options in script for conditional execution.

## Prompt 7

**Prompt:** Flowscript is a simple custom DSL for building process flows. FLOWscript specifics are provided below:

Keywords: "digraph", "subgraph", "label", "shape", "true", "false", "point", "none"

Job Types: "compile", "errorparse", "rest", "coderepair", "show", "test"

Lexical Tokens:

1. KEYWORD - any of the words in the keywords list OR the job types list
2. IDENTIFIER - names for the jobs
3. OPERATOR - "=" or "->"
  - (a) = operator is used to assign values to "label" & "shape"
  - (b) -> operator is used to indicate the job before the operator must finish before the job after the operator can execute
4. BRACKET - "[" or "]"
5. BLOCK - "{" or "}"
6. TERMINATOR - ";"
  - (a) every line except for BLOCK\_TITLES and BLOCK\_BOUNDS must be concluded with a TERMINATOR

Valid Syntax:

1. BLOCK\_TITLE : {KEYWORD, IDENTIFIER}
2. DEPENDENT\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
3. UI\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, IDENTIFIER, BRACKET, TERMINATOR}

4. END\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}

(a) required to indicate process end point

5. EXECUTIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, TERMINATOR}

6. CONDITIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR }

(a) keyword 1 must be label

(b) keyword 2 must be true or false

(C) any job that has a conditional dependency is required to have a both a job to execute if true and if false

7. BLOCK\_BOUND : {BLOCK}

8. START\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}

(a) required to indicate the first job in the process

In this example:

Job A is created with input "demo"

if Job A is successful ("true"), the process ends and no other jobs are created or executed.

If job A fails ("false"), an errorparse job, job B is created.

After Job B finishes, a show job, job C is created.

After Job C finishes the process is complete

...

digraph exampleflow

{

subgraph clusterExample

{

start[shape=none];

a[label=demo];

b[label=errorparse];

c[label=show]

end[shape=point];

start->a

a->b[label=false];

a->end[label=true];

c->end;

b->c;

}

}

...

This example follows the following syntactical format:

<BLOCK\_TITLE>

<BLOCK\_BOUND>

<BLOCK\_TITLE>

<BLOCK\_BOUND>

<START\_POINT>

```

<UI_JOB>
<DEPENDENT_JOB>
<DEPENDENT_JOB>
<END_POINT>
<EXECUTIONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<EXECUTIONAL_DEPENDENCY>
<EXECUTIONAL_DEPENDENCY>

```

-----

Your task is to generate the Flowscript script for the following process flow:

Create job A with input "demoerror"

If job A is successful, the process ends and no other jobs should be created or executed

If A fails , create an errorparse job, job B

After job B finishes, create job C with input "restjob"

Affter job C finishes, create a coderepair job, job D

if Job D fails, the process ends

If job D is successful, job A executes again

Do not include anything other than the Flowscript script in your response

**Response:** ```

```

digraph exampleflow
{
subgraph clusterExample
{
start[shape=none];
a[label=demoerror];
b[label=errorparse];
c[label=restjob];
d[label=coderepair];
end[shape=point];
start->a;
a->b[label=false];
a->end[label=true];
b->c;
c->d;
d->end;
}
}
```

```

**Reasoning:** I added the line about all conditional dependencies requiring both a true and false path so the LLM would not generate only one of the two options. I also added the line in the example about no other jobs being executed, so it would match the verbiage in the prompt for the specific process script to be written.



**Feedback:** Not registering the second conditional dependency, and occasionally uses “success” or “failure” instead of true or false for value labels.

## Prompt 8

Prompt: Flowscript is a simple custom DSL for building process flows. FLoowscript specifics are provided below:

Keywords: "digraph", "subgraph", "label", "shape", "true", "false", "point", "none"

Job Types: "compile", "errorparse", "rest", "coderepair", "show", "test"

Lexical Tokens:

1. KEYWORD - any of the words in the keywords list OR the job types list
2. IDENTIFIER - names for the jobs
3. OPERATOR - "=" or "->"
  - (a) = operator is used to assign values to "label" & "shape"
  - (b) -> operator is used to indicate the job before the operator must finish before the job after the operator can execute
4. BRACKET - "[" or "]"
5. BLOCK - "{" or "}"
6. TERMINATOR - ";"
  - (a) every line except for BLOCK\_TITLES and BLOCK\_BOUNDS must be concluded with a TERMINATOR

Valid Syntax:

1. BLOCK\_TITLE : {KEYWORD, IDENTIFIER}
2. DEPENDENT\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
3. UI\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, IDENTIFIER, BRACKET, TERMINATOR}
4. END\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
  - (a) required to indicate process end point
5. EXECUTIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, TERMINATOR}
6. CONDITIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR }
  - (a) dependency labels can only be assigned to true or false
  - (b) any job that has a conditional dependency is required to have a both a job to execute if true and if false
7. BLOCK\_BOUND : {BLOCK}
8. START\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
  - (a) required to indicate the first job in the process

In this example:

Job A is created with input "demo"

if Job A is successful, the process ends and no other jobs are created or executed.

If job A fails, an errorparse job, job B is created.

After Job B finishes, a show job, job C is created.

After Job C finishes the process is complete

```

```
digraph exampleflow
{
subgraph clusterExample
{
start[shape=none];
a[label=demo];
b[label=errorparse];
c[label=show]
end[shape=point];
start->a
a->b[label=false];
a->end[label=true];
c->end;
b->c;
}
}
}```
```

This example follows the following syntactical format:

```
<BLOCK_TITLE>
<BLOCK_BOUND>
<BLOCK_TITLE>
<BLOCK_BOUND>
<START_POINT>
<UI_JOB>
<DEPENDENT_JOB>
<DEPENDENT_JOB>
<END_POINT>
<EXECUTIONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<EXECUTIONAL_DEPENDENCY>
<EXECUTIONAL_DEPENDENCY>
```

-----

Your task is to generate the Flowscript script for the following process flow:

Create job A with input "demoerror"

If job A is successful, the process ends and no other jobs should be created or executed

If A fails , create an errorparse job, job B

After job B finishes, create job C with input "restjob"

Affter job C finishes, create a coderepair job, job D

If job D is successful, job A executes again

if Job D fails, the process ends

Do not include anything other than the Flowscript script in your response

**Response:** ```

```
digraph exampleflow
{
subgraph clusterExample
{
start[shape=none];
a[label=demoerror];
b[label=errorparse];
c[label=restjob];
d[label=coderepair];
end[shape=point];
start->a;
a->b[label=false];
a->end[label=true];
b->c;
c->d;
d->a[label=false];
d->end[label=true];
}
}
```
```

**Reasoning:** I added the first change about dependency labels to indicate to the LLM that “true” and “false” are the only valid values for a dependency label. I also switched the order of the 2nd & 3rd to last lines of the prompt so the LLM does not read “the process ends” and think the process description is over.

**Feedback:** This script is almost exactly correct, but the dependency labels for job D dependencies are reversed.

## Prompt 9

**Prompt:** Flowscript is a simple custom DSL for building process flows. FLOWscript specifics are provided below:

Keywords: "digraph", "subgraph", "label", "shape", "true", "false", "point", "none"

Job Types: "compile", "errorparse", "rest", "coderepair", "show", "test"

Lexical Tokens:

1. KEYWORD - any of the words in the keywords list OR the job types list
2. IDENTIFIER - names for the jobs
3. OPERATOR - "=" or "->"
  - (a) = operator is used to assign values to "label" & "shape"
  - (b) -> operator is used to indicate the job before the operator must finish before the job after the operator can execute
4. BRACKET - "[" or "]"
5. BLOCK - "{" or "}"
6. TERMINATOR - ";"

(a) every line except for BLOCK\_TITLES and BLOCK\_BOUNDS must be concluded with a TERMINATOR

Valid Syntax:

1. BLOCK\_TITLE : {KEYWORD, IDENTIFIER}
2. DEPENDENT\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
3. UI\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, IDENTIFIER, BRACKET, TERMINATOR}
4. END\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}

(a) required to indicate process end point

5. EXECUTIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, TERMINATOR}
6. CONDITIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR }

(a) dependency labels can only be assigned to true or false

- success is "true"

- fail is "false"

(b) any job that has a conditional dependency is required to have a both a job to execute if true and if false

7. BLOCK\_BOUND : {BLOCK}
8. START\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}

(a) required to indicate the first job in the process

In this example:

Job A is created with input "demo"

if Job A is successful, the process ends and no other jobs are created or executed.

If job A fails, an errorparse job, job B is created.

After Job B finishes, a show job, job C is created.

After Job C finishes the process is complete

...

digraph exampleflow

{

subgraph clusterExample

{

start[shape=none];

a[label=demo];

b[label=errorparse];

c[label=show]

end[shape=point];

start->a

a->b[label=false];

a->end[label=true];

c->end;

b->c;

```
}  
}  
``
```

This example follows the following syntactical format:

```
<BLOCK_TITLE>  
<BLOCK_BOUND>  
<BLOCK_TITLE>  
<BLOCK_BOUND>  
<START_POINT>  
<UI_JOB>  
<DEPENDENT_JOB>  
<DEPENDENT_JOB>  
<END_POINT>  
<EXECUTIONAL_DEPENDENCY>  
<CONDIITONAL_DEPENDENCY>  
<CONDIITONAL_DEPENDENCY>  
<EXECUTIONAL_DEPENDENCY>  
<EXECUTIONAL_DEPENDENCY>
```

-----

Your task is to generate the Flowscript script for the following process flow:

Create job A with input "demoerror"

If job A is successful, the process ends and no other jobs should be created or executed

If A fails , create an errorparse job, job B

After job B finishes, create job C with input "restjob"

Affter job C finishes, create a coderepair job, job D

If job D is successful, job A executes again

if Job D fails, the process ends

Do not include anything other than the Flowscript script in your response

**Response:** ``

digraph exampleflow

{

subgraph clusterExample

{

start[shape=none];

a[label=demoerror];

b[label=errorparse];

c[label=restjob];

d[label=coderepair];

end[shape=point];

start->a;

a->b[label=false];

a->end[label=true];

b->c;

c->d;

```
d->a;  
}  
}  
...
```

**Reasoning:** I added the sub-bullet points about which values equate to true and false, so the LLM maps the conditional dependency values correctly.

**Feedback:** The LLM did not register the 2nd conditional dependency in the requested process. Further runs also sometimes registered the 2nd conditional dependency, but would only produce one of the two execution paths.

## Prompt 10

**Prompt:** Flowscript is a simple custom DSL for building process flows. FFlowscript specifics are provided below:

Keywords: "digraph", "subgraph", "label", "shape", "true", "false", "point", "none"

Job Types: "compile", "errorparse", "rest", "coderepair", "show", "test"

Lexical Tokens:

1. KEYWORD - any of the words in the keywords list OR the job types list
2. IDENTIFIER - names for the jobs
3. OPERATOR - "=" or "->"
  - (a) = operator is used to assign values to "label" & "shape"
  - (b) -> operator is used to indicate the job before the operator must finish before the job after the operator can execute
4. BRACKET - "[" or "]"
5. BLOCK - "{" or "}"
6. TERMINATOR - ";"

(a) every line except for BLOCK\_TITLES and BLOCK\_BOUNDS must be concluded with a TERMINATOR

Valid Syntax:

1. BLOCK\_TITLE : {KEYWORD, IDENTIFIER}
2. DEPENDENT\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
3. UI\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, IDENTIFIER, BRACKET, TERMINATOR}
4. END\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
5. EXECUTIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, TERMINATOR}
6. CONDITIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR }

(a) dependency labels can only be assigned to true or false

- success is "true"
- fail is "false"

(b) any job that has a conditional dependency is required to have a both a job to execute if true and if false

7. BLOCK\_BOUND : {BLOCK}

8. START\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}

(a) required to indicate the first job in the process

In this example:

Job A is created with input "demo"

if Job A is successful, the process ends and no other jobs are created or executed.

If job A fails, an errorparse job, job B is created.

After Job B finishes, a show job, job C is created.

After Job C finishes the process is complete

...

digraph exampleflow

{

subgraph clusterExample

{

start[shape=none];

a[label=demo];

b[label=errorparse];

c[label=show]

end[shape=point];

start->a

a->b[label=false];

a->end[label=true];

c->end;

b->c;

}

}

...

This example follows the following syntactical format:

<BLOCK\_TITLE>

<BLOCK\_BOUND>

<BLOCK\_TITLE>

<BLOCK\_BOUND>

<START\_POINT>

<UI\_JOB>

<DEPENDENT\_JOB>

<DEPENDENT\_JOB>

<END\_POINT>

<EXECUTIONAL\_DEPENDENCY>

<CONDIITONAL\_DEPENDENCY>

<CONDIITONAL\_DEPENDENCY>

<EXECUTIONAL\_DEPENDENCY>

<EXECUTIONAL\_DEPENDENCY>

-----

!! any job that has a conditional dependency is required to have a both a job to execute if true and if false

Your task is to generate the Flowscript script for the following process flow:

Create job A with input "demoerror"

If job A is successful, the process ends and no other jobs should be created or executed

If A fails , create an errorparse job, job B

After job B finishes, create job C with input "restjob"

After job C finishes, create a coderepair job, job D

If job D is successful, job A executes again

if Job D fails, the process ends

Do not include anything other than the Flowscript script in your response

**Response:** ```

```
digraph exampleflow
{
  subgraph clusterExample
  {
    start[shape=none];
    a[label=demoerror];
    b[label=errorparse];
    c[label=restjob];
    d[label=coderepair];
    end[shape=point];
    start->a;
    a->b[label=false];
    b->c;
    c->d[label=true];
    d->a[label=true];
  }
}
```
```

**Reasoning:** I added the first line about conditional dependencies to inform the LLM that all jobs with a conditional dependency should have two possible flows of execution following their execution.

**Feedback:** Conditional dependencies are still incorrect – missing the path if true for job A and the path if false for job D, and the dependency from job C to job D should not be conditional.

## Prompt 11

**Prompt:** Flowscript is a simple custom DSL for building process flows. Flowscript specifics are provided below:

Keywords: "digraph", "subgraph", "label", "shape", "true", "false", "point", "none"

Job Types: "compile", "errorparse", "rest", "coderepair", "show", "test"

Lexical Tokens:

1. KEYWORD - any of the words in the keywords list OR the job types list
2. IDENTIFIER - names for the jobs
3. OPERATOR - "=" or "->"

(a) = operator is used to assign values to "label" & "shape"



(b) -> operator is used to indicate the job before the operator must finish before the job after the operator can execute

4. BRACKET - "[" or "]"

5. BLOCK - "{" or "}"

6. TERMINATOR - ";"

(a) every line except for BLOCK\_TITLES and BLOCK\_BOUNDS must be concluded with a TERMINATOR

Valid Syntax:

1. BLOCK\_TITLE : {KEYWORD, IDENTIFIER}

2. DEPENDENT\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}

3. UI\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, IDENTIFIER, BRACKET, TERMINATOR}

4. END\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}

(a) required to indicate process end point

5. EXECUTIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, TERMINATOR}

6. CONDITIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR }

(a) dependency labels can only be assigned to true or false

- success is "true"

- fail is "false"

(b) any job that has a conditional dependency is required to have a both a job to execute if true and if false

7. BLOCK\_BOUND : {BLOCK}

8. START\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}

(a) required to indicate the first job in the process

In this example:

Job A is created with input "demo"

if Job A is successful, the process ends and no other jobs are created or executed.

If job A fails, an errorparse job, job B is created.

After Job B finishes, a show job, job C is created.

if Job c fails, job A runs again

if job C is successful, the process ends and no other jobs are created

digraph exampleflow

{

subgraph clusterExample

{

start[shape=none];

a[label=demo];

b[label=errorparse];

c[label=show]

end[shape=point];

```

start->a
a->b[label=false];
a->end[label=true];
c->end[label=true];
c->a[label=false];
b->c;
}
}

```

This example follows the following syntactical format:

```

<BLOCK_TITLE>
<BLOCK_BOUND>
<BLOCK_TITLE>
<BLOCK_BOUND>
<START_POINT>
<UI_JOB>
<DEPENDENT_JOB>
<DEPENDENT_JOB>
<END_POINT>
<EXECUTIONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<EXECUTIONAL_DEPENDENCY>

```

-----

There should be an equal amount of true and false dependencies. Success dependencies should be labeled as "true", and failure dependencies should be labeled as false

any job that has a conditional dependency is required to have a both a job to execute if true and if false

Your task is to generate the Flowscript script for the following process flow:

Create job A with input "demoerror"

If job A is successful, the process ends and no other jobs should be created or executed

If A fails , create an errorparse job, job B

After job B finishes, create job C with input "restjob"

Affter job C finishes, create a coderepair job, job D

If job D is successful, job A executes again

if Job D fails, the process ends and no other jobs are created

There should be 4 jobs total, one start point, one end point, 4 conditional dependencies & 3 executional dependencies

Do not include anything other than the Flowscript script in your response

**Response:** ```flowscript

digraph exampleflow

{

subgraph clusterExample

```

{
start[shape=none];
a[label=demoerror];
b[label=errorparse];
c[label=restjob];
d[label=coderepair];
end[shape=point];
start->a;
a->b[label=false];
a->end[label=true];
c->end[label=true];
d->c[label=false];
}
}
...

```

**Reasoning:** I added the cycle in the Flowscript example I provided so the LLM has the information to handle a cycle when generating a script. I also added the line about an equal number of true and false executional dependencies to make sure the LLM produced both possible flows of execution for conditional dependencies. Additionally, I added the 2nd to last line to inform the LLM of how many of each type of Flowscript statement should be included in the generated script.

**Feedback:** The conditional dependencies are now balanced, but they are not assigned to the correct job dependencies and there is a hole in the execution as the LLM did not include the dependency between job B and job C.

## Prompt 12

**Prompt:** Flowscript is a simple custom DSL for building process flows. FFlowscript specifics are provided below:

Keywords: "digraph", "subgraph", "label", "shape", "true", "false", "point", "none"

Job Types: "compile", "errorparse", "rest", "coderepair", "show", "test"

Lexical Tokens:

1. KEYWORD - any of the words in the keywords list OR the job types list

2. IDENTIFIER - names for the jobs

3. OPERATOR - "=" or "->"

(a) = operator is used to assign values to "label" & "shape"

(b) -> operator is used to indicate the job before the operator must finish before the job after the operator can execute

4. BRACKET - "[" or "]"

5. BLOCK - "{" or "}"

6. TERMINATOR - ";"

(a) every line except for BLOCK\_TITLES and BLOCK\_BOUNDS must be concluded with a TERMINATOR

Valid Syntax:

1. BLOCK\_TITLE : {KEYWORD, IDENTIFIER}

2. DEPENDENT\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}

3. UI\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, IDENTIFIER, BRACKET, TERMINATOR}

4. END\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}

(a) required to indicate process end point

5. EXECUTIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, TERMINATOR}

6. CONDITIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR }

(a) dependency labels can only be assigned to true or false

- success is "true"

- fail is "false"

(b) any job that has a conditional dependency is required to have a both a job to execute if true and if false

7. BLOCK\_BOUND : {BLOCK}

8. START\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}

(a) required to indicate the first job in the process

In this example:

Job A is created with input "demo"

if Job A is successful, the process ends and no other jobs are created or executed.

If job A fails, an errorparse job, job B is created.

After Job B finishes, a show job, job C is created.

if Job c fails, job A runs again

if job C is successful, the process ends and no other jobs are created

digraph exampleflow

```
{
subgraph clusterExample
{
start[shape=none];
a[label=demo];
b[label=errorparse];
c[label=show]
end[shape=point];
start->a
a->b[label=false];
a->end[label=true];
b->c;
c->end[label=true];
c->a[label=false];
}
}
```

This example follows the following syntactical format:

```

<BLOCK_TITLE>
<BLOCK_BOUND>
<BLOCK_TITLE>
<BLOCK_BOUND>
<START_POINT>
<UI_JOB>
<DEPENDENT_JOB>
<DEPENDENT_JOB>
<END_POINT>
<EXECUTIONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<EXECUTIONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>

```

-----

There should be an equal amount of true and false dependencies. Success dependencies should be labeled as "true", and failure dependencies should be labeled as false

any job that has a conditional dependency is required to have a both a job to execute if true and if false

Your task is to generate the Flowscript script for the following process flow:

Create job A with input "demoerror"

If job A is successful, the process ends and no other jobs should be created or executed

If A fails , create an errorparse job, job B

After job B finishes, create job C with input "restjob"

After job C finishes, create a coderepair job, job D

If job D is successful, job A executes again

if Job D fails, the process ends and no other jobs are created

**There should be 4 jobs total, one start point, one end point, 4 conditional dependencies & 3 executional dependencies**

Do not include anything other than the Flowscript script in your response

**Response:** ``

digraph exampleflow

```
{
```

```
subgraph clusterExample
```

```
{
```

```
start[shape=none];
```

```
a[label=demoerror];
```

```
b[label=errorparse];
```

```
c[label=restjob];
```

```
d[label=coderepair];
```

```
end[shape=point];
```

```
start->a;
```

```
a->b[label=false];
```

```
a->end[label=true];
```

```

b->c;
c->d;
d->end[label=true];
d->a[label=false];
}
}
...

```

**Reasoning:** I changed the order of the example Flowscript to have the dependencies be in an executive order (moved b->c from last line to underneath the dependencies for job A) to see if having that order would help the LLM understand the process needs to have a closed flow – any job in a process needs to be dependent on the execution of the job preceding it. Additionally, I removed the line that informs the LLM how many of each statement there are, because the inclusion of that line did not do anything to optimize the LLM response.

**Feedback:** This is almost the exact script I am requesting, but the conditional dependency labels for job D are inverted. I want job A to execute again if job D is successful and the process to end otherwise.

## Prompt 13

Prompt: Flowscript is a simple custom DSL for building process flows. FLOWscript specifics are provided below:

Keywords: "digraph", "subgraph", "label", "shape", "true", "false", "point", "none"

Job Types: "compile", "errorparse", "rest", "coderepair", "show", "test"

Lexical Tokens:

1. KEYWORD - any of the words in the keywords list OR the job types list
2. IDENTIFIER - names for the jobs
3. OPERATOR - "=" or "->"
  - (a) = operator is used to assign values to "label" & "shape"
  - (b) -> operator is used to indicate the job before the operator must finish before the job after the operator can execute
4. BRACKET - "[" or "]"
5. BLOCK - "{" or "}"
6. TERMINATOR - ";"
  - (a) every line except for BLOCK\_TITLES and BLOCK\_BOUNDS must be concluded with a TERMINATOR

Valid Syntax:

1. BLOCK\_TITLE : {KEYWORD, IDENTIFIER}
2. DEPENDENT\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
3. UI\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, IDENTIFIER, BRACKET, TERMINATOR}
4. END\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}
  - (a) required to indicate process end point
5. EXECUTIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, TERMINATOR}

6. CONDITIONAL\_DEPENDENCY : { IDENTIFIER, OPERATOR, IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR }

(a) dependency labels can only be assigned to true or false

- success is "true"

- fail is "false"

(b) any job that has a conditional dependency is required to have a both a job to execute if true and if false

7. BLOCK\_BOUND : { BLOCK }

8. START\_POINT : { IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR }

(a) required to indicate the first job in the process

In this example:

Job A is created with input "demo"

if Job A is successful, the process ends and no other jobs are created or executed.

If job A fails, an errorparse job, job B is created.

After Job B finishes, a show job, job C is created.

if Job c is successful, job A runs again

if job C fails, the process ends and no other jobs are created

digraph exampleflow

```
{
subgraph clusterExample
{
start[shape=none];
a[label=demo];
b[label=errorparse];
c[label=show]
end[shape=point];
start->a
a->b[label=false];
a->end[label=true];
b->c;
c->end[label=false];
c->a[label=true];
}
}
```

This example follows the following syntactical format:

<BLOCK\_TITLE>

<BLOCK\_BOUND>

<BLOCK\_TITLE>

<BLOCK\_BOUND>

<START\_POINT>

<UI\_JOB>

<DEPENDENT\_JOB>

<DEPENDENT\_JOB>

<END\_POINT>  
<EXECUTIONAL\_DEPENDENCY>  
<CONDIITONAL\_DEPENDENCY>  
<CONDIITONAL\_DEPENDENCY>  
<EXECUTIONAL\_DEPENDENCY>  
<CONDIITONAL\_DEPENDENCY>  
<CONDIITONAL\_DEPENDENCY>

-----

There should be an equal amount of true and false dependencies. Success dependencies should be labeled as "true", and failure dependencies should be labeled as false

any job that has a conditional dependency is required to have a both a job to execute if true and if false

Your task is to generate the Flowscript script for the following process flow:

Create job A with input "demoerror"

If job A is successful, the process ends and no other jobs should be created or executed

If A fails , create an errorparse job, job B

After job B finishes, create job C with input "restjob"

Affter job C finishes, create a coderepair job, job D

If job D is successful, job A executes again

if Job D fails, the process ends and no other jobs are created

Do not include anything other than the Flowscript script in your response

**Response:** digraph exampleflow

```
{
subgraph clusterExample
{
start[shape=none];
a[label=demoerror];
b[label=errorparse];
c[label=restjob];
d[label=coderepair];
end[shape=point];
start->a;
a->b[label=false];
a->end[label=true];
b->c;
c->d;
d->end[label=false];
d->a[label=true];
}
}
```

**Reasoning:** I switched the paths of execution for job C in the example Flowscript to give the LLM an example of a job failing leading to process completion and a job succeeding and leading to process completion. I think the LLM was under the influence that failed jobs lead to process completion.

**Feedback:** This is now the correct Flowscript code for the process I was describing.



# Final LLM Prompts

## Flowscript Generation Prompt

### Final Prompt

*Prompt: Flowscript is a simple custom DSL for building process flows. FLOWScript specifics are provided below:*

*Keywords: "digraph", "subgraph", "label", "shape", "true", "false", "point", "none"*

*Job Types: "compile", "errorparse", "rest", "coderepair", "show", "test"*

*Lexical Tokens:*

- 1. KEYWORD - any of the words in the keywords list OR the job types list*
- 2. IDENTIFIER - names for the jobs*
- 3. OPERATOR - "=" or "->"*
  - (a) = operator is used to assign values to "label" & "shape"*
  - (b) -> operator is used to indicate the job before the operator must finish before the job after the operator can execute*
- 4. BRACKET - "[" or "]"*
- 5. BLOCK - "{" or "}"*
- 6. TERMINATOR - ";"*
  - (a) every line except for BLOCK\_TITLES and BLOCK\_BOUNDS must be concluded with a TERMINATOR*

*Valid Syntax:*

- 1. BLOCK\_TITLE : {KEYWORD, IDENTIFIER}*
- 2. DEPENDENT\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}*
- 3. UI\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, IDENTIFIER, BRACKET, TERMINATOR}*
- 4. END\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}*
  - (a) required to indicate process end point*
- 5. EXECUTIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, TERMINATOR}*
- 6. CONDITIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR }*
  - (a) dependency labels can only be assigned to true or false*
    - success is "true"*
    - fail is "false"*
  - (b) any job that has a conditional dependency is required to have a both a job to execute if true and if false*
- 7. BLOCK\_BOUND : {BLOCK}*

8. *START\_POINT* : {*IDENTIFIER*, *BRACKET*, *KEYWORD*, *OPERATOR*, *KEYWORD*, *BRACKET*, *TERMINATOR*}

(a) required to indicate the first job in the process

In this example:

Job A is created with input "demo"

if Job A is successful, the process ends and no other jobs are created or executed.

If job A fails, an errorparse job, job B is created.

After Job B finishes, a show job, job C is created.

if Job c is successful, job A runs again

if job C fails, the process ends and no other jobs are created

digraph exampleflow

```
{
subgraph clusterExample
{
start[shape=none];
a[label=demo];
b[label=errorparse];
c[label=show]
end[shape=point];
start->a
a->b[label=false];
a->end[label=true];
b->c;
c->end[label=false];
c->a[label=true];
}
}
```

This example follows the following syntactical format:

```
<BLOCK_TITLE>
<BLOCK_BOUND>
<BLOCK_TITLE>
<BLOCK_BOUND>
<START_POINT>
<UI_JOB>
<DEPENDENT_JOB>
<DEPENDENT_JOB>
<END_POINT>
<EXECUTIONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<EXECUTIONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
<CONDIITONAL_DEPENDENCY>
```

-----

*There should be an equal amount of true and false dependencies. Success dependencies should be labeled as "true", and failure dependencies should be labeled as false*

*any job that has a conditional dependency is required to have a both a job to execute if true and if false*

*Your task is to generate the Flowscript script for the following process flow:*

*{Custom process description in Data/LLMFlowscriptPrompt.txt}*

*Do not include anything other than the Flowscript script in your response*

## **Prompt Methodology**

*Flowscript is a simple custom DSL for building process flows. FLoowscript specifics are provided below:*

*Keywords: "digraph", "subgraph", "label", "shape", "true", "false", "point", "none"*

*Job Types: "compile", "errorparse", "rest", "coderepair", "show", "test"*

*Lexical Tokens:*

*1. KEYWORD - any of the words in the keywords list OR the job types list*

*2. IDENTIFIER - names for the jobs*

*3. OPERATOR - "=" or "->"*

*(a) = operator is used to assign values to "label" & "shape"*

*(b) -> operator is used to indicate the job before the operator must finish before the job after the operator can execute*

*4. BRACKET - "[" or "]"*

*5. BLOCK - "{" or "}"*

*6. TERMINATOR - ";"*

*(a) every line except for BLOCK\_TITLES and BLOCK\_BOUNDS must be concluded with a TERMINATOR*

*Valid Syntax:*

*1. BLOCK\_TITLE : {KEYWORD, IDENTIFIER}*

*2. DEPENDENT\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}*

*3. UI\_JOB : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, IDENTIFIER, BRACKET, TERMINATOR}*

*4. END\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}*

*(a) required to indicate process end point*

*5. EXECUTIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, TERMINATOR}*

*6. CONDITIONAL\_DEPENDENCY : {IDENTIFIER, OPERATOR, IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR }*

*(a) dependency labels can only be assigned to true or false*

*- success is "true"*

*- fail is "false"*

*(b) any job that has a conditional dependency is required to have a both a job to execute if true and if false*

*7. BLOCK\_BOUND : {BLOCK}*

*8. START\_POINT : {IDENTIFIER, BRACKET, KEYWORD, OPERATOR, KEYWORD, BRACKET, TERMINATOR}*

*(a) required to indicate the first job in the process*

I included the above portion of the LLM prompt to explain the specifics of the Flowscript language I designed. By defining each lexical token, I was able to inform the LLM of all the possible values for the different tokens in an organized way. This also made it easier to specify the proper syntax later in the prompt. Using the predefined lexical tokens, I was able to layout all the possible combinations of tokens to create valid Flowscript statements. This format was also helpful because it kept the Flowscript logic I was trying to explain more organized and I could describe the statement-specific rules, for example I could inform the LLM that the 2nd keyword in a conditional dependency statement must either be true or false.

*In this example:*

*Job A is created with input "demo"*

*if Job A is successful, the process ends and no other jobs are created or executed.*

*If job A fails, an errorparse job, job B is created.*

*After Job B finishes, a show job, job C is created.*

*if Job c is successful, job A runs again*

*if job C fails, the process ends and no other jobs are created*

In this portion of the prompt, I describe the logic of the Flowscript code I am using as an example. Here, I talk about the jobs in terms of their identifiers, so the LLM does not confuse the job identifier and the job label value. For example, if I have a process that includes two different compile jobs, if I referred to the jobs by their identifiers, the LLM would not be able to identify which of the two compile jobs I was referring to. The format and language of this process description also match that of the actual instructional prompt being passed into the LLM (located in Data/LLMFlowscriptPrompt.txt).

*digraph exampleflow*

```
{
subgraph clusterExample
{
start[shape=none];
a[label=demo];
b[label=errorparse];
c[label=show]
end[shape=point];
start->a
a->b[label=false];
a->end[label=true];
b->c;
c->end[label=false];
c->a[label=true];
}
}
```

This section of the prompt is where I provide the actual Flowscript code example. In the example I included both conditional and executional dependencies, a job execution from the previous job failing and a job execution from the previous job succeeding, and a cycle. I chose to include all these different Flowscript features, so the LLM would have the information necessary to include any of these features in the script it is generating if the input requests it.

*This example follows the following syntactical format:*

```
<BLOCK_TITLE>
<BLOCK_BOUND>
<BLOCK_TITLE>
<BLOCK_BOUND>
<START_POINT>
<UI_JOB>
<DEPENDENT_JOB>
<DEPENDENT_JOB>
<END_POINT>
<EXECUTIONAL_DEPENDENCY>
<CONDITIONAL_DEPENDENCY>
<CONDITIONAL_DEPENDENCY>
<EXECUTIONAL_DEPENDENCY>
<CONDITIONAL_DEPENDENCY>
<CONDITIONAL_DEPENDENCY>
```

Here, I provide the LLM with the syntactical format of the Flowscript example I provided, so the LLM can map the Flowscript to the statement type, as well as so the LLM has an example of each of the different types of Flowscript statements.

## C++ Code Repair Prompt

### Final Prompt

*This is the format of an error object:*

```
{
  {'colNum': #},
  {'errorMessage': ""},
  {'file': ""},
  {'lineNum': #},
  {'nextLine': ""},
  {'previousLine': ""},
  {'resDescr': ""},
  {'src': ""},
  {'srcResolved': ""}
}
```

*Your response should be in the following valid json format:*

```
{
  "file.cpp": [
    {"colNum": #,
      "errorMessage": "",
      "file": "",
      "lineNum": #,
      "nextLine": "",
      "previousLine": "",
      "resDescr": "",
      "src": "",
      "srcResolved": ""}
    , {"colNum": #,
      "errorMessage": "",
      "file": "",
      "lineNum": #,
      "nextLine": "",
      "previousLine": "",
      "resDescr": "",
      "src": "",
      "srcResolved": ""}
  ]
}
```

*For each error object given, provide the resolved C++ code in the 'srcResolved' member, and provide a description of the fix in the 'resDescr' member.*

*Leave the incorrect code in the 'src' member.*

*'srcResolved' should only contain valid c++ code. Replace "file.cpp" with the 'file' memeber in the error object.*

*Do not prompt your response.*

*{errors}*

## **Prompt Methodology**

*This is the format of an error object:*

```
{
  {'colNum': #},
  {'errorMessage': ""},
  {'file': ""},
  {'lineNum': #},
  {'nextLine': ""},
  {'previousLine': ""},
  {'resDescr': ""},
  {'src': ""},
  {'srcResolved': ""}
```

```
}
```

In this portion of the LLM prompt, I inform the LLM of the format of the error objects that will be passed in.

*Your response should be in the following valid json format:*

```
{  
  "file.cpp": [  
    {"colNum": #,  
      "errorMessage": "",  
      "file": "",  
      "lineNum": #,  
      "nextLine": "",  
      "previousLine": "",  
      "resDescr": "",  
      "src": "",  
      "srcResolved": ""}  
    ,{"colNum": #,  
      "errorMessage": "",  
      "file": "",  
      "lineNum": #,  
      "nextLine": "",  
      "previousLine": "",  
      "resDescr": "",  
      "src": "",  
      "srcResolved": ""}  
  ]  
}
```

Here I give the LLM the format I want its response to be returned in to ensure it returns proper json and there will not be any json read errors when the coderepair jobs executes.

*For each error object given, provide the resolved C++ code in the 'srcResolved' member, and provide a description of the fix in the 'resDescr' member.*

In this section of the prompt I explicitly instruct the LLM what I want it to produce (the resolved code and a description of the resolution) and where exactly to put the information produced.

*Leave the incorrect code in the 'src' member.*

I include this line for error handling. In the coderepair execute function, if the srcResolved member of the error object is empty, the repair job will then grab the previous src and use that. If the src member holds the old code, when the repair job completes and a compile job is fired again, the same error will be produced and the LLM will again be prompted to solve the error.

*'srcResolved' should only contain valid c++ code.*

I included this line so the LLM knows not to include any information on the resolution in the srcResolved error object member.

*Replace "file.cpp" with the 'file' member in the error object.*

This line works to instruct the LLM to nest the error objects under the path to the file these errors occurred in. The LLM succeeds at doing this most of the time, but in the case where it does not, I have also programmed an error check in the coderepair job that will check to see if the file path in the error object is the same as the file path the object is nested under.

*Do not prompt your response.*

I included this final line of the prompt so the LLM only returns the updated json, making parsing and code repair much easier.