# FMC - Final Report

*By Kenny Zhang (ID: 48618419)*

## Prompts Introduction

For this report, I used GPT4All with the Mistral OpenOrca model, which was trained by NomicAI and fine tuned on the OpenOrca dataset. I developed both prompts so that they would use the amount of memory needed for the task as GPT4All would frequently crash when it used too much because CLion also uses a lot of memory as it sends the request. This would result in 100% usage of my computer's memory and GPT4All accessing a memory location it wasn't supposed to or that wasn't free which caused crashes. For both prompts, I used one-shot training as zero-shot training would return wrong results and few-shot training only seemed to confuse the model, as well as increase response times and memory usage, which I want to avoid.

## Flowscript Prompt

*Using flowscript, a language based on DOT, create a graph.*

*Flowscript Example:*

```
digraph conditionalTestSuccess{
  A;
  B-> A;
  C-> A;

  D-> B;
  B-> E;
  D-> C;
  E-> F;
}
```

*Available Jobs:*
*compileJobErrorTest.json*
*compileJobSuccessTest.json*
*customJob.json*

*renderJob.json*
*renderJobA.json*
*renderJobBfail.json*
*renderJobBsuccess.json*
*renderJobC.json*
*renderJobD.json*
*renderJobE.json*
*renderJobF.json*

*Make a graph that executes the job "[JOB NAME]". To do this, make a digraph with only one node: [JOB NAME].json. Your response should have the code and nothing else. Do not put the code into a special format.*

### Flowscript Prompt Explanation

This prompt was specifically tailored for making a pipeline for compiling. Since my compile job does all of the necessary steps necessary for compilation (compilation, error parsing, outputting to a json file, etc.) I didn't really need to have a complex flowscript graph for this program specifically. Because of this, the instructions are quite simple. The flowscript example also helps clarify syntax for the model.

However, for extendability, I included a list of the available jobs if a more complicated pipeline is ever needed. On the specific wording of the prompt, I mentioned that flowscript is based on the DOT language. I did not think this would make an appreciable difference but after testing, this did improve results. I also specified that the response should have code and only code so that I could just take the whole response and put it directly into the flowscript graph file without any processing. The part about not wanting special formatting was due to the model putting the code into those embedded windows/boxes which, again, would require processing if I were to inject this directly into the graph file.

## Code Fix Request Prompt

*In the previous code there are the following errors:*
*[BROKEN CODE]*

*Error on line [LINE NUMBER], column [COLUMN NUMBER]:*
*[ERROR DESCRIPTION]*

*Fix the code. Your response should include nothing but raw code. Do not put the code into a special format.*

## Code Fix Request Prompt Explanation

For this prompt, I found that putting the entirety of the error json's text into the prompt made it quite lengthy and would cause crashes on my computer. This error json would include the column number, line number, description, error type, and a code snippet. Some of this is extraneous like the error type and the code snippet. With the code snippet taking so much space, removing it helped make the feedback more clear and did not remove any information as the LLM already has access to the code. Putting everything so that it only took up two lines was also helpful, for me at least, as it made it much easier to look through. I'm pretty sure that doing this way helps the LLM as well, but I am not completely sure. Either way, this format was fairly successful in correcting code.

# Jobs

---

## CompileJob

This job is created by inputting a json object or an ifstream to a json file into the `CreateAndQueue()` function. The file type is determined by what is input into jobType in the json file, which in this case is "compile". The job system will read the job type and create the appropriate job, while also passing along the input parameters.

The compile job is then executed when the job preceding it finished. This doesn't happen in this program, as the compile job is the only job in the pipeline, but it would if there were other jobs.

If the compilation has errors, the job will output a json file containing all the errors encountered sorted by file. If not, the json file will just say “Files compiled successfully. No errors found.” The job will also output a text file with all of the code from the files that were compiled so that it can be passed to the chat job. For this specific program, the compileJob just compiles the files that were chosen as targets in the makefile command.

Prompt formats are shown at the beginning of this report.

## CompileJob Inputs:

- jobName - The name you want the job to have when checking job statuses.
- jobType - The type of job
- inputs - an array of values that are used as parameters for the job
    - makefileContents - the makefile command you want to execute
    - outputFileName - the name of the file your output will be named

## ChatJob

This job is created by inputting a json object or an ifstream to a json file into the `CreateAndQueue()` function. The file type is determined by what is input into jobType in the json file, which in this case is “chat”. The job system will read the job type and create the appropriate job, while also passing along the input parameters.

The chat job is then executed directly as it is not needed for the compiling pipeline. This lets me run it separately from the compile job and verify the outputs of the compile job, as well as show it to the camera in the video documentation. This also lets me run my pause function, which lets me interact with the job system (check status, stop system, etc.) in the middle of the program.

When executed, the chat job sends a POST request to the endpoint and then has a callback that gets its response. The response is formatted like this example:

```
{
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "logprobs": null,
      "text": "Who is Michael Jordan?\nMichael Jordan is a former
professional basketball player who played for the Chicago Bulls in the
NBA. He was born on December 30, 1963, and retired from playing
basketball in 1998."
    }
  ],
  "created": 1684260896,
  "id": "foobarbaz",
  "model": "gpt4all-j-v1.3-groovy",
  "object": "text_completion",
  "usage": {
    "completion_tokens": 35,
    "prompt_tokens": 39,
    "total_tokens": 74
  }
}
```

This response has the prompt and response in the same string, which isn't great when I need to transfer its response into another file. That's why it is processed to only include the answer (by making a substring of the text that starts at the index that is equal to the prompt's length) which is then put into a json file to be read later.

Prompt formats for these jobs are shown at the beginning of this report.

### ChatJob Inputs:

- jobName - The name you want the job to have when checking job statuses.
- jobType - The type of job
- inputs - an array of values that are used as parameters for the job
  - endpoint - the url that the POST request will be sent to
  - model - the model you want to process your prompt
  - prompt - the prompts you want the model to use

## Video Demo

https://youtu.be/1MYliGPo3ts?si=0j0Ag1z2n4pOvufY

# Walkthrough of Video Demo

First, the chat job json for the FlowScript code is opened and its contents are turned into a json. Then, a function writes the prompt (shown at the beginning of this report) and inserts that value into the json. The raw chat job json looks like this:

```
{
   "input": {
       "endpoint": "http://localhost:4891/v1/chat/completions",
       "model": "mistral-7b-openorca.Q4_0.gguf",
       "prompt": "\nUsing flowscript, a language based on DOT, build a
compiling pipeline that will return all errors from the compilation of a
C++ code project. This can be done by building a digraph and using the
names of the job as nodes. An example and a list of jobs is shown
below.\n\nFlowscript Example:\ndigraph conditionalTestSuccess{\n   A;\n
B-> A;\n   C-> A;\n\n   D-> B;\n   B-> E[condition=\"success\"];\n   D->
C;\n   E-> F;\n}\n\nAvailable
Jobs:\ncompileJobErrorTest.json\ncompileJobSuccessTest.json\ncustomJob.js
on\nrenderJob.json\nrenderJobA.json\nrenderJobBfail.json\nrenderJobBsucce
ss.json\nrenderJobC.json\nrenderJobD.json\nrenderJobE.json\nrenderJobF.js
on\n\nFor the compiling pipeline I want you to return, make a pipeline
that executes the job called compileJobErrorTest.json. To do this, make a
digraph with only one node: compileJobErrorTest.json. Only include the
raw text for the pipeline and nothing else. Also, do not include it in a
code box.\n"
   },
   "jobName": "chatTaskError",
   "jobType": "chat"
}
```

Then the chat job is queued. It outputs a json document that has the model's response (the code for the FlowScript pipeline) and the prompt I gave it:

```
{

   "prompt": "\nUsing flowscript, a language based on DOT, build a
compiling pipeline that will return all errors from the compilation of a
C++ code project. This can be done by building a digraph and using the
names of the job as nodes. An example and a list of jobs is shown
below.\n\nFlowscript Example:\ndigraph conditionalTestSuccess{\n   A;\n
B-> A;\n   C-> A;\n\n   D-> B;\n   B-> E[condition=\"success\"];\n   D->
C;\n   E-> F;\n}\n\nAvailable
Jobs:\ncompileJobErrorTest.json\ncompileJobSuccessTest.json\ncustomJob.js
on\nrenderJob.json\nrenderJobA.json\nrenderJobBfail.json\nrenderJobBsucce
ss.json\nrenderJobC.json\nrenderJobD.json\nrenderJobE.json\nrenderJobF.js
on\n\nFor the compiling pipeline I want you to return, make a pipeline
that executes the job called compileJobErrorTest.json. To do this, make a
digraph with only one node: compileJobErrorTest.json. Only include the
raw text for the pipeline and nothing else. Also, do not include it in a
code box.\n",
   "response": "digraph compileJobErrorTest{\n\tcompileJobErrorTest;\n}"
}
```

The json document's response is put into a graph file:

```
digraph compileJobErrorTest{
   compileJobErrorTest;
}
```

Which is then read and given to the flowscript interpreter, which reads and runs the FlowScript graph.

Then the agent starts fixing the code. The code I am using for this test is this:

```
#include <vector>
#include <iostream>

using namespace std;

void test3() {
  int total = 0;
  vector<int> data;
  Int error = 5;

  for (int i = 0; i < 1000; i++) {
     data.push_back(i);
  }

  std::vector<int>::iterator it = data.begin();

  cout < "text" << endl;

  for (; it != data.end(); ++it) {
     total += *it;
  }

  it = data.begin();

  for (; it != data.end(); ++it) {
     total += *it;
  }
}
```

There are two errors in this file. The first is “int” being capitalized as “Int” at the beginning of test3(). The second is the cout statement writing the “<<” operator as “<”.

The agent fixes the code through a loop. The general loop goes like this: compile target code -> check if code compiled successfully, break the loop if it has -> write new prompt with the outputs of the compileJob (error json and code text) and insert it into the chatJob json -> run chatJob and send message to LLM -> output of LLM is inserted into the target code file -> return to beginning of the loop.

So let’s step through this loop. CompileJob is run and outputs this error file:

```
{
   "./TestCodeError/test2.cpp": [
       {
           "code_snippet": [
               "\tint total = 0;",
               "\tvector<int> data;",
               "\tInt error = 5;",
               "",
               "\tfor (int i = 0; i < 1000; i++) {"
           ],
           "column_number": "9",
           "description": "'Int' was not declared in this scope; did you mean 'int'?",
           "error_type": "error",
           "line_number": "9"
       },
       {
           "code_snippet": [
               "\tstd::vector<int>::iterator it = data.begin();",
               "",
               "\tcout < \"text\" << endl;",
               "",
               "\tfor (; it != data.end(); ++it) {"
           ],
           "column_number": "23",
           "description": "invalid operands of types 'const char [5]' and '<unresolved overloaded function type>' to binary 'operator<<'",
           "error_type": "error",
           "line_number": "17"
       }
   ]
}
```

as well as this code from the target code file:

```
#include <vector>
#include <iostream>

using namespace std;

void test3() {
  int total = 0;
  vector<int> data;
  Int error = 5;

  for (int i = 0; i < 1000; i++) {
     data.push_back(i);
  }

  std::vector<int>::iterator it = data.begin();

  cout < "text" << endl;

  for (; it != data.end(); ++it) {
     total += *it;
  }

  it = data.begin();

  for (; it != data.end(); ++it) {
     total += *it;
  }
}
```

Since the error json contains errors, we write a new prompt:

```
Relative Filepath: ./TestCodeError/test2.cpp
#include <vector>
#include <iostream>

using namespace std;

void test3() {
	int total = 0;
	vector<int> data;
	Int error = 5;

	for (int i = 0; i < 1000; i++) {
		data.push_back(i);
	}

	std::vector<int>::iterator it = data.begin();

	cout < "text" << endl;

	for (; it != data.end(); ++it) {
		total += *it;
	}

	it = data.begin();

	for (; it != data.end(); ++it) {
		total += *it;
	}
}


In the previous code there are the following errors:
Error on line "9", column "9":
	"'Int' was not declared in this scope; did you mean 'int'?"
Error on line "17", column "23":
	"invalid operands of types 'const char [5]' and '<unresolved
overloaded function type>' to binary 'operator<<'"


Fix the code. Your response should include nothing but raw code. Do not
put the code into a special format.
```

This prompt is put into the chatJob json and then sent to the LLM, following the same format as outlined before. We get this response:

```
#include <vector>
#include <iostream>


using namespace std;


void test3() {
	int total = 0;
	vector<int> data;
```

```
        for (int i = 0; i < 1000; i++) {
                data.push_back(i);
        }

        std::vector<int>::iterator it = data.begin();
```

This is the whole response by the way, which is very clearly wrong. The response is put into the target code file. The loop restarts. The target code file is compiled. The errors and code are received and put into a prompt. The prompt is then sent to the LLM using a chatJob. The following errors are received as a json. Since it has the same format as the other error jsons I will just show them in a condensed format. There are the following errors :

```
Error on line "14", column "50":
        "expected '}' at end of input"

Error on line "6", column "14":
        "to match this '{'
```

The errors and code are put into a prompt:

```
{
   "prompt": "Relative Filepath: ./TestCodeError/test2.cpp\n#include
<vector>\n#include <iostream>\n\nusing namespace std;\n\nvoid test3() {\n
int total = 0;\n    vector<int> data;\n    \n    for (int i = 0; i <
1000; i++) {\n        data.push_back(i);\n    }\n\n
std::vector<int>::iterator it = data.begin();\n\nIn the previous code
there are the following errors:\nError on line \"14\", column \"50\":\n\t
\"expected '}' at end of input\"\nError on line \"6\", column \"14\":\n\t
\"to match this '{'\"\n\n Fix the code. Your response should include
nothing but raw code. Do not put the code into a special format.",
   "response": "#include <vector>\n#include <iostream>\n\nusing namespace
std;\nvoid test3() {\n    int total = 0, sum = 0;\n    vector<int>
data;\n    \n    for (int i = 0; i < 1000; i++) {\n
data.push_back(i);\n    }\n    \n    std::vector<int>::iterator it =
data.begin();\n}"
}
```

The prompt returns this code:

```
#include <vector>
#include <iostream>

using namespace std;

void test3() {
        int total = 0;
        vector<int> data;
        for (int i = 0; i < 1000; i++) {
                data.push_back(i);
        }

        std::vector<int>::iterator it = data.begin();
}
```

While this code is incorrect, it compiles successfully and the program ends.

# Program Overview

Now for a final overview and review of the agent's capabilities. The agent is able to use LLMs to create a compiling pipeline using FlowScript based on the prompt given. It can also take the outputs of compileJobs and translate them into a usable prompt to give to LLMs to obtain corrected versions of the target code. Through this, it is able to fix code (or at least make it compilable). It doesn't work 100% of the time but it does a pretty good job for the most part.

The processes of the program, and therefore the capabilities of the agent, are as follows: send FlowScript generation prompt to LLM using chatJob -> use FlowScript Interpreter to interpret -> use FlowScript to create compiling pipeline -> start loop -> compile target code -> check if code compiled successfully, break the loop if it has -> write new prompt with the outputs of the compileJob (error json and code text) and insert it into the chatJob json -> run chatJob and send message to LLM -> output of LLM is inserted into the target code file -> return to beginning of the loop.