

Final Lab Report

Video Demonstration (youtube link):

<https://youtu.be/Ef22pNMQgrY>

Max link 48511338

Summary of important milestone changes:

- USING GPT4ALL with enabled REST API server. Communicating on port 4891 here in LLMCall.py file:

```
- openai.api_base = "http://localhost:4891/v1" #TODO - pass local host 4891 to  
  LLMCall.py from customjob.cpp after passed to customJob from main? URL NOT IP SO  
  NO?  
-
```

- Run code with “make compile” and “make run” in the root directory
- Final LLM prompts in main.cpp:

- Prompt for flowscript:

```
- std::string writeFlowScriptPrompt = "Please write flowscript code based on the  
  context. Context: Here is flowscript for you to base your flowscript off of. The flowScript  
  must connect jobs in order where 'compilejob' is the first job, 'parsingjob' is the second
```

job, and 'outputjob' is the third job. Only those three jobs should be created. Question: Please write flowscript that mimics this flowscript: " + flowScript;

- Prompt for fixing code errors:

```
std::string fixErrorsPrompt = "Answer question based on the context. Context: Here is code with errors in a json file format. This json file format does not include all the errors so please refer to the full source files that are at the end of this provided prompt. Fix the errors in each of these .cpp files and format your output on the full source code files at the end of this input. In your response, output the entire .cpp files with all errors fixed. These fixed .cpp files should be returned as neatly formatted strings that can be written to .cpp files to be ran later. Add a description of what you did to fix the errors. Please output your fixes in a SINGLE json file like this format with a [ { 'File Path: ' field, followed by a 'Fixed .cpp: ' field, and finally ending with a 'Description: ' field } ]. Please make sure that every 'Fixed .cpp:' field starts with an #include <iostream> header'. Before you output the json file be sure to say 'Here are the fixed .cpp files:' Question: " + errorsWithSourceCode;
```

in main.cpp the code is in a while loop that breaks when the output.json file reads “No errors or warnings – compiled successfully”. The while loop loops 10 times to attempt to fix the code. This hard limit was put in place so that github actions could compile the code without looping forever, but you could theoretically take off this limit and loop forever and fix the code and break. My program fixes the code before it reaches this hard limit and breaks because the output.json file reads “No errors or warnings – compiled successfully”.

The github repo has the ./test-code/ directory full of errors currently, so that you can test the LLM’s ability to solve the errors in the ./test-code/ directory. The ./test-code-backup directory exists so that you can uncomment the “reloadErrors()” function call above the “return 0;” in main.cpp to re-write the errors to the ./test-code/ directory. This allows

you to easily solve the errors and retry so that you can test multiple times. The ./fixed-test-code/ directory exists so that you can paste fully functional code into the ./test-code directory and test the while loop breaking mechanic.

The rest of this report is a running journal of my LLM prompt development and other challenges that had to be solved, but I hope this summary section helps for a quicker understanding of concepts.

Overview of main.cpp:

```
#include <iostream>
#include "JobSystemAPI.h"
#include "customjob.h"

#include <fstream> //for reading in files
#include "../lib/json.hpp" //for json manipulation
#include <sstream> //for reading the output.json file in main to check if it contains "No errors found!"
using json = nlohmann::json;

//helper functions

std::string readFileContents(const std::string& filePath) {
    //call processFile function to read file contents
    std::string file_content;
    std::ifstream file(filePath);
    if (file.is_open()) {
```

```

        file_content.assign((std::istreambuf_iterator<char>(file)),
                            std::istreambuf_iterator<char>());

        file.close();
    } else {
        std::cerr << "Unable to open file: " << filePath << std::endl;
    }
    return file_content;
}

std::string processFile(const std::string& filePath) {
    std::string file_content = readFileContents(filePath);

    if (filePath.size() >= 5 && filePath.substr(filePath.size() - 5) == ".json") {
        // Parse the JSON data
        json json_data = json::parse(file_content);
        // Serialize JSON object to string
        return json_data.dump(4); // '4' for pretty printing
    } else if (filePath.size() >= 4 && (filePath.substr(filePath.size() - 4) == ".txt" ||
filePath.substr(filePath.size() - 4) == ".cpp" || filePath.substr(filePath.size() - 2) == ".h")) {
        // For .txt, .cpp, and .h files, just read and return the file content
        return file_content;
    } else {
        std::cerr << "Unsupported file type for file: " << filePath << std::endl;
        return "";
    }
}

std::string escapeShellArgument(const std::string& arg) {
    std::string escaped;

```

```

for (char c : arg) {
    // Add a backslash before special characters
    if (c == '"' || c == '\\' || c == '$') {
        escaped += '\\';
    }
    escaped += c;
}
return escaped;
}

//TODO - deprecated function, remove later?
void waitForJobToComplete(MyJobSystemAPI& jobSystem, const std::string& jobType){
    //wait for compilejob to finish
    int jobStatus = jobSystem.getJobStatus(jobType);
    int completed = 3;

    //busy while loop for compilejob
    while(jobStatus != completed){
        if(jobStatus == -1){
            std::cout << "invalid job, I cannot wait on a non-existant job. Exiting while-loop" <<
std::endl;
            break;
        }
        std::cout << "waiting for" << jobType << " to complete" << std::endl;
        jobStatus = jobSystem.getJobStatus(jobType);
    }

    std::cout << jobType + " finished!" << std::endl;
}

```

```

void flowScriptInterpreter(MyJobSystemAPI& jobSystem, const std::string&
jobType){//interpreting flowscript with this function

    jobSystem.createJob("compilejob"); //create compile job
    jobSystem.createJob("parsingjob"); //create parsing job
    jobSystem.createJob("outputjob"); //create output job

    //TODO - pass jobs vector into here by ref
    //interpret flowscript files
    jobSystem.createJob("interpreterjob"); //create interpreter job

    //jobSystem.initialize(); //initialize job system -- creates threads in isolated function to avoid
creating too many threads & running out of resources
    jobSystem.queueSpecJob("interpreterjob"); //queue interpreter job
}

void LLMCall(MyJobSystemAPI& jobSystem, const std::string& jobType, std::string&
promptToPassIn){ //instantiate LLM job thread

    //make custom job to call to LLM
    jobSystem.registerCustomJob(CustomJob::CreateCustomJob, "LLM"); //register custom job
    jobSystem.createCustomJob("LLM"); //create custom job
    //give customJob prompt and IP address
    jobSystem.getLLMJob()->SetPrompt(promptToPassIn); //returns LLM job type for
manipulation
    jobSystem.getLLMJob()->SetIP("Dummy-there-is-no-ip"); //sets the IP value for the LLM job
type. In this case localhost 4891 port #.

```

```

jobSystem.queueSpecJob("LLM"); //queue interpreter job

//TODO - is this going to malfunction? Does this wait work? --- says cannot wait on invalid job
because job finishes before getting to this wait function, so is this wait function irrelevant?

//waitForJobToComplete(jobSystem, "LLM"); //wait for LLM job to finish
}

void reloadErrors(){//this function repopulates the ./test-code directory with the backup error files
from the ./test-code=backups directory

// Array of file names that need to be reset

std::array<std::string, 3> file_names = {"test-set-one.cpp", "test-set-three.cpp", "test-set-
two.cpp"};

// Iterate through the file names and copy each from the backup directory to the test-code
directory
for (const std::string& file_name : file_names) {
    std::string command = "cp ./test-code-backups/" + file_name + " ./test-code/" + file_name;
    // Execute the copy command
    int result = system(command.c_str());
    // Check if the command was successful
    if (result != 0) {
        std::cerr << "Failed to copy " << file_name << std::endl;
    } else {
        std::cout << "Successfully reset errors for " << file_name << std::endl;
    }
}
}

int main(void){

```

```

//TODO - change from an int to a forever loop
int i = 0;
while(i != 10){//while loop that breaks when the output.json file reads "No errors or warnings -
compiled successfully"

    //call API class functions for testing
    MyJobSystemAPI jobSystem; //create job system object

    jobSystem.initialize(); //initialize job system -- creates threads in isolated function to avoid
creating too many threads & running out of resources

    //JSON file path for flowScript prompt
    std::string flowScriptPromptFilePath = "./Data/fsp1.txt";
    std::string flowScript = processFile(flowScriptPromptFilePath); //read the json file and store
the output in a string called code
    flowScript = escapeShellArgument(flowScript); //escape the code string to be used in the
command string below

    std::string writeFlowScriptPrompt = "Please write flowscript code based on the context.
Context: Here is flowscript for you to base your flowscript off of. The flowScript must connect
jobs in order where 'compilejob' is the first job, 'parsingjob' is the second job, and 'outputjob' is
the third job. Only those three jobs should be created. Question: Please write flowscript that
mimics this flowscript: " + flowScript;

    //TODO - pass a generic prompt to this thread from main so that you can run this function
multiple times and create multiple LLM jobs in the running of main. One job should write the
flowscript with zero/fewshot training, then the second call should fix the errors with a while
loop...

    LLMCall(jobSystem, "LLM", writeFlowScriptPrompt); //instantiate LLM job thread to write
flowScript

    //interpret flowscript that LLMCall writes on line 121 of main.cpp

```



```
flowScriptInterpreter(jobSystem, "interpreterjob"); //interpreting flowscript with this function

// JSON file path for output.json file from lab one that contains coding errors
std::string json_file_path = "./Data/output.json";

std::string code = processFile(json_file_path); //read the json file and store the output in a
string called code

code = escapeShellArgument(code); //escape the code string to be used in the command
string below

//read source code files from ./test-code directory
std::string sourceCodeMain = "./test-code/main.cpp";
std::string sourceCodeTestOneCpp = "./test-code/test-set-one.cpp";
std::string sourceCodeTestTwoCpp = "./test-code/test-set-two.cpp";
std::string sourceCodeTestThreeCpp = "./test-code/test-set-three.cpp";

//read in the .h files
std::string headerFileTestSetOne = "./test-code/test-set-one.h";
std::string headerFileTestSetTwo = "./test-code/test-set-two.h";
std::string headerFileTestSetThree = "./test-code/test-set-three.h";

//TODO - figure out why test-code-one.cpp does not include the headers when it is fed to
the LLM from the combinedSourceCode string below. Something with how it is read into the
combinedSourceCode from the processFile?

std::string combinedSourceCode = processFile(sourceCodeMain) +
processFile(sourceCodeTestOneCpp) + processFile(headerFileTestSetOne) +
processFile(sourceCodeTestTwoCpp) + processFile(headerFileTestSetTwo) +
processFile(sourceCodeTestThreeCpp) + processFile(headerFileTestSetThree); //read the json
file and store the output in a string called code

combinedSourceCode = escapeShellArgument(combinedSourceCode); //escape the code
string to be used in the command string below
```

```

std::cout << "\n\n\ncombinedSourceCode: \n\n\n\n\n" << combinedSourceCode <<
std::endl;

//std::string errorsWithSourceCode = code + "\n\n\nThose were the errors, now here is
the source code: \n\n\n\n" + combinedSourceCode;

std::string errorsWithSourceCode = code + combinedSourceCode;

std::cout << "\n\n\nerrorsWithSourceCode: \n\n\n\n\n" << errorsWithSourceCode <<
std::endl;

//load a json prompt format from a json file for the LLM to use to format the output.json file
//TODO - change from hard coded test format to suggestion format? make current a .bak

std::string fixErrorsPrompt = "Answer question based on the context. Context: Here is code
with errors in a json file format. This json file format does not include all the errors so please
refer to the full source files that are at the end of this provided prompt. Fix the errors in each of
these .cpp files and format your output on the full source code files at the end of this input. In
your response, output the entire .cpp files with all errors fixed. These fixed .cpp files should be
returned as neatly formatted strings that can be written to .cpp files to be ran later. Add a
description of what you did to fix the errors. Please output your fixes in a SINGLE json file like
this format with a [ { 'File Path: ' field, followed by a 'Fixed .cpp: ' field, and finally ending with a
'Description: ' field } ]. Please make sure that every 'Fixed .cpp:' field starts with an #include
<iostream> header'. Before you output the json file be sure to say 'Here are the fixed .cpp files:'
Question: " + errorsWithSourceCode;

LLMCall(jobSystem, "LLM", fixErrorsPrompt); //instantiate LLM job thread to fix errors in
C++ code

jobSystem.destroyJobSystem();

//read in output.json file and check if it says "No errors found!"

```

```

// Open the file
std::ifstream file("./Data/output.json");
if (!file.is_open()) {
    std::cerr << "Could not open the file 'output.json'" << std::endl;
    return 1;
}

// Read the file into a string
std::stringstream buffer;
buffer << file.rdbuf();
std::string fileContent = buffer.str();

// Check if the file contains "No errors found!"
if (fileContent.find("No errors or warnings - compiled successfully") != std::string::npos) {
    std::cout << "No errors found!" << std::endl;
    break; //exit the while loop if no errors are found
} else {

    //continue the while loop if errors are found
    std::cout << "Errors were found or the specific message is not present." << std::endl;
}

// Close the file
file.close();

i++; //increment i to exit the while loop after 10 iterations so that github actions doesn't run
forever

}

//This function can be uncommented to easily reload the errors in the ./test-code directory
from the files in the ./test-code-backup directory.

```

```
//The ./fixed-test-code directory also exists if you would like to test fixed code with no errors  
by pasting that code into the ./test-code directory.
```

```
//reloadErrors(); //this function repopulates the ./test-code directory with the backup error files  
from the ./test-code=backups directory
```

```
return 0;  
  
}
```

--END OF SUMMARY, REST IS RUNNING DOCUMENTATION----

Dec 2nd 8:46 pm – objective: Get through threading working!

- Added functions in main to achieve lab 3 flowscript interpreting and lab 4 LLM Calling

```
#include <iostream>
```

```
#include "JobSystemAPI.h"
```

```
#include "customjob.h"
```

```
//helper functions
```

```
//TODO - deprecated function, remove later?
```

```
void waitForJobToComplete(MyJobSystemAPI& jobSystem, const std::string& jobType){
```

```

//wait for compilejob to finish
int jobStatus = jobSystem.getJobStatus(jobType);
int completed = 3;

//busy while loop for compilejob
while(jobStatus != completed){
    if(jobStatus == -1){
        std::cout << "invalid job, I cannot wait on a non-existant job. Exiting while-loop" <<
std::endl;
        break;
    }
    std::cout << "waiting for" << jobType << " to complete" << std::endl;
    jobStatus = jobSystem.getJobStatus(jobType);
}

std::cout << jobType + " finished!" << std::endl;
}

```

```

void flowScriptInterpreter(MyJobSystemAPI& jobSystem, const std::string&
jobType){//interpreting flowscript with this function

```

```

    //call API class functions for testing

```

```

    //MyJobSystemAPI jobSystem; //create job system object

```

```

    //jobSystem.initialize(); //initialize job system -- creates threads in isolated function to avoid
creating too many threads & running out of resources

```

```

    //TODO- for lab 3 - main needs to have the flowscript files and needs to create and pass files
to interpreter job

```

```
//read in flowscript files and pass to interpreter job

//customJob registering:
//TODO - pass jobs vector into here by ref
jobSystem.registerCustomJob(CustomJob::CreateCustomJob, "customjob"); //register
custom job

jobSystem.createCustomJob("customjob"); //create custom job -- parses flowscript files
//TODO - customjob can have dual functionality, can ExecuteParse to read flowscript files
and can Execute to call to LLMCall.py, do I need to separate this functionality or is this okay to
have in one job thread? check rubric.

//std::string flowScriptFile = "Data/fsp1.txt";
//jobSystem.setJobSystemObj(&jobSystem); //set job system object for custom job

//create native jobs
jobSystem.createJob("countingjob"); //create counting job //TODO - add to api functionality
completely just like interpreter job
jobSystem.createJob("compilejob"); //create compile job
jobSystem.createJob("parsingjob"); //create parsing job
jobSystem.createJob("outputjob"); //create output job

//TODO - pass jobs vector into here by ref
//interpret flowscript files
jobSystem.createJob("interpreterjob"); //create interpreter job

//jobSystem.initialize(); //initialize job system -- creates threads in isolated function to avoid
creating too many threads & running out of resources
jobSystem.queueSpecJob("interpreterjob"); //queue interpreter job
```

//TODO - is this going to malfunction? Does this wait work? --- says cannot wait on invalid job because job finishes before getting to this wait function, so is this wait function irrelevant?

```
//waitForJobToComplete(jobSystem, "interpreterjob"); //wait for interpreter job to finish
```

```
//assume flowscript jobs are known!!! Create and register them here!!!
```

```
// jobSystem.registerCustomJob(CustomJob::CreateCustomJob, "customjob"); //register custom job
```

```
// jobSystem.createCustomJob("customjob"); //create custom job
```

//TODO - combine into one single register function later? All main should do is register jobs pre-execution of interpreter, and this does it but is convoluted in two lines

```
// jobSystem.registerCustomJob(CustomJob::CreateCustomJob, "customjob"); //register custom job
```

```
// jobSystem.createCustomJob("customjob"); //create custom job
```

```
//jobSystem.startJobSystem(); //add jobs to job system
```

```
//jobSystem.destroyJobSystem();
```

```
}
```

```
void LLMCall(MyJobSystemAPI& jobSystem, const std::string& jobType){ //instantiate LLM job thread
```

```
    std::string promptToPassIn = "Answer question based on the context. Context: Here is code in a json file format with errors. Please fix these errors and add a description of what you did to fix the errors. Question: ";
```

```

//make custom job to call to LLM
jobSystem.registerCustomJob(CustomJob::CreateCustomJob, "LLM"); //register custom job
jobSystem.createCustomJob("LLM"); //create custom job
//give customJob prompt and IP address
jobSystem.getLLMJob()->SetPrompt(promptToPassIn); //returns LLM job type for
manipulation
jobSystem.getLLMJob()->SetIP("Dummy-there-is-no-ip"); //sets the IP value for the LLM job
type. In this case localhost 4891 port #.

//jobSystem.initialize(); //initialize job system -- creates threads in isolated function to avoid
creating too many threads & running out of resources

//jobSystem.queueSpecJob("compilejob"); //queue interpreter job
//jobSystem.queueSpecJob("parsingjob"); //queue interpreter job
//jobSystem.queueSpecJob("outputjob"); //queue interpreter job
// waitForJobToComplete(jobSystem, "compilejob");
// waitForJobToComplete(jobSystem, "parsingjob");
// waitForJobToComplete(jobSystem, "outputjob");

jobSystem.queueSpecJob("LLM"); //queue interpreter job

//TODO - is this going to malfunction? Does this wait work? --- says cannot wait on invalid job
because job finishes before getting to this wait function, so is this wait function irrelevant?
//waitForJobToComplete(jobSystem, "LLM"); //wait for LLM job to finish
}

int main(void){
//call API class functions for testing

```



```
MyJobSystemAPI jobSystem; //create job system object

jobSystem.initialize(); //initialize job system -- creates threads in isolated function to avoid
creating too many threads & running out of resources

flowScriptInterpreter(jobSystem, "interpreterjob"); //interpreting flowscript with this function
//LLMCall(jobSystem, "LLM"); //instantiate LLM job thread

// jobSystem.registerCustomJob(CustomJob::CreateCustomJob, "customjob"); //register
custom job

// jobSystem.createCustomJob("customjob"); //create custom job

// //std::string flowScriptFile = "Data/fsp1.txt";
// //jobSystem.setJobSystemObj(&jobSystem); //set job system object for custom job

// //create native jobs

// jobSystem.createJob("countingjob"); //create render job //TODO - add to api functionality
completely just like interpreter job

// jobSystem.createJob("compilejob"); //create compile job
// jobSystem.createJob("parsingjob"); //create parsing job
// jobSystem.createJob("outputjob"); //create output job

// jobSystem.createJob("interpreterjob"); //create interpreter job

// jobSystem.initialize(); //initialize job system -- creates threads in isolated function to avoid
creating too many threads & running out of resources

// jobSystem.queueSpecJob("interpreterjob"); //queue interpreter job

//make compilejob
```

```
//jobSystem.createJob("compilejob"); //create compile job
//jobSystem.createJob("parsingjob"); //create parsing job
//jobSystem.createJob("outputjob"); //create output job

//prompt passing to LLM customjob without code var. Code added later in customJob.cpp.
//TODO - change this later to have code appended to prompt here?

//TODO - here I am relying on AI response so the json file format can be changed by
changing this prompt

// //make custom job to call to LLM
// jobSystem.registerCustomJob(CustomJob::CreateCustomJob, "LLM"); //register custom job
// jobSystem.createCustomJob("LLM"); //create custom job
// //give customJob prompt and IP address
// jobSystem.getLLMJob()->SetPrompt(promptToPassIn); //returns LLM job type for
manipulation

// jobSystem.getLLMJob()->SetIP("Dummy-there-is-no-ip"); //sets the IP value for the LLM job
type. In this case localhost 4891 port #.

// jobSystem.initialize(); //initialize job system -- creates threads in isolated function to avoid
creating too many threads & running out of resources

// //jobSystem.queueSpecJob("compilejob"); //queue interpreter job
// //jobSystem.queueSpecJob("parsingjob"); //queue interpreter job
// //jobSystem.queueSpecJob("outputjob"); //queue interpreter job
// // waitforJobToComplete(jobSystem, "compilejob");
// // waitforJobToComplete(jobSystem, "parsingjob");
// // waitforJobToComplete(jobSystem, "outputjob");

// jobSystem.queueSpecJob("LLM"); //queue interpreter job
```

```

jobSystem.destroyJobSystem();

return 0;

//TODOS I COULD NOT GET TO BUT THAT SHOULD BE DONE IN THE FUTURE
//1. use the job system after you stop it.... the jobssystem can be used but you get seg faults
and have to hard exit with exit(0); command
//2. be able to query the status of jobs that were reordered for dependencies with dependency
function. As I currently have it, the dependency function makes new ID's for each job in the
order they need to be in, and these dependencies are not updated in the job system, so the
status of the jobs will not be able to be found if a dependency is ran... This could be it.. or it
could just be threads running too quickly wiht concurrency etc. The jobstatus queries that were
not being found were being queried after the job system was stopped with stopjobssystem
function, so this whole comment could be an overreaction, but I should still investigate this to
make sure...

}

```

In main instantiating and running interpreterJob by calling flowScriptInterpreter(jobSystem, "interpreterjob"); function. This function makes interpreter job and runs it. Interpreter job was given a new function called "ExecuteParse". This ExecuteParse function takes the functionality of customJob's "Execute" function in lab 3 because customJob talks to the LLM in this lab with its "Execute" function, so the "ExecuteParse" functionality had to be moved from customJob's "Exeucute" function elsewhere. I did not want to have to add ExecuteParse to every single job thread, so I did not add ExecuteParse to Job.h, instead I added ExecuteParse to the interpreter.cpp/h file for encapsulation purposes because the ExecuteParse function will only need to be used once inside of interpreter to parse the flowscript files.

```
#include "interpreter.h"
#include <iostream>
#include <string>
#include <array>
#include <sstream> //for stringstream to split output into words by spaces
#include "fstream" //for writing to file
#include "../lib/json.hpp" //for json file writing

#include "../lib/job.h"
#include "../lib/jobsystem.h"

//for job manipulation in interpreter class
#include "JobSystemAPI.h" //for kicking off jobs with the parsed flowscript....
#include "customjob.h"

using json = nlohmann::json; //for json manipulation

#include "antlr4-runtime/antlr4-runtime.h"
#include "antlr/grammar-files/FlowscriptGrammarLexer.h"
#include "antlr/grammar-files/FlowscriptGrammarParser.h"

#include "customvisitor.h" //for visiting the node_definitions in the tree

#include "customerrorlistener.h" //for flowscript error logging

void Interpreter::ExecuteParse(){//TODO - this function should be in a separate thread and not
interpreter? in lab 3 it was in customjob.cpp as "Execute" function...

    std::cout << "EXECUTING PARSING FROM CUSTOMJOB" << std::endl;
```

```

CustomErrorListener errorListener; //create error listener object

// Path to flowscript input file taken in
std::ifstream stream("./Data/fsp3.txt"); // Adjust the path to the location of your .txt file
if (!stream) {
    std::cerr << "Cannot open file: ../Data/fsp#.txt" << std::endl;
    return; // Handle the error as you see fit
}
antlr4::ANTLRInputStream input(stream);

// Create a lexer that feeds off of the input stream
FlowscriptGrammarLexer lexer(&input);

// Create a buffer of tokens pulled from the lexer
antlr4::CommonTokenStream tokens(&lexer);

// Create a parser that feeds off the tokens buffer
FlowscriptGrammarParser parser(&tokens);

//attach the customerror listener to the parser
parser.removeErrorListeners(); //remove default error listener
parser.addErrorListener(&errorListener); //add custom error listener

//attach the custom listener to the lexer
lexer.removeErrorListeners(); //remove default error listener
lexer.addErrorListener(&errorListener); //add custom error listener

// Begin parsing at the root rule (replace 'script' with your actual root rule name)
antlr4::tree::ParseTree *tree = parser.flowscript();

```

```

//parsing now complete, so write errors to json file if there are any errors
//errorListener.writeErrorsToFile("./Data/flowscript-errors.json");

// Now you can use the tree however you need to
//std::cout << tree->toStringTree(&parser) << std::endl; //TODO - use this tree to run the jobs
in interpreter situationally based on what is in the tree.....

std::cout << "visiting tree!" << std::endl;

//TODO - use visitor design pattern to visit tree

//TODO - JobQUEUE is holding info, but jobLabels is not... why?
CustomVisitor visitor; //create visitor object
visitor.visit(tree); //visit tree. Will call visitNode_definition function in customvisitor.cpp and
push back the strings "Job A", "Job B", and "Job C" to the jobQueue vector in customvisitor.cpp

// Aggregate errors from errorListener and visitor
json combinedErrors = json::array(); // Initialize as an empty JSON array

for (const auto& error : errorListener.getErrors()) {
    combinedErrors.push_back(error);
}

//TODO - commented out because customerrorlistener class gets both errors. Left in for future
reference & feature implementation
// for (const auto& error : visitor.getSemanticErrors()) {
//     combinedErrors.push_back(error);

```

```

// }

// Write to file
std::ofstream errorFile("./Data/flowscript-error-output.json");
if (errorFile.is_open()) {
    errorFile << combinedErrors.dump(4);
    errorFile.close();
} else {
    std::cerr << "Unable to open error log file." << std::endl;
}

//TODO - jobLabels holds "outputjob", "compilejob", "parsingjob", "outputjob"... this out of
order is from grabbing only 4 of the 6 strings from the customvisitor jobqueue because
customjob is a thread that finishes before the customvisitor class is done running... make a
control structure for customjob to wait for costumvisitor to finish and make sure that jobQueue
only holds three string values, not 6

jobLabels = visitor.getJobLabels(); // Get the job labels

//iterate through jobLabels to check if they are correct
// for(const auto& label : jobLabels){
//     std::cout << "\n\nCUSTOMJOB JOB LABELS: \n\n" << label << std::endl;
// }

// Print the job labels (optional)

//write label to a file
std::ofstream file("./Data/jobprocessflow.txt"); //open file to write to
for (const std::string& label : jobLabels) {
    file << label << std::endl;

    //TODO - printing labels for testing only...

```

//std::cout << label << std::endl; //jobs original naming convention in the fsp1.txt file was "Job A", "Job B", "Job C", but they were changed to "compilejob", "parsingjob", and "outputjob" respectfully for testing purposes.

} //TODO - get labels vector into interpreter and run jobs based on those strings. Here the strings are "Job A", "Job B", "Job C" but for testing purposes I should change these strings in the fsp1.txt file to be "compilejob", "parsejob", "outputjob" to ensure I can run the jobs in the jobssystem. Then I can adopt the jobssystem's string identifiers it uses to run the native jobs to match the flowscript name conventions later. ASK HIM ABOUT THIS NAMING CONVENTION FOR STRING IDENTIFIERS.

file.close(); //close file

std::cout << "CustomJob [ID: " << this->GetUniqueID() << "] has been executed." << std::endl;

//call jobcompletedCallback

JobCompletedCallback();

}

void Interpreter::Execute(){

std::cout << "INTERPRETING" << std::endl;

//TODO - ADD INTERPRETER FUNCTIONALITY IN ALL API SECTIONS, RIGHT NOW CAN JUST CREATE INTERPRETER JOB

// jobSystemObj->registerCustomJob(CustomJob::CreateCustomJob, "customjob"); //register custom job

// jobSystemObj->createCustomJob("customjob"); //create custom job

// Job* job = jobsForAPIMap["customjob"];

//TODO - PARSE WITH ANTLR AND EXECUTE CUSTOMJOBS THAT ARE PRE-REG IN MAIN HERE!!!


```

//execute customJobs
//jobsForAPIMap["customjob"]->Execute();

//TODO - this should be done by customjob.cpp or no? Right now it is handled by the above
function "ExecuteParse" in interpreter.cpp. in lab 3 it was handled by customjob.cpp execute
function. bad practice to have one thread do multiple things...

//jobsForAPIMap["customjob"] -> ExecuteParse(); //parse flowscript files like in lab 3
ExecuteParse(); //parse flowscript files like in lab 3

//TODO - PASS VECTOR INTO CUSTOMJOB AND WAIT FOR CUSTOMJOB TO FINISH.
PASS VECTOR WHEN YOU CREATE CUSTOMJOB IN MAIN.... THEN ALSO PASS THAT
SAME VECTOR INTO INTERPRETER....

//TODO - busy wait for customjob to finish, then execute the rest of the jobs... eventually
move to a condvar wait like in JobSystemAPI.cpp queueSpecJob function...


//getter for setting the vector to the parsed job identifiers from the customjob
//TODO - remove customjob getter and setter....


//jobs = {"compilejob", "parsingjob", "outputjob"};

//TODO - return customjob parsing in a stream and then use that stream to call the three
native jobsystem jobs below...

//TODO - want to return line by line because doing approach one...


//read in strings from joblables.txt
std::ifstream file("./Data/jobprocessflow.txt"); //open file
std::string jobType;
while(std::getline(file, jobType)){ //get each line from file

    //std::cout << "EXECUTING JOB: " << jobType << std::endl;

```

```

//TODO - temporary conditional ignore for bad strings. remove once reading by
edge_definition not node_definition

std::cout << "EXECUTING JOB: " << jobType << std::endl;

if(jobType == "controlstatement"){
    CondFlow(); //execute conditional flow
    break;
}

else if(jobType == "loopstatement"){
    Loop(); //execute loop logic
    jobsForAPIMap["outputjob"]->Execute(); //execute output job
    break; //break out of while loop -- done!

    //TODO - replace break with something that actually really exits the entire else
statement...
}

//TODO - add a check here with a hardcoded set or the same problem? redundant???
Already did this with jobQueue?

//else if(){
    jobsForAPIMap[jobType]->Execute(); //execute job

//}

}

//jobsForAPIMap[jobType]->Execute(); //execute job

file.close(); //close file
//delete joblabels.txt file
//std::remove("./Data/joblabels.txt"); //delete file

```

//overloading execute function from job.h base class, so execute can call these, based on string? too confusing...

//execute needs to take flowscript parsed data passed in and call the jobs based on that data.

//either pass a string to execute and have execute call the correct job based on the string, or have execute take in a vector of jobs and call them all.

//OR give customjob the ability to call those functions here?

//jobsForAPIMap["customjob"]->ExecuteCompileJob(); //execute compile job

//would like to do this but cannot because job.h has no ExecuteCompileJob, so I could change job.h and add the job functionality, but thats bad encapsulation....

//he does not care about how I run the customjob or what I put in it, so I should mess with its innerworkings...

//TODO - make jobSystemAPI a friend class to interpreter.cpp so that interpreter can access jobSystemAPI's data members in a cleaner way? Would not have to pass in jobsForAPI map

//TODO - WHY AM I USING CUSTOMJOB HERE? BECAUSE RUNTIME CAN GET AROUND JOBSYSTEM ERRORING OUT WHEN STARTED TWICE? HOW DO I RUN OTHER THREADS OFF THIS CHILD THREAD?????

//TODO - //jobSystemObj->addJobs(); //add jobs to job system

// jsForAPI -> QueueJob(job); //queue custom job

//TODO - decouple the create threads function and add jobs to the job system function.

```

    //jobSystemObj->startJobSystem(); //start job system, will create threads and run jobs in
background

    //MyJobSystemAPI jobSystem;

    //if we parsed compileJob

    // jobSystem.registerCustomJob(CustomJob::CreateCustomJob, "customjob"); //same
functionality as compilejob, runs in background to demonstrate custom jobs at runtime. Can
replace compilejob as the only call needed for this type of job.

    // jobSystem.createCustomJob("customjob");


    //TODO - get jobsystem from main.cpp into interpreter.cpp

    //TODO - job systems are linked, so you are working with one class object that manages a
single job system. The job system is created in the JobSystemAPI.cpp

    //TODO - THE OBJECT IN MAIN IS WHAT ALLOWS YOU TO CREATE CUSTOM JOBS,
PASS THAT OBJECT HERE!!!! & CREATE CUSTOM JOBS BASED ON THE ANTLR
PARSING!!!

    std::cout << "Interpreter [ID: " << this->GetUniqueID() << "] has been executed." << std::endl;
    JobCompletedCallback();
}

void Interpreter::JobCompletedCallback(){
    std::cout << "interpreter Job done" << std::endl;
    //notify completion
    signalCompletion(); //TODO - encapsulated version of thread control with condvar
}

void Interpreter::CondFlow(){

```

```

if(fsCond1 > fsCond2){
    std::cout << "fsCond1 > fsCond2" << std::endl;
    jobsForAPIMap["countingjob"]->Execute(); //execute compile job
} else{
    std::cout << "fsCond1 < fsCond2" << std::endl;
    jobsForAPIMap["parsingjob"]->Execute(); //execute parsing job
    jobsForAPIMap["outputjob"]->Execute(); //execute output job
}
}

void Interpreter::Loop(){
    //change loopHit to 1 to confirm we are in the loop
    std::ofstream loopHitFile("./Data/loopHit.txt"); //open file
    if(!loopHitFile){
        std::cout << "ERROR: FILE NOT FOUND" << std::endl;
    }
    loopHitFile << "1" << std::endl; //write 1 to file
    loopHitFile.close(); //close file
    //grab boolean value from file
    std::ifstream file("./Data/loopbool.txt"); //open file
    if(!file){
        std::cout << "ERROR: FILE NOT FOUND" << std::endl;
    }
    std::string boolData;
    std::getline(file, boolData); //get each line from file
    file.close(); //close file
    loopCond = std::stoi(boolData); //convert string to int
    if(loopCond == 0){ //false, so run parsingjob again
        //TODO - The parsingJob called above this loopCond should not increment the
        parsingJobCounter file... parsingJobCounter file should only be incremented in this loop...
    }
}

```

```
jobsForAPIMap["parsingjob"]->Execute(); //execute output job
Loop(); //call loop again - recursion
}

//erase parsingJobLoopCounter.txt file and write 0 to it
std::ofstream file2("./Data/parsingJobLoopCounter.txt"); //open file to write to
file2 << "0" << std::endl; //write 0 to file
file2.close(); //close file

//erase loopbool.txt file and write 0 to it
std::ofstream file3("./Data/loopbool.txt"); //open file to write to
file3 << "0" << std::endl; //write 0 to file
file3.close(); //close file

// //write 0 to loopHit.txt file to confirm we are out of the loop
// std::ofstream loopHitFile2("./Data/loopHit.txt"); //open file
// if(!loopHitFile2){
//     std::cout << "ERROR: FILE NOT FOUND" << std::endl;
// }
// loopHitFile2 << "0" << std::endl; //write 0 to file
// loopHitFile2.close(); //close file

//TODO - is outputjob execution needed here?...
// //boolean evaluated to false so run outputjob
// jobsForAPIMap["outputjob"]->Execute(); //execute output job
}
```

```
//code automatically compiling means you can:  
//set code off and do other things  
//automated regression testing -> compile code and automatically run tests on code.  
//when errors output it tells you error number and error messages -> put error messages into  
LLM's to fix errors  
//take code file and web info from LLM and say "fix it for me"  
//AI TOOL TO FIX CODE FOR YOU -> BETTER STANDARDIZATION = BETTER FIXES  
//Can write as terminal job instead of compile job (written as compile job here)
```

Now testing running both `flowScriptInterpreter(jobSystem, "interpreterjob")` function and `LLMCall(jobSystem, "LLM")` at once. This works for testing now because the `output.json` is preinstantiated for the LLM to read, but I need to add a wait between these function calls so that the jobs from `flowScriptInterpreter` can run before the LLM call has to read the `output.json` file. This wait will ensure that the `output.json` file is instantiated before the `LLMCall(jobSystem, "LLM")` function calls the LLM and passes the `output.json` as input to the LLM.

```
flowScriptInterpreter(jobSystem, "interpreterjob"); //interpreting flowscript with this function  
//TODO - need to add a wait here so that the Output.json is instantiated before the LLM job is  
called.  
//TODO - currently not getting response from LLM because output is null because LLM is  
called before have an output.json file to append as input to LLM  
//TODO - works right now because the output.json from previous lab runs is instantiated, but  
test without a preinstantiated output.json and see if it works. Also how are you going to handle  
clases were output.json is not instantiated by flowscript interpretation? I dont think this is  
required to handle.
```

```
LLMCall(jobSystem, "LLM"); //instantiate LLM job thread
```

Notes 12/4/23

Broke lab up into sections for better grasp of lab goals:

Lab final by section

section one: LLM write valid flowscript

- Using an LLM, have it write FlowScript to build a compiling pipeline that will return all errors from the compilation of C++ code project (Lab 1, 2 and 4) <-- prompt
- You will need to provide single/few shot training to ensure the LLM writes valid FlowScript
- Ideally, this would be something that could be launched as a job to write FlowScript for a given prompt, but it can be customized to focus specifically on compiling
- You will need to provide the LLM with a list of jobs that are available, so it can write the proper FlowScript to execute those jobs.
- Using an LLM, have it submit and execute a FlowScript and obtain the results. (Lab 3 and 4)

Section two: LLM suggests fixes based on compiling errors

- Using an LLM, have it suggest and rewrite a section/all of a code file based upon compilation results (Lab 4).
- Using an LLM, provide feedback on errors within a file with an explanation of changes that need to be made to correct the errors. (Lab 4)

- Using an LLM, have it verify the results from suggested code improvements to verify the suggestion resolved compilation errors, and repeat the process if not (Lab 4)

HOUSEKEEPING:

CAN USE ANY ONE LLM MODEL

JOB CAN BE WRITTEN IN ANY LANGUAGE BUT MUST BE LAUNCHED FROM JOB SYSTEM

I am using Mistral Instruct for this lab:

```
modelName = "mistral-7b-instruct-v0.1.Q4_0.gguf"
```

SECTION ONE CODE:

Interpreting flowscript and running code pipeline

Updated TODOs in main.cpp 12/4/23 3:26pm

```
//TODO - write these prompts. The fix errors prompt should be pulled from the
LLMCall.py/customjob.cpp file that already exists and be passed in to there...

std::string writeFlowScriptPrompt = ".....";
std::string fixErrorsPrompt = ".....";

//TODO - pass a generic prompt to this thread from main so that you can run this function
multiple times and create multiple LLM jobs in the running of main. One job should write the
flowscript with zero/fewshot training, then the second call should fix the errors with a while
loop...

LLMCall(jobSystem, "LLM"); //instantiate LLM job thread

//interpret flowscript that LLMCall writes on line 121 of main.cpp
```

Took this prompt for the LLM and moved it to main() as promptToPassIn:

```

int main(void){
    //call API class functions for testing
    MyJobSystemAPI jobSystem; //create job system object

    jobSystem.initialize(); //initialize job system -- creates threads in isolated function to avoid
    creating too many threads & running out of resources

    std::string promptToPassIn = "Answer question based on the context. Context: Here is code
    in a json file format with errors. Please fix these errors and add a description of what you did to
    fix the errors. Question: ";
    Added these three helper functions and headers to main.cpp:

#include <fstream> //for reading in files
#include "../lib/json.hpp" //for json manipulation

using json = nlohmann::json;

//helper functions

std::string readFileContents(const std::string& filePath) {
    //call processFile function to read file contents
    std::string file_content;
    std::ifstream file(filePath);
    if (file.is_open()) {
        file_content.assign((std::istreambuf_iterator<char>(file)),
                           std::istreambuf_iterator<char>());
        file.close();
    } else {
        std::cerr << "Unable to open file: " << filePath << std::endl;
    }
}

```

```

    }

    return file_content;
}

std::string processFile(const std::string& filePath) {
    // Check if the file path ends with ".json"
    if (filePath.size() >= 5 && filePath.substr(filePath.size() - 5) == ".json") {
        std::string file_content = readFileContents(filePath);

        // Parse the JSON data
        json json_data = json::parse(file_content);

        // TODO: Reformat json_data as needed

        // Serialize JSON object to string
        return json_data.dump(4); // '4' for pretty printing
    }
    else if (filePath.size() >= 4 && filePath.substr(filePath.size() - 4) == ".txt") {
        // For .txt files, just read and return the file content
        return readFileContents(filePath);
    }
    else {
        std::cerr << "Unsupported file type." << std::endl;
        return "";
    }
}

std::string escapeShellArgument(const std::string& arg) {
    std::string escaped;

```

```

for (char c : arg) {
    // Add a backslash before special characters
    if (c == '"' || c == '\\' || c == '$') {
        escaped += '\\';
    }
    escaped += c;
}
return escaped;
}

```

These functions in main.cpp let me create a code string variable in main so that I can modify the prompt to give to the LLM in main.cpp. This makes the LLMCall function generic and able to work with all prompts:

```

int main(void){
    //call API class functions for testing
    MyJobSystemAPI jobSystem; //create job system object

    jobSystem.initialize(); //initialize job system -- creates threads in isolated function to avoid
    creating too many threads & running out of resources

    // JSON file path
    std::string json_file_path = "./Data/output.json";

    std::string code = processFile(json_file_path); //read the json file and store the output in a
    string called code

    code = escapeShellArgument(code); //escape the code string to be used in the command
    string below

```

```
std::string promptToPassIn = "Answer question based on the context. Context: Here is code  
in a json file format with errors. Please fix these errors and add a description of what you did to  
fix the errors. Question: " + code;
```

Adding these helper functions to get code errors in main to make prompts generic for LLMCall, now can give LLMCall function any prompts by passing them in. This lets LLMCall write flowscript with few shot training or fix code errors based on prompt all from main.cpp call.

Few Shot training:

Applying processFile function and escapeShellArgument helper functions in main to grab the example flowscript from fsp1.txt in my ./Data/ directory. I want to grab this fsp1.txt file so that I can use few shot training on the LLM so that it can write the correct flowscript:

```
//JSON file path for flowScript prompt  
std::string flowScriptPromptFilePath = "./Data/fsp1.txt";  
std::string flowScript = processFile(flowScriptPromptFilePath); //read the json file and store  
the output in a string called code  
flowScript = escapeShellArgument(flowScript); //escape the code string to be used in the  
command string below
```

This code stores the flowScript from fsp1.txt into a string called flowScript and escapeShellArgument ensures that it can correctly be parsed by the LLMCall.py function once it is passed in the prompt without being treated weirdly by my parsing method. Now I can append the flowScript example to the end of my prompt for the LLM:

```
std::string writeFlowScriptPrompt = "Please write flowscript code based on the context.  
Context: Here is flowscript for you to base your flowscript off of. The flowScript must connect  
jobs in order where 'compilejob' is the first job, 'parsingjob' is the second job, and 'outputjob' is
```

the third job. Only those three jobs should be created. Question: Please write flowscript that mimics this flowscript: " + flowScript;

This prompt uses few shot training by giving the LLM an example of flowscript and expecting it to write similar flowScript. The LLM prompt specifies that the jobs should be named ‘compilejob’, ‘parsingjob’ and ‘outputjob’ so that the jobssystem can run these job types later. The prompt also specifies that these three jobs are the only ones that the LLM should create, so that the LLM does not connect them with weird random job names. Before I included the specifics that the LLM should connect jobs in a specific order and only use those three job naming conventions, I would get output like this:

```
digraph JobGraph {  
  job1 -> job2;  
  job3 -> job4;  
}
```

And like this:

```
css  
digraph JobGraph {  
  compilejob -> parsingjob;  
  parsingjob -> outputjob;  
}
```

It was only when I specified that the LLM should use those specific job names and just generate those three jobs and nothing else in the flowScript that I got this final satisfactory output:

```
digraph JobGraph {  
  compilejob -> parsingjob;  
  parsingjob -> outputjob;
```

```
}
```

I also had to change my customjob.cpp prompt to be generic and not append code to itself:

original:

```
std::cout << "\n\nPROMPT FROM MAIN: \n\n" << promptFromMain << std::endl;
```

updated line:

```
std::string prompt = promptFromMain; //Generic prompt passed in from main.cpp with  
LLMCall() function
```

This generic prompt lets any prompt be passed into the customjob.cpp thread from main, which lets the LLM do anything the prompt specifies, whether it be flowScript writing or fixing coding errors.

Running the job system based on the flowScript that the LLMCall.py generates:

First I had to make LLMCall.py write its flowscript to a .txt file to be read by the job that had to run the flowScript. Initially I was writing my flowscript to a .txt file like this:

Please write flowscript code based on the context. Context: Here is flowscript for you to base your flowscript off of. The flowScript must connect jobs in order where 'compilejob' is the first job, 'parsingjob' is the second job, and 'outputjob' is the third job. Only those three jobs should be created. Question: Please write flowscript that mimics this flowscript: digraph JobGraph {

```
compilejob -> parsingjob;  
parsingjob -> outputjob;
```

```
}
```

Here is an example of FlowScript code that connects the three jobs in the specified order:

```
```vbnet
digraph JobGraph {
 compilejob -> parsingjob;
 parsingjob -> outputjob;
}
```
```

This code creates a directed graph with three nodes, 'compilejob', 'parsingjob', and 'outputjob'. The edges between the nodes represent the flow of data or execution. In this case, the first job is 'compilejob' which will execute before 'parsingjob', and 'parsingjob' will execute before 'outputjob'.

As you can see, this is everything that was given to the LLM and its output, which is not what we want. All we want is this to be written to a .txt file:

```
digraph JobGraph {
    compilejob -> parsingjob;
    parsingjob -> outputjob;
}
```

To do that, I added this code to the end of my LLMCall.py file:

```
#specify the outputfile path
output_file_path = "./Data/Flowscript-Response.txt"

#extract the text from the response
generated_text = response['choices'][0]['text']

# Split the text into lines
lines = generated_text.splitlines()

# Find the line that starts with "digraph JobGraph {"
```



```

start_index = next((i for i, line in enumerate(lines) if line.strip().startswith("digraph JobGraph {")),
None)

if start_index is not None:
    # Find the matching closing brace "}"
    end_index = next((i for i, line in enumerate(lines[start_index:], start=start_index) if line.strip()
== "}"), None)

    if end_index is not None:
        # Extract the "digraph JobGraph { ... }" part
        extracted_text = "\n".join(lines[start_index:end_index + 1])

        # Specify the output file path
        output_file_path = "./Data/Flowscript-Response.txt"

        # Write the extracted text to a .txt file
        with open(output_file_path, "w") as output_file:
            output_file.write(extracted_text)
    else:
        print("No 'digraph JobGraph {' found in the response.")

```

This code is grabbing the filepath of where to output the .txt file in the output_file_path variable. Generated_text holds the text from the AI response, this is everything that goes into the LLM and comes out. Then the lines variable is created to split the text into lines and find the lines that start with digraph. The closing brace is located } and in between digraph and } is stored in the extracted_text variable. This content is then written to LLM-Flowscript-Response.txt inside of Data.

First I have to wait for the LLMCall that writes the flowscript to completely finish with this code:

The jobCompletedCallback has a 'signalCompletion' function here:

```
void CustomJob::JobCompletedCallback() {  
    std::cout << "CustomJob callback is done!" << std::endl;  
  
    //getJobLabels(); //return job labels to interpreter  
    signalCompletion(); //signal completion  
  
}
```

This signalCompletion function uses cv and notify one to signal to the jobssystem when a thread is done running before other threads can start. The signalCompletion is in job.h:

```
void signalCompletion() { //signals job complete  
    std::lock_guard<std::mutex> lock(jobMutex);  
    condition = true;  
    cv.notify_one();  
}
```

The flowscript is interpreted with this function in main.cpp:

```
flowScriptInterpreter(jobSystem, "interpreterjob"); //interpreting flowscript with this function
```

which creates the required jobs and creates and queues the interpreter job:

```

void flowScriptInterpreter(MyJobSystemAPI& jobSystem, const std::string&
jobType){//interpreting flowscript with this function
jobSystem.createJob("compilejob"); //create compile job
    jobSystem.createJob("parsingjob"); //create parsing job
    jobSystem.createJob("outputjob"); //create output job

    //TODO - pass jobs vector into here by ref
    //interpret flowscript files
    jobSystem.createJob("interpreterjob"); //create interpreter job

    //jobSystem.initialize(); //initialize job system -- creates threads in isolated function to avoid
    creating too many threads & running out of resources
    jobSystem.queueSpecJob("interpreterjob"); //queue interpreter job

```

My queueSpecJob looks like this in my JobSystemAPI.cpp:

```

void MyJobSystemAPI::queueSpecJob(const std::string& JobType) {//TODO - take out? Not
being used anymore?

    //TODO - add jobs to job system here for customJob...
    //for(const std::string& jobType : jobOrder){
        Job* job = jobsForAPI[JobType]; //retrieve specific job from map
        std::cout << "Queuing WITH JOB ID: " << job->GetUniqueID() << std::endl;

        std::promise<void> jobCompletionPromise;
        std::future<void> jobCompletionFuture = jobCompletionPromise.get_future();

        jsForAPI->QueueJob(job); //queue job
    }
}

```

```
//pass condition bool to child threads --> make a function in job that gets the condition bool,
sets it to true. Use that function in child jobs.
```

```
std::unique_lock<std::mutex> lock(jobMutex);

job->cv.wait(lock, [job]{ return job->condition; });
//job->waitForCompletion();
//}
}
```

This waits between jobs with the `cv.wait(lock, [job]{return job->condition; });` with a mutex before for thread control.

In the `interpreter.cpp` `Execute` runs first:

```
void Interpreter::Execute(){
    std::cout << "INTERPRETING" << std::endl;

    //TODO - ADD INTERPRETER FUNCTIONALITY IN ALL API SECTIONS, RIGHT NOW CAN
    JUST CREATE INTERPRETER JOB

    // jobSystemObj->registerCustomJob(CustomJob::CreateCustomJob, "customjob"); //register
    custom job

    // jobSystemObj->createCustomJob("customjob"); //create custom job

    // Job* job = jobsForAPIMap["customjob"];

    //TODO - PARSE WITH ANTLR AND EXECUTE CUSTOMJOBS THAT ARE PRE-REG IN
    MAIN HERE!!!

    //execute customJobs
```

```

//jobsForAPIMap["customjob"]->Execute();

//TODO - this should be done by customjob.cpp or no? Right now it is handled by the above
function "ExecuteParse" in interpreter.cpp. in lab 3 it was handled by customjob.cpp execute
function. bad practice to have one thread do multiple things...

//jobsForAPIMap["customjob"] -> ExecuteParse(); //parse flowscript files like in lab 3
ExecuteParse(); //parse flowscript files like in lab 3

```

And runs the ExecuteParse(); function. The ExecuteParse function interprets the flowScript file with antlr by making a complex syntax tree (CST):

```

void Interpreter::ExecuteParse(){//TODO - this function should be in a separate thread and not
interpreter? in lab 3 it was in customjob.cpp as "Execute" function...

```

```

std::cout << "EXECUTING PARSING FROM CUSTOMJOB" << std::endl;

CustomErrorListener errorListener; //create error listener object

// Path to flowscript input file taken in
std::ifstream stream("./Data/LLM-Flowscript-Response.txt"); // Adjust the path to the location
of your .txt file
if (!stream) {
    std::cerr << "Cannot open file: ../Data/LLM-Flowscript-Response.txt" << std::endl;
    return; // Handle the error as you see fit
}
antlr4::ANTLRInputStream input(stream);

// Create a lexer that feeds off of the input stream
FlowscriptGrammarLexer lexer(&input);

// Create a buffer of tokens pulled from the lexer

```

```

antlr4::CommonTokenStream tokens(&lexer);

// Create a parser that feeds off the tokens buffer
FlowscriptGrammarParser parser(&tokens);

//attach the customerror listener to the parser
parser.removeErrorListeners(); //remove default error listener
parser.addErrorListener(&errorListener); //add custom error listener

//attach the custom listener to the lexer
lexer.removeErrorListeners(); //remove default error listener
lexer.addErrorListener(&errorListener); //add custom error listener

// Begin parsing at the root rule (replace 'script' with your actual root rule name)
antlr4::tree::ParseTree *tree = parser.flowscript();

//parsing now complete, so write errors to json file if there are any errors
//errorListener.writeErrorsToFile("./Data/flowscript-errors.json");

// Now you can use the tree however you need to
//std::cout << tree->toStringTree(&parser) << std::endl; //TODO - use this tree to run the jobs
in interpreter situationally based on what is in the tree.....

std::cout << "visiting tree!" << std::endl;

//TODO - use visitor design pattern to visit tree

//TODO - JobQUEUE is holding info, but jobLabels is not... why?
CustomVisitor visitor; //create visitor object

```

```
    visitor.visit(tree); //visit tree. Will call visitNode_definition function in customvisitor.cpp and  
    push back the strings "Job A", "Job B", and "Job C" to the jobQueue vector in customvisitor.cpp
```

```
// Aggregate errors from errorListener and visitor
```

```
json combinedErrors = json::array(); // Initialize as an empty JSON array
```

```
for (const auto& error : errorListener.getErrors()) {  
    combinedErrors.push_back(error);  
}
```

```
//TODO - commented out because customerrorlistener class gets both errors. Left in for future  
reference & feature implementation
```

```
// for (const auto& error : visitor.getSemanticErrors()) {  
//     combinedErrors.push_back(error);  
// }
```

```
// Write to file
```

```
std::ofstream errorFile("./Data/flowscript-error-output.json");
```

```
if (errorFile.is_open()) {  
    errorFile << combinedErrors.dump(4);  
    errorFile.close();  
} else {  
    std::cerr << "Unable to open error log file." << std::endl;  
}
```

```
//TODO - jobLabels holds "outputjob", "compilejob", "parsingjob", "outputjob"... this out of  
order is from grabbing only 4 of the 6 strings from the customvisitor jobqueue because  
customjob is a thread that finishes before the customvisitor class is done running... make a
```

control structure for customjob to wait for costumvisitor to finish and make sure that jobQueue only holds three string values, not 6

```
jobLabels = visitor.getJobLabels(); // Get the job labels

//iterate through jobLabels to check if they are correct
// for(const auto& label : jobLabels){
//     std::cout << "\n\nCUSTOMJOB JOB LABELS: \n\n" << label << std::endl;
// }
// Print the job labels (optional)
//write label to a file
std::ofstream file("./Data/jobprocessflow.txt"); //open file to write to
for (const std::string& label : jobLabels) {
    file << label << std::endl;
    //TODO - printing labels for testing only...
    //std::cout << label << std::endl; //jobs original naming convention in the fsp1.txt file was
    "Job A", "Job B", "Job C", but they were changed to "compilejob", "parsingjob", and "outputjob"
    respectfully for testing purposes.
} //TODO - get labels vector into interpreter and run jobs based on those strings. Here the
strings are "Job A", "Job B", "Job C" but for testing purposes I should change these strings in
the fsp1.txt file to be "compilejob", "parsejob", "outputjob" to ensure I can run the jobs in the
jobssystem. Then I can adopt the jobssystem's string identifiers it uses to run the native jobs to
match the flowscript name conventions later. ASK HIM ABOUT THIS NAMING CONVENTION
FOR STRING IDENTIFIERS.
file.close(); //close file
std::cout << "CustomJob [ID: " << this->GetUniqueID() << "] has been executed." <<
std::endl;
//call jobcompletedCallback
}
```


There is then a `jobCompletedCallBack` at the end of the `Execute` function in interpreter to alert that the interpreter job has completed:

```
std::cout << "Interpreter [ID: " << this->GetUniquelD() << "] has been executed." << std::endl;
JobCompletedCallback();
```

Made the following files .baks to test and make sure that the `flowScript` interpreter function actually created the output from lab one:

1. `compileJobOutput.json.bak`
2. `Errors.json.bak`
3. `Output.json.bak`

Then I removed these deprecated lines from `customjob.cpp` because the prompt to the LLM is passed from main now and not solely reliant on reading the `output.json` file anymore:

```
// // JSON file path
// std::string json_file_path = "./Data/output.json";

// std::string code = processFile(json_file_path); //read the json file and store the output in a
string called code

// code = escapeShellArgument(code); //escape the code string to be used in the command
string below
```

Then I ran these lines in `main.cpp`:

```
LLMCall(jobSystem, "LLM", writeFlowScriptPrompt); //instantiate LLM job thread to write
flowScript

//interpret flowscript that LLMCall writes on line 121 of main.cpp
flowScriptInterpreter(jobSystem, "interpreterjob"); //interpreting flowscript with this function
```

These lines generate and interpret the flowscript giving me these output files in my ./Data/ directory:

1. compileJobOutput.json
2. parsedErrors.json [renamed from errors.json to parsedErrors.json]
3. output.json

SECTION TWO CODE:

LLM Suggets fixes based on compiling errors

Picking up where we left off in main.cpp we have this:

```
//TODO - pass a generic prompt to this thread from main so that you can run this function
multiple times and create multiple LLM jobs in the running of main. One job should write the
flowscript with zero/fewshot training, then the second call should fix the errors with a while
loop...

LLMCall(jobSystem, "LLM", writeFlowScriptPrompt); //instantiate LLM job thread to write
flowScript

//interpret flowscript that LLMCall writes on line 121 of main.cpp
flowScriptInterpreter(jobSystem, "interpreterjob"); //interpreting flowscript with this function
```

These function calls talk to the LLM and interpret the flowscript files to generate the desired error output files. Now adding this to main:

```
// JSON file path for output.json file from lab one that contains coding errors
std::string json_file_path = "./Data/output.json";

std::string code = processFile(json_file_path); //read the json file and store the output in a
string called code

code = escapeShellArgument(code); //escape the code string to be used in the command
string below
```

```

std::string fixErrorsPrompt = "Answer question based on the context. Context: Here is code in
a json file format with errors. Please fix these errors and add a description of what you did to fix
the errors. Question: " + code;

LLMCall(jobSystem, "LLM", fixErrorsPrompt); //instantiate LLM job thread to fix errors in C++
code

```

Now calling to the LLM function a second time to instantiate another LLM thread, this prompt tells the LLM to fix the coding errors and return fixes. So far this prompt only fixes sections of the code and does not output the entire fixed code. To remedy this, we should modify the existing prompt to also take in the entire source files and tell the LLM to write new files C++ files that run correctly. Here is how the sourceCode files are being read in as a string:

```

//read source code files from ./test-code directory

std::string sourceCodeMain = "./test-code/main.cpp";
std::string sourceCodeTestOne = "./test-code/test-set-one.cpp";
std::string sourceCodeTestTwo = "./test-code/test-set-two.cpp";
std::string sourceCodeTestThree = "./test-code/test-set-three.cpp";

std::string combinedSourceCode = processFile(sourceCodeMain) +
processFile(sourceCodeTestOne) + processFile(sourceCodeTestTwo) +
processFile(sourceCodeTestThree); //read the json file and store the output in a string called
code

combinedSourceCode = escapeShellArgument(combinedSourceCode); //escape the code
string to be used in the command string below

std::string errorsWithSourceCode = code + combinedSourceCode;

```

Here is the new prompt:

```
std::string fixErrorsPrompt = "Answer question based on the context. Context: Here is code in a json file format with errors. Following this I have provided all test-set.cpp files that have the errors to give you a full view of each file. You should be able to come up with the .h files for each of these .cpp files. Using the errors provided in the json file and each of these .cpp files, I want you to fix the errors in each of these .cpp files and output each of these .cpp files in full, with all errors fixed. These fixed .cpp files should be returned as neatly formatted strings that can be written to .cpp files to be ran later. Add a description of what you did to fix the errors. Question: " + errorsWithSourceCode;
```

This prompt yields output like this:

```
css
test-code/test-set-one.cpp:
int main(){
<< "test_set_one" << std::endl;
//another_function();
}

test-code/test-set-three.cpp:
#include <iostream> ;
void test_set_three() {
cout << "test_set_three" << endl;
}

test-code/test-set-two.cpp:
#include <iostream>;
void test_set_two(){
cout << "test_set_two" << endl;
```

```
}
```

This is almost there but is failing to fix multiple errors in test-set-one for example. The prompt can be modified further:

```
std::string fixErrorsPrompt = "Answer question based on the context. Context: Here is code in  
a json file format with errors. Following this I have provided all test-set.cpp files that have the  
errors to give you a full view of each file. You should be able to come up with the .h files for  
each of these .cpp files. Using the errors provided in the json file and each of these .cpp files, I  
want you to fix the errors in each of these .cpp files. Each .cpp file can have multiple errors that  
need to be fixed. Output each of these .cpp files in full, with all errors fixed. Please also output  
their .h files. These fixed .cpp files should be returned as neatly formatted strings that can be  
written to .cpp files to be ran later. Add a description of what you did to fix the errors. Question: "  
+ errorsWithSourceCode;
```

The output is now like this for each .cpp file:

```
++  
#include <iostream>  
using namespace std;  
int main(){  
    cout << "test_set_one" << endl;  
}
```

Which includes multiple fixes. Mistral Instruct is still not outputting the .h files, but this can be fixed by giving the .h files in the errorsWithSourceCode string:

```
//read in the .h files  
std::string headerFileTestSetOne = "./test-code/test-set-one.h";  
std::string headerFileTestSetTwo = "./test-code/test-set-two.h";
```

```
std::string headerFileTestSetThree = "./test-code/test-set-three.h";

std::string combinedSourceCode = processFile(sourceCodeMain) +
processFile(sourceCodeTestOneCpp) + processFile(headerFileTestSetOne) +
processFile(sourceCodeTestTwoCpp) + processFile(headerFileTestSetTwo) +
processFile(sourceCodeTestThreeCpp) + processFile(headerFileTestSetThree); //read the json
file and store the output in a string called code
```

Here is the new prompt that supplies the .cpp and .h files:

```
std::string fixErrorsPrompt = "Answer question based on the context. Context: Here is code in
a json file format with errors. Following this I have provided all test-set.cpp files that have the
errors to give you a full view of each file. I have also provided the associated .h files for each of
these .cpp files. Using the errors provided in the json file and each of these .cpp files, I want you
to fix the errors in each of these .cpp files. Each .cpp file can have multiple errors that need to
be fixed. Output each of these .cpp files in full, with all errors fixed. Please also output the .cpp
files associated .h files. These fixed .cpp files and their associated .h files should be returned as
neatly formatted strings that can be written to .cpp and .h files to be ran later. Add a description
of what you did to fix the errors. Question: " + errorsWithSourceCode;
```

Here is the output from that prompt:

```
[
{
"File Path": "test-code/test-set-one.cpp",
"Fixed Code": "int main(){ << \"test_set_one\" <<
std::endl; //another_function();}",
"Associated .h File": "#ifndef
TEST_SET_ONE\n#define(TEST_SET_ONE)\n\nvoid
test_set_one();\n\nend{define}"
},
```

```
{
"File Path": "test-code/test-set-three.cpp",
"Fixed Code": "#include \"test-set-three.h\"\\nvoid
test_set_three() {\\nstd::cout << \"test_set_three\\\"
<< std::endl;\\n}\",
"Associated .h File": "#ifndef
TEST_SET_THREE\\n#define(TEST_SET_THREE)\\n\\nvoid
test_set_three();\\n\\nend{define}\"
},
{
"File Path": "test-code/test-set-two.cpp",
"Fixed Code": "#include <iostream> \\nvoid
test_set_two(){ std::cout << \"test_set_two\\\" <<
std::endl;}\",
"Associated .h File": "#ifndef
TEST_SET_TWO\\n#define(TEST_SET_TWO)\\n\\nvoid
test_set_two();\\n\\nend{define}\"
}
]
```

The LLM is trying to return them as strings to be written to files, but is failing to fix multiple errors.

Switching models to use Mistral OpenOrca in LLMCall.py:

```
modelName = "mistral-7b-openorca.Q4_0.gguf"
# modelName = "mistral-7b-instruct-v0.1.Q4_0.gguf"
```

Orca is a little smarter giving neater output like this:

```
cpp

#include "test-set-one.h"
int main() {
    << "test_set_one" << std::endl;
}

test-set-three.cpp:

cpp

#include "test-set-three.h"
void test_set_three() {
    cout << "test_set_three" << endl;
}

test-set-two.cpp:

cpp

include <iostream>
void test_set_two(){
    std::cout << "test_set_two" << std::endl;
}
```

Here is the last modification to the prompt, adding “some errors will not be listed in the .json errors file, so fix what you think needs to be fixed for the code to run at your own discretion”

```
std::string fixErrorsPrompt = "Answer question based on the context. Context: Here is code in a json file format with errors. Following this I have provided all test-set.cpp files that have the errors to give you a full view of each file. I have also provided the associated .h files for each of these .cpp files. Using the errors provided in the json file and each of these .cpp files, I want you to fix the errors in each of these .cpp files. Each .cpp file can have multiple errors that need to be fixed, sometimes not listed in the .json error file, so fix what you think needs to be fixed for the code to run at your own discretion. Output each of these .cpp files in full, with all errors fixed. Please also output the .cpp files associated .h files. These fixed .cpp files and their associated
```



```
.h files should be returned as neatly formatted strings that can be written to .cpp and .h files to be ran later. Add a description of what you did to fix the errors. Question: " + errorsWithSourceCode;
```

I realized that openOrca treats the flowScript interpreting prompt differently than mistral instruct, so I switched back to mistral instruct instead to preserve my flowscript interpreting as is because both mistral instruct and openOrca have trouble fixing all the errors in one pass, so I am just creating more work for myself if I switch to openOrca from mistralInstruct.

Here is Mistral instructs output with the current prompt for fixing errors:

Prompt:

```
std::string fixErrorsPrompt = "Answer question based on the context. Context: Here is code in a json file format with errors. Following this I have provided all test-set.cpp files that have the errors to give you a full view of each file. I have also provided the associated .h files for each of these .cpp files. Using the errors provided in the json file and each of these .cpp files, I want you to fix the errors in each of these .cpp files. Each .cpp file can have multiple errors that need to be fixed, sometimes not listed in the .json error file, so fix what you think needs to be fixed for the code to run at your own discretion. Output each of these .cpp files in full, with all errors fixed. Please also output the .cpp files associated .h files. These fixed .cpp files and their associated .h files should be returned as neatly formatted strings that can be written to .cpp and .h files to be ran later. Add a description of what you did to fix the errors. Question: " + errorsWithSourceCode;
```

Outputted FlowScript/Fixed Code output by Mistral Instruct:

P

Please write flowscript code based on the context. Context: Here is flowscript for you to base your flowscript off of. The flowScript must connect jobs in order second job, and 'outputjob' is the third job. Only those three jobs should be created. Question: Please write flowscript that mimics this flowscript: digraph Job

```
compilejob -> parsingjob;  
parsingjob -> outputjob;
```

```
}
```

R

Here's a sample FlowScript code based on the given context:

```
flowscript  
digraph JobGraph {  
    compilejob -> parsingjob;  
    parsingjob -> outputjob;  
}
```

c

```
++
#include <iostream>
using namespace std;
int main(){
    cout << "test_set_one" << endl;
}
```

2. test-code/test-set-two.cpp:

c

```
++
#include <iostream>
using namespace std;
void test_set_two(){
    cout << "test_set_two" << endl;
}
```

3. test-code/test-set-three.cpp:

c

```
++
#include <iostream>
using namespace std;
void test_set_three() {
    cout << "test_set_three" << endl;
}
```

Here Mistral Instruct wrote the flowscript correctly and did fix every .cpp file completely.

Mistral Instructs response should be written to .cpp files in the ./test-code directory. I have mocked up a basic version of that here in the else statement in the python LLMCall.py file:

```
import os #for checking if directory exists and creating it if it doesn't in else statement
```

```
else:
```

```
    print("No 'digraph JobGraph {' found in the response.")
```

```
    # Create the ./test-code/ directory if it doesn't exist
```

```
    if not os.path.exists("./test-code"):
```

```

os.makedirs("./test-code/")

# Specify the output .cpp file path
cpp_file_path = "./test-code/GeneratedCode.cpp"

# Write the response text to a .cpp file
with open(cpp_file_path, "w") as cpp_file:
    cpp_file.write(responseString)

```

This code throws the entire LLM response into a GeneratedCode.cpp file in the ./test-code/ directory, which is a start. Now only the corrected code portion should be thrown into associated .cpp files.

This new prompt makes sure that the LLM writes its output as a json file with needed info fields:

```

std::string fixErrorsPrompt = "Answer question based on the context. Context: Here is code in
a json file format with errors. Following this I have provided all test-set.cpp files that have the
errors to give you a full view of each file. I have also provided the associated .h files for each of
these .cpp files. Using the errors provided in the json file and each of these .cpp files, I want you
to fix the errors in each of these .cpp files. Each .cpp file can have multiple errors that need to
be fixed, sometimes not listed in the .json error file, so fix what you think needs to be fixed for
the code to run at your own discretion. Output each of these .cpp files in full, with all errors fixed.
Please also output the .cpp files associated .h files. These fixed .cpp files and their associated
.h files should be returned as neatly formatted strings that can be written to .cpp and .h files to
be ran later. Add a description of what you did to fix the errors. Please output your fixes in a json
file format like this format with a 'File Path: ' field, followed by a 'Fixed Code: ' field, and finally
ending with a 'Description: ' field that describes the fix to the code. Question: " +
errorsWithSourceCode;

```

Now that the Json file response format is guaranteed, that response can be parsed since the json file format is known in the LLMCall.py file:

```
else:
    print("Fixing code!") #TODO - fix this else statement to actually fix the code in the ./test-code/ directory. I have
    backed up the code in the ./test-code/ directory for security.

    # Parse the response as JSON
    try:
        response_data = json.loads(responseString)
    except json.JSONDecodeError:
        print("Error parsing JSON response.")
        response_data = None

    if response_data and "choices" in response_data:
        # Create the ./test-code/ directory if it doesn't exist
        if not os.path.exists("./test-code/"):
            os.makedirs("./test-code/")

        for item in response_data["choices"]:
            # Extract the "Fixed Code," "File Path," and "Description" from each item in the response
            fixed_code = item.get("Fixed Code", "")
            file_path = item.get("File Path", "")
            description = item.get("Description", "")

            if fixed_code and file_path:
                # Combine the ./test-code/ directory path with the file path
                full_file_path = os.path.join("./test-code/", file_path)

                # Write the extracted C++ code to the associated .cpp file
                with open(full_file_path, "w") as cpp_file:
                    cpp_file.write(fixed_code)

                # Optionally, write a description to a separate file (e.g., .txt) for reference
                description_file_path = full_file_path + ".txt"
                with open(description_file_path, "w") as desc_file:
                    desc_file.write(description)
```

```
else:  
    print("No valid JSON data found in the response.")
```

This code is a working draft of how it would be parsed and needs to be fixed to actually parse the response code blocks and write them into the .cpp files that are in the ./test-code/ directory.

Here I am testing that process with this hard coded response_data variable in the python script:

```
else:  
    print("Fixing code!")  
    # The JSON response data  
    response_data = [  
        {  
            "File Path": "test-code/test-set-one.cpp",  
            "Fixed Code": "int main(){  << \"test_set_one\" << std::endl;  //another_function();}",  
            "Description": "Added the missing expression after the cout statement and added a semicolon at the end of the  
line."  
        },  
        {  
            "File Path": "test-code/test-set-three.cpp",  
            "Fixed Code": "#include \"test-set-three.h\"\\nvoid test_set_three() {tstd::cout << \"test_set_three\" <<  
std::endl;\\t}",  
            "Description": "Added the missing semicolon after the #include directive and added curly braces around the cout  
statement."  
        },  
        {  
            "File Path": "test-code/test-set-two.cpp",  
            "Fixed Code": "#include <iostream> \\nvoid test_set_two(){  std::cout << \"test_set_two\" << std::endl;\\t}",  
            "Description": "Added the missing semicolon after the #include directive and added curly braces around the cout  
statement."  
        }  
    ]  
  
    # Process each item and write the fixed code to the corresponding files
```

```

for item in response_data:
    file_path = item["File Path"]
    fixed_code = item["Fixed Code"]

    # Ensure the directory exists
    directory = os.path.dirname(file_path)
    if not os.path.exists(directory):
        os.makedirs(directory)

    # Write the fixed code to the file
    with open(file_path, 'w') as file:
        file.write(fixed_code)

print("Updated files:", [item["File Path"] for item in response_data])

```

This hard coded `response_data` should be replaced with the LLM's response.

New prompt for LLM to make the LLM output the entire .cpp and .h files in a cleaner format:

```

std::string fixErrorsPrompt = "Answer question based on the context. Context: Here is code in
a json file format with errors. Following this I have provided all test-set.cpp files that have the
errors to give you a full view of each file. I have also provided the associated .h files for each of
these .cpp files. Using the errors provided in the json file and each of these .cpp files, I want you
to fix the errors in each of these .cpp files. In your response, output the entire .cpp files with all
errors fixed. Please also output the .cpp files associated .h files. These fixed .cpp files and their
associated .h files should be returned as neatly formatted strings that can be written to .cpp and
.h files to be ran later. Add a description of what you did to fix the errors. Please output your
fixes in a json file format like this format with a 'File Path: ' field, followed by a 'Fixed .cpp: ' field,

```

followed by a 'Fixed .h: ' field, and finally ending with a 'Description: ' field that describes the fix to the code. Question: " + errorsWithSourceCode;

Output from new prompt:

```
json
{
  "File Path": "test-code/test-set-one.cpp",
  "Fixed .cpp": "int main(){  << \"test_set_one\" << std::endl;  //another_function();}",
  "Description": "Added a semicolon after the cout statement and removed the extra space before the function call."
}

Here is the fixed .cpp file for test-code/test-set-three.cpp:

json
{
  "File Path": "test-code/test-set-three.cpp",
  "Fixed .cpp": "#include <iostream>  \nvoid test_set_three() {  std::cout << \"test_set_three\" << std::endl;}",
  "Description": "Added a semicolon after the include directive and removed extra spaces before the function call."
}

Here is the fixed .cpp file for test-code/test-set-two.cpp:

json
{
  "File Path": "test-code/test-set-two.cpp",
  "Fixed .cpp": "#include <iostream>  \nvoid test_set_two(){  std::cout << \"test_set_two\" << std::endl;}",
  "Description": "Added a semicolon after the include directive and removed extra spaces before the function call."
}
```

I have written this function in my LLMCall.py to parse the code fixes from the LLM:

```
def parseLLMFixes(response_text_from_LLM):
    response_text = response_text_from_LLM;

    # Find the delimiter in the text and extract the JSON formatted fixed code
    try:
        # The delimiter to start the JSON data for fixed code
        delimiter = "Here are the fixed .cpp files with their associated .h files:\n```\njson\n"

        # Find the starting index of the JSON data
        start_index = response_text.index(delimiter) + len(delimiter)
```



```
# Find the ending index of the JSON data (assuming it ends with triple backticks)
end_index = response_text.index("\n```", start_index)

# Extract the JSON part
json_data = response_text[start_index:end_index]

# Parse the JSON data
response_data = json.loads(json_data)

# Iterate through each item in the response data
for item in response_data:
    # Extract the file path and the fixed code
    file_path = item["File Path"]
    fixed_code = item["Fixed .cpp"] # Assuming the key in JSON is 'Fixed .cpp'

    # Ensure the directory exists
    directory = os.path.dirname(file_path)
    if not os.path.exists(directory):
        os.makedirs(directory)

    # Write the fixed code to the corresponding file
    with open(file_path, 'w') as file:
        file.write(fixed_code)

print("Files updated successfully!")
except ValueError as e:
    # Handle errors such as the delimiter not being found
    print("Failed to process the LLM response:", e)
except json.JSONDecodeError as e:
    # Handle JSON parsing errors
    print("Failed to parse JSON data:", e)
```

Ran into an issue with the delimiter line changing from “Here are the json files” to “Here is the json file” based on what the LLM decided to output the json as (either collated or separate), so I refined the prompt as follows:

```
std::string fixErrorsPrompt = "Answer question based on the context. Context: Here is code in a json file format with errors. Following this I have provided all test-set.cpp files that have the errors to give you a full view of each file. I have also provided the associated .h files for each of these .cpp files. Using the errors provided in the json file and each of these .cpp files, I want you to fix the errors in each of these .cpp files. In your response, output the entire .cpp files with all errors fixed. These fixed .cpp files should be returned as neatly formatted strings that can be written to .cpp files to be ran later. Add a description of what you did to fix the errors. Please output your fixes in a SINGLE json file like this format with a [ { 'File Path: ' field, followed by a 'Fixed .cpp: ' field, and finally ending with a 'Description: ' field } ], etc. that describes the fix to the code. Before you output the json file be sure to say 'Here are the fixed .cpp files:' Question: " + errorsWithSourceCode;
```

ignoring the .h files in this prompt and specifying that the LLM should output a SINGLE json file with the sentence before being “Here are the fixed .cpp files:”. I also specified the format of the json I would like. The output of this looks like:

```
[
  {
    "File Path": "test-code/test-set-one.cpp",
    "Fixed .cpp": "int main(){  << \"test_set_one\" << std::endl;
//another_function();}",
    "Description": "Added semicolons after the cout statement and removed extra whitespace."
  },
]
```

```

{
    "File Path": "test-code/test-set-three.cpp",
    "Fixed .cpp": "#include \"test-set-three.h\"\nvoid test_set_three()
{\nstd::cout << \"test_set_three\" << std::endl;\n}",
    "Description": "Removed extra whitespace and added semicolons
after the cout statement."
},
{
    "File Path": "test-code/test-set-two.cpp",
    "Fixed .cpp": "#include <iostream>\nvoid test_set_two(){
std::cout << \"test_set_two\" << std::endl;}",
    "Description": "Added semicolons after the cout statement and
removed extra whitespace."
}
}
]

```

Now to actually grab this single LLM response without my question preceding it I had three options, either:

1. modify the prompt given to the LLM
2. Parse the LLM's response to remove my preceding question
3. Simply subtract my prompt entered into the LLM from the LLM's returned text field

Method 1 and 2 proved overly complicated with lacking results, either not parsing correctly, or running into problems even finding the substring to parse. Method 3 can be looked at as a form of parsing, but all method 3 is doing is removing the prompt I am giving the LLM from the LLM's output. This method leaves the user with just the LLM response regardless of what question they ask because the question given to the LLM is always known and stays the same when the program gets to the LLMCall.py file. This method was done in python so it only required two extra lines to do:


```
[{"column": 2, "description": "invalid preprocessing directive", "file_path": "test-code/test-set-three.cpp", "line": 1, "source_code": "#include \"test-set-three.h\"", "column": 5, "description": "use of undeclared identifier 'std'", "file_path": "test-code/test-set-three.cpp", "line": 4, "source_code": "#include \"test-set-three.h\"\\nvoid test_set_three() {\\nstd::cout << \"test_set_three\" << std::endl;\\n}\\n"}, {"column": 38, "description": "use of undeclared identifier 'std'", "file_path": "test-code/test-set-three.cpp", "line": 4, "source_code": "#include \"test-set-three.h\"\\nvoid test_set_three() {\\nstd::cout << \"test_set_three\" << std::endl;\\n}\\n"}, {"column": 1, "description": "expected '}'", "file_path": "test-code/test-set-two.cpp", "line": 7, "source_code": "\\nstd::cout << \"test_set_two\" << std::endl;\\n}\\n"}, {"column": 20, "description": "to match this '{'", "file_path": "test-code/test-set-two.cpp", "line": 4, "source_code": "#include <iostream>\\nvoid test_set_two() {\\nstd::cout << \"test_set_two\" << std::endl;\\n}\\n"}], [{"file_path": "test-code/test-set-one.cpp", "fixed_cpp": "int main() { << \"test_set_one\" << std::endl; //another_function();}\\n", "description": "Added semicolons after the cout statement and removed extra whitespace."}, {"file_path": "test-code/test-set-three.cpp", "fixed_cpp": "#include \"test-set-three.h\"\\nvoid test_set_three() {\\nstd::cout << \"test_set_three\" << std::endl;\\n}\\n"}]
```

```

\"Description\": \"Removed extra whitespace and added semicolons after
the cout statement.\\n  },\\n  {\\n    \"File Path\": \"test-code/test-set-
two.cpp\",\\n    \"Fixed .cpp\": \"#include <iostream>\\nvoid
test_set_two(){  std::cout << \\\"test_set_two\\\" << std::endl;}\",\\n
\"Description\": \"Added semicolons after the cout statement and
removed extra whitespace.\\n  }\\n]\\n```
    }
],
    "created": 1701804830,
    "id": "foobarbaz",
    "model": "Mistral Instruct",
    "object": "text_completion",
    "usage": {
        "completion_tokens": 274,
        "prompt_tokens": 954,
        "total_tokens": 1228
    }
}

```

Here is method 3 being applied in python:

```

# Extract the LLM's response
response_text = response['choices'][0]['text']

# Remove the prompt from the response to get only the LLM's response
fixed_code_json_string = response_text.replace(prompt, "").strip()

print("\n\n\n RESPONSE WITH JUST FIXES: ", fixed_code_json_string, "\n\n\n")

```

Here is the output of method 3:

RESPONSE WITH JUST FIXES: Here are the fixed .cpp files:
```json

```
[
 {
 "File Path": "test-code/test-set-one.cpp",
 "Fixed .cpp": "int main(){ << \"test_set_one\" << std::endl;
//another_function();}",
 "Description": "Added semicolons after the cout statement and
removed extra whitespace."
 },
 {
 "File Path": "test-code/test-set-three.cpp",
 "Fixed .cpp": "#include \"test-set-three.h\"\\nvoid test_set_three()
{\\nstd::cout << \"test_set_three\" << std::endl;\\n}",
 "Description": "Removed extra whitespace and added semicolons
after the cout statement."
 },
 {
 "File Path": "test-code/test-set-two.cpp",
 "Fixed .cpp": "#include <iostream>\\nvoid test_set_two(){
std::cout << \"test_set_two\" << std::endl;}",
 "Description": "Added semicolons after the cout statement and
removed extra whitespace."
 }
]
```

```

This response is just the fixed code. Now the next step is to parse this code into code blocks with python. For this, a dictionary data structure can be used where the keys are the file paths and the values are the file contents:

```
formatted_code_blocks = parseLLMFixes(fixed_code_json_string) #fixed_code_json_string is
the response text from LLM with just the fixes. This should be parsed to get ready to write to
files.
```

parseLLMFixes is a function that takes the fixed_code_json_string (LLM code snippets that are fixed) and throws each .cpp's fixed code content into a dictionary based on filepath here:

```
def parseLLMFixes(fixed_code_json_string):
    try:
        # The delimiter to start the JSON data for fixed code
        delimiter = "Here are the fixed .cpp files:"
        json_data = fixed_code_json_string

        if delimiter in json_data:
            # Find the start of the actual JSON content
            json_start = json_data.find(delimiter) + len(delimiter)
            json_content = json_data[json_start:]

            # Find the start and end of the JSON data block enclosed in triple backticks
            json_data_start = json_content.find("```json") + len("```json")
            json_data_end = json_content.find("```", json_data_start)
            json_data = json_content[json_data_start:json_data_end].strip()
        else:
            raise ValueError("Delimiter not found in the response text.")

        # Parse the JSON data
        response_data = json.loads(json_data)

        # Dictionary to hold the formatted code blocks
        formatted_code_blocks = {}

        # Iterate through each item in the response data
```



```

for item in response_data:

    # Extract the file path and the fixed code
    file_path = item["File Path"]
    fixed_code = item["Fixed .cpp"]

    # Add the fixed code to the dictionary
    formatted_code_blocks[file_path] = fixed_code

print("Formatted code blocks parsed successfully!")
return formatted_code_blocks

except ValueError as e:

    # Handle errors such as the delimiter not being found
    print("Failed to process the LLM response:", e)
    return None

except json.JSONDecodeError as e:

    # Handle JSON parsing errors
    print("Failed to parse JSON data:", e)
    return None

```

Now back in the main python code, the formatted_code_blocks dictionary content can be printed out like this:

```

formatted_code_blocks = parseLLMFixes(fixed_code_json_string) #fixed_code_json_string is the
response text from LLM with just the fixes. This should be parsed to get ready to write to files.

# Check if formatted_code_blocks is not None
if formatted_code_blocks is not None:

```

```

# Iterate through the dictionary and print its contents
for file_path, code_block in formatted_code_blocks.items():
    print(f"File Path: {file_path}")
    print("Fixed C++ Code:")
    print(code_block)
    print("\n" + "-"*60 + "\n") # Just to separate each code block for readability
else:
    print("No formatted code blocks were parsed.")

```

The console output looks like this:

Formatted code blocks parsed successfully!

File Path: test-code/test-set-one.cpp

Fixed C++ Code:

```
int main(){  << "test_set_one" << std::endl;  //another_function();}
```

File Path: test-code/test-set-three.cpp

Fixed C++ Code:

```
#include "test-set-three.h"
```

```
void test_set_three() { std::cout << "test_set_three" << std::endl;
}
```

File Path: test-code/test-set-two.cpp

Fixed C++ Code:

```
#include <iostream>
```

```
void test_set_two(){  std::cout << "test_set_two" << std::endl;      }
```

This parseLLMFixes function essentially associates the fixes with a file path, so now the fixes can be written to the ./test-code/ directory.

Sometimes this function cannot write content to formatted_code_blocks because the function didn't know what to do when it ran into '_' notation, so this python function called 'correct_invalid_escapes(json_string):' replaces any '_' characters with '_':

```
import re #for fixing invalid escape characters in json string with correct_invalid_escapes()
function

def correct_invalid_escapes(json_string):
    # Replace invalid escape sequences like '\\_' with '_'
    corrected_string = re.sub(r'\\_', '_', json_string)
    return corrected_string
```

This correct_invalid_escapes(json_string) is called here right before the parseLLMFixes function here:

```
#check for invalid escape characters
fixed_code_json_string = correct_invalid_escapes(fixed_code_json_string)

# Parse the LLM's response to get the fixed code blocks
formatted_code_blocks = parseLLMFixes(fixed_code_json_string) #fixed_code_json_string is
the response text from LLM with just the fixes. This should be parsed to get ready to write to
files.
```

Now the code blocks must be written into the appropriate .cpp files in the ./test-code/ directory. To start the writeToFiles function can be stubbed in with print statements:

```
writeToFiles(formatted_code_blocks) #TODO - write to files.
```

```
def writeToFiles(formatted_code_blocks):
```

```

print("\nWriting fixed code to appropriate files!\n")

# Check if formatted_code_blocks is not None
if formatted_code_blocks is not None:

    # Iterate through the dictionary and print its contents
    for file_path, code_block in formatted_code_blocks.items():
        print(f"File Path: {file_path}")
        print("Fixed C++ Code:")
        print(code_block)

        print("\n" + "-"*60 + "\n") # Just to separate each code block for readability
    else:
        print("No formatted code blocks were parsed.")

```

Now this function should be modified to actually write to the associated files using the key as the means to grab the file contents and then writing to that same key (the key is the file_path associated to each file's contents):

Before I test this function I want to build a reloadErrors() function in main.cpp to easily repopulate my errors in my ./test-code/ directory from my ./test-code-backup directory here:

```

void reloadErrors(){//this function repopulates the ./test-code directory with the backup error files
from the ./test-code=backups directory

    // Array of file names that need to be reset
    std::array<std::string, 3> file_names = {"test-set-one.cpp", "test-set-three.cpp", "test-set-
two.cpp"};

    // Iterate through the file names and copy each from the backup directory to the test-code
directory
    for (const std::string& file_name : file_names) {

```

```

std::string command = "cp ./test-code-backups/" + file_name + " ./test-code/" + file_name;
// Execute the copy command
int result = system(command.c_str());
// Check if the command was successful
if (result != 0) {
    std::cerr << "Failed to copy " << file_name << std::endl;
} else {
    std::cout << "Successfully reset errors for " << file_name << std::endl;
}
}
}

```

On mac this function uses the “cp” command in a string to copy contents from one directory to another, but if you are on windows this will have to be changed to use “copy”.

Now that the reload function works, the code in ./test-code/ directory can safely be written to by the LLMCall.py script:

```

def writeToFiles(formatted_code_blocks):
    print("\nWriting fixed code to appropriate files!\n")
    # Check if formatted_code_blocks is not None
    if formatted_code_blocks:
        # Iterate through the dictionary and write its contents to files
        for file_path, code_block in formatted_code_blocks.items():
            # Ensure the directory exists
            directory = os.path.dirname(file_path)
            if not os.path.exists(directory):
                os.makedirs(directory)

```

```

# Write the fixed code to the corresponding file
try:
    with open(file_path, 'w') as file:
        file.write(code_block)
        print(f"Successfully wrote to {file_path}")
except IOError as e:
    print(f"Failed to write to {file_path}: {e}")

else:
    print("No formatted code blocks were parsed.")

```

The writeToFile function has been modified to write each fixed code snippet to their corresponding .cpp file in the ./test-code/ directory.

Output looks like this:

Test-set-one.cpp:

```

int main(){ << "test_set_one" << std::endl; //another_function();}

```

Test-set-two.cpp:

```

#include <iostream>
void test_set_two(){
std::cout << "test_set_two" << std::endl; }

```

Test-set-three.cpp:

```

#include "test-set-three.h"
void test_set_three() {
std::cout << "test_set_three" << std::endl;
}

```

This is the corrected code written to all appropriate .cpp files in the ./test-code/ directory.

The code in main.cpp should be wrapped in a while loop with the break condition being that compilejob produces no compile errors. This break condition can either mean that the compileJobOutput.json is read as null and then the while loop is broken or that the compilejob thread returns an alert when there are no errors that tell the while loop to break. Since the main.cpp will be wrapped in a while loop and go through the code multiple times to fix all errors, then the LLM can afford to be slightly off each time, and each time it will get closer to the answer until it solves the problem. Here is the loop in practice:

In outputjob.cpp there is a condition that assigns the output.json this line if no errors are found “No errors or warnings – compiled successfully”. This line can be used as a break statement in the while loop in main.cpp. Here is the line in outputjob.cpp:

```
//if errors is empty, then write "No errors/warnings" to output.json
if (errors.empty()) {
    json outputJson = json::array();
    json errorJson;
    errorJson["Error Description"] = "No errors or warnings - compiled successfully";
    outputJson.push_back(errorJson);

    std::ofstream outputFile("Data/output.json");
    outputFile << outputJson.dump(4); // 4 spaces for pretty printing
    outputFile.close();

    // //delete errors.json file
    //std::remove("Data/errors.json");
```

```
return;  
}
```

In main.cpp create the while loop:

```
while(true){//while loop that breaks when the output.json file reads "No errors or warnings -  
compiled successfully"
```

... rest of code

```
//read in output.json file and check if it says "No errors found!"  
  
// Open the file  
std::ifstream file("./Data/output.json");  
if (!file.is_open()) {  
    std::cerr << "Could not open the file 'output.json'" << std::endl;  
    return 1;  
}  
  
// Read the file into a string  
std::stringstream buffer;  
buffer << file.rdbuf();  
std::string fileContent = buffer.str();  
  
// Check if the file contains "No errors found!"  
if (fileContent.find("No errors or warnings - compiled successfully") != std::string::npos) {  
    std::cout << "No errors found!" << std::endl;  
    break; //exit the while loop if no errors are found  
} else {  
  
    //continue the while loop if errors are found  
    std::cout << "Errors were found or the specific message is not present." << std::endl;  
}
```



```
// Close the file  
file.close();
```

Now the LLM can be given multiple queries with error lines in attempts to fix the broken code.....

The LLM can fix every test-code file except test-set-one.cpp because it has multiple errors in the file as you can see here:

```
#include "test-set-one.h"  
#include <iostream>  
  
void test_set_one(){  
    << "test_set_one" << std::endl  
    //another_function();  
}
```

The LLM can get the file this far:

```
void test_set_one(){ std::cout << "test_set_one" << std::endl;}
```

but fails to include the `#include <iostream>`. This can be fixed by re-engineering the `fixErrorsPrompt` to specify that necessary headers like `#include <iostream>` should be included in the code files:

current prompt that does not specify:

```
std::string fixErrorsPrompt = "Answer question based on the context. Context: Here is code in a json file format with errors. Following this I have provided all test-set.cpp files that have the errors to give you a full view of each file. I have also provided the associated .h files for each of these .cpp files. Using the errors provided in the json file and each of these .cpp files, I want you to fix the errors in each of these .cpp files. In your response, output the entire .cpp files with all errors fixed. These fixed .cpp files should be returned as neatly formatted strings that can be written to .cpp files to be ran later. Add a description of what you did to fix the errors. Please output your fixes in a SINGLE json file like this format with a [ { 'File Path: ' field, followed by a 'Fixed .cpp: ' field, and finally ending with a 'Description: ' field } ], etc. that describes the fix to the code. Before you output the json file be sure to say 'Here are the fixed .cpp files:' Question: " + errorsWithSourceCode;
```

New prompt that specifies that necessary headers like iostream should be included:

```
std::string fixErrorsPrompt = "Answer question based on the context. Context: Here is code in a json file format with errors. Following this I have provided all test-set.cpp files that have the errors to give you a full view of each file. I have also provided the associated .h files for each of these .cpp files. Using the errors provided in the json file and each of these .cpp files, I want you to fix the errors in each of these .cpp files. In your response, output the entire .cpp files with all errors fixed. These fixed .cpp files should be returned as neatly formatted strings that can be written to .cpp files to be ran later. Add a description of what you did to fix the errors. Please output your fixes in a SINGLE json file like this format with a [ { 'File Path: ' field, followed by a 'Fixed .cpp: ' field, and finally ending with a 'Description: ' field } ], etc. that describes the fix to the code. Be sure to include necessary header files like '#include <iostream>' in your fixes. Before you output the json file be sure to say 'Here are the fixed .cpp files:' Question: " + errorsWithSourceCode;
```

The new output looks like this for each code file:

1. Test-set-one.cpp unfixed:

```
2. #include "test-set-one.h"
3. #include <iostream>
4.
5. void test_set_one(){
6.     << "test_set_one" << std::endl
7.     //another_function();
8.
9. }
```

10. Test-set-one.cpp fixed:

```
#include <iostream>
```

11. Test-set-two.cpp unfixed:

```
12. #include "test-set-two.h"
13. #include <iostream>
14.
15. void test_set_two(){
16.
17.     std::cout << "test_set_two" << std::endl;
18. }
```

19. Test-set-two.cpp fixed:

```
#include <iostream>
```

20. Test-set-three.cpp unfixed:

```
21. #incl
```

```
22. #include "test-set-three.h"
23. void test_set_three() {
24.     std::cout << "test_set_three" << std::endl;
25. }
26.
```

27. Test-set-three.cpp fixed:

```
#include <iostream>
```

The new prompt erases all code and only includes iostream headers. This is interesting behavior and has to be resolved with further prompt engineering. The prompt should now be given a disclaimer that the ‘#include <iostream>’ should not be placed in a file to replace other code, but should only be added to the first line of the file if it is not there to begin with. Here is the new prompt:

```
std::string fixErrorsPrompt = "Answer question based on the context. Context: Here is code in a json file format with errors. Following this I have provided all test-set.cpp files that have the errors to give you a full view of each file. I have also provided the associated .h files for each of these .cpp files. Using the errors provided in the json file and each of these .cpp files, I want you to fix the errors in each of these .cpp files. In your response, output the entire .cpp files with all errors fixed. These fixed .cpp files should be returned as neatly formatted strings that can be written to .cpp files to be ran later. Add a description of what you did to fix the errors. Please output your fixes in a SINGLE json file like this format with a [ { 'File Path: ' field, followed by a 'Fixed .cpp: ' field, and finally ending with a 'Description: ' field } ], etc. that describes the fix to the code. Be sure to include necessary header files like '#include <iostream>' in your fixes, but please do not erase the rest of the code that was there before with these includes, just add these includes to the top of the file if they are not there to begin with as additions. YOU SHOULD NOT ERASE ANY CODE FROM THE FILES, JUST ADD FIXES TO THE FILES. Before you output the json file be sure to say 'Here are the fixed .cpp files:' Question: " + errorsWithSourceCode;
```

This new prompt is too complicated and makes the LLM obsess too much over header includes. What I did instead was add my missing `std::cout` error in my `test-code-one.cpp` file so that the only error the LLM had to fix in that file was the missing `;` error. The LLM struggles to fix multiple errors in one file because of my logging method in the `output.json` and cannot detect missing `std::cout` statements. Here is the run through with a simpler prompt and only one error in `test-set-one.cpp`:

I was failing to include the full reference file in the `test-set-one.cpp` file so the LLM was not writing that full file on fixing, here that is fixed. Here is a new `processFile` function that supports `.cpp/.h` files to ensure that all my `.cpp` files that are read have their headers included in the source code example provided to the LLM so that the LLM has the full code to write back to the files once fixed:

```
std::string processFile(const std::string& filePath) {
    std::string file_content = readFileContents(filePath);

    if (filePath.size() >= 5 && filePath.substr(filePath.size() - 5) == ".json") {
        // Parse the JSON data
        json json_data = json::parse(file_content);
        // Serialize JSON object to string
        return json_data.dump(4); // '4' for pretty printing
    } else if (filePath.size() >= 4 && (filePath.substr(filePath.size() - 4) == ".txt" ||
filePath.substr(filePath.size() - 4) == ".cpp" || filePath.substr(filePath.size() - 2) == ".h")) {
        // For .txt, .cpp, and .h files, just read and return the file content
        return file_content;
    } else {
```

```
std::cerr << "Unsupported file type for file: " << filePath << std::endl;
return "";
}
}
```

The .cpp and .h extensions were added to the else if that returns the entire unmodified file contents here:

```
} else if (filePath.size() >= 4 && (filePath.substr(filePath.size() - 4) == ".txt" ||
filePath.substr(filePath.size() - 4) == ".cpp" || filePath.substr(filePath.size() - 2) == ".h")) {
    // For .txt, .cpp, and .h files, just read and return the file content
    return file_content;
}
```

so now the .cpp and .h files are supported along with the ability to serialize the .json files.

The prompt had to be simplified and the LLM had to be told to `#include <iostream>` at the start of its ‘Fixed .cpp:’ field output. LLM model had to explicitly be told to refer to and use the source code at the very bottom of the input provided. This is what made the biggest difference in the output of the LLM including all headers and full file format as before etc. Now the LLM can fix multiple errors in the test-set-one.cpp file because of the prompt refinements.

Prompt:

```
std::string fixErrorsPrompt = "Answer question based on the context. Context: Here is code with errors in a json file format. This json file format does not include all the errors so please refer to the full source files that are at the end of this provided prompt. Fix the errors in each of these .cpp files and format your output on the full source code files at the end of this input. In your response, output the entire .cpp files with all errors fixed. These fixed .cpp files should be returned as neatly formatted strings that can be written to .cpp files to be ran later. Add a
```

description of what you did to fix the errors. Please output your fixes in a SINGLE json file like this format with a [{ 'File Path: ' field, followed by a 'Fixed .cpp: ' field, and finally ending with a 'Description: ' field }]. Please make sure that every 'Fixed .cpp:' field starts with an #include <iostream> header'. Before you output the json file be sure to say 'Here are the fixed .cpp files:'
Question: " + errorsWithSourceCode;

28. Test-set-one.cpp unfixed:

```
29. #include "test-set-one.h"  
30. #include <iostream>  
31.  
32. void test_set_one(){  
33.     << "test_set_one" << std::endl  
34.     //another_function();  
35.  
36. }
```

37. Test-set-one.cpp fixed:

```
38. #include <iostream>  
39.  
40. void test_set_one(){  
41.     std::cout << "test_set_one" << std::endl;  
42. }
```

43. Test-set-two.cpp unfixed:

```
44. #include "test-set-two.h"  
45. #include <iostream>  
46.  
47. void test_set_two(){  
48.  
49.     std::cout << "test_set_two" << std::endl;  
50.
```

51. Test-set-two.cpp fixed:

```
52. #include <iostream>
53.
54. void test_set_two(){
55.     std::cout << "test_set_two" << std::endl;
56. }
```

57. Test-set-three.cpp unfixed:

```
58. #incl
59. #include "test-set-three.h"
60. void test_set_three() {
61.     std::cout << "test_set_three" << std::endl;
62. }
63.
```

64. Test-set-three.cpp fixed:

```
#include <iostream>

void test_set_three() {
    std::cout << "test_set_three" << std::endl;
}
```

The last modification was to update the prompt to fix the coding errors with the added requirement to include all the '.h' files in their associated .cpp files:

```
std::string fixErrorsPrompt = "Answer question based on the context. Context: Here is code with errors in a json file format. This json file format does not include all the errors so please
```


refer to the full source files that are at the end of this provided prompt. Fix the errors in each of these .cpp files and format your output on the full source code files at the end of this input. In your response, output the entire .cpp files with all errors fixed. These fixed .cpp files should be returned as neatly formatted strings that can be written to .cpp files to be ran later. Add a description of what you did to fix the errors. Please output your fixes in a SINGLE json file like this format with a [{ 'File Path: ' field, followed by a 'Fixed .cpp: ' field, and finally ending with a 'Description: ' field }]. Please make sure that every 'Fixed .cpp:' field starts with an #include <iostream> header'. Make sure to include any .h files that are associated with the .cpp files in your 'Fixed .cpp: ' output field. Before you output the json file be sure to say 'Here are the fixed .cpp files:' Question: " + errorsWithSourceCode;

refer to video demonstration for final format of fixed test-code/ directory code files.