CLEMSON CAPSTONE MICHELIN PROPOSAL

Final Report

AJ Garner, Jacob Cox, Troy Butler,

Caroline Baker, Carson Crockett

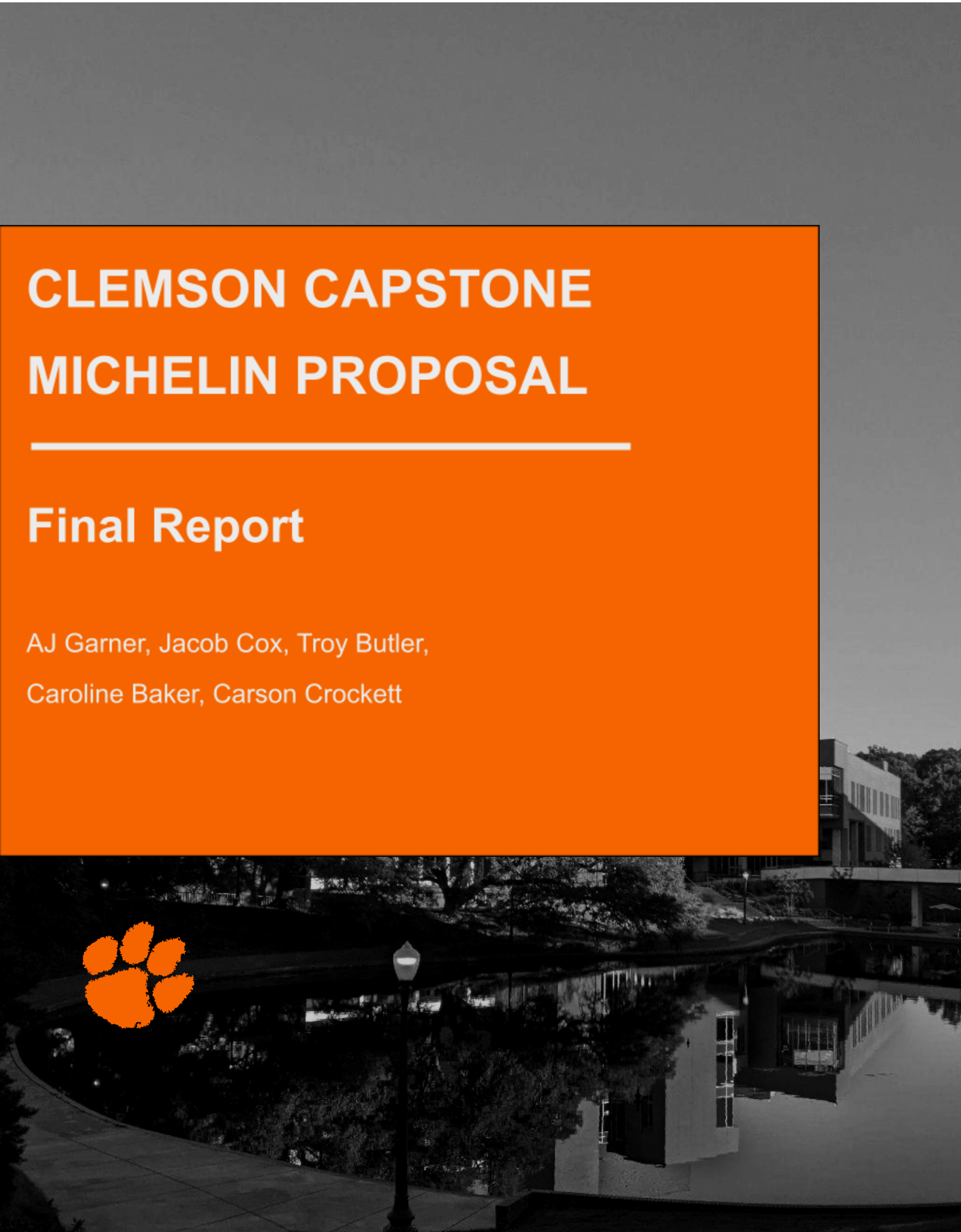

Table of Contents

Team Summary

- AJ Garner - Team Lead
- Jacob Cox - RAG Pipeline Developer
- Carson Crockett - Semantic Analysis Developer
- Troy Butler - Expert Relations and Research
- Caroline Baker - UI Design and Research

Introduction & Background

Michelin is a global leader in tire manufacturing and is renowned for its innovation, quality, and commitment to sustainability. It is a global company with operations in over 170 countries and a wide range of markets. As an established business, Michelin is searching for a way to accelerate the migration away from legacy IT infrastructure. Modern generative AI technology presents a solution by helping developers rapidly understand and answer questions about the legacy system during the migration process.

Aging legacy systems are an industry-wide problem: these systems can be insecure, outdated, or otherwise difficult to maintain, making it an attractive move to replace them with modern technology. However, due to the heterogeneous nature of these old systems, there is no single streamlined way to move to newer technologies, and there is no single target system that legacy migrations move toward. Using current practices, migrating away from legacy systems is an expensive, time-consuming process that requires a large amount of manual labor from company developers and IT staff. This means there is significant value in finding ways to automate some or all of this process.

An important consideration in legacy migration is the conservation of business rules: behavior encoded in the logic of the legacy system that represents functional “whats and whens” of the business’s operations. For example, suppose business X has a long-standing relationship with customer A. As a result, business X has a special agreement with customer A that they don’t offer to other customers, so this logic would be encoded in the relevant legacy system as a special rule for only customer A. When many of these rules accumulate within legacy systems over decades such that they become deeply intertwined with the business’s IT operations, care must be taken when replacing the system to prevent some rules from being excluded from the new system. To avoid damaging customer relationships, preserving these rules is especially important for outward-facing systems that handle invoicing and ordering.

This project explores how modern generative AI technologies such as large language models (LLMs) can assist the legacy migration process by automating the extraction of business rules from legacy systems with minimal manual intervention. This would save time and resources for companies and facilitate a more reliable migration process than traditional techniques alone.

Project Scope

To meet this need, we propose a modern, automated solution that leverages generative AI technology to streamline the legacy system migration process. Our system lets users query a customized AI assistant designed to address queries from both technical and non-technical users on the business logic of the legacy system and other relevant code and documentation

files provided by the user. This assistant helps businesses and employees manage the time, cost, and complexity of legacy migration by leveraging the massive information processing capabilities of modern LLMs. We will accomplish this by creating a retrieval-augmented generation (RAG) system capable of dynamically searching the legacy system's source code and documentation to answer user queries during the migration process. Using Clemson's high-performance computing cluster, we describe and provide an implementation of such a system and discuss avenues for further development of this system by future teams.

We expect this proposal to span multiple semesters before reaching completion. Michelin is also working with other prominent industry partners on a solution for this problem. For this semester, our focus has been exploring different approaches to the problem while fostering a collaborative relationship between Michelin and Clemson for future projects. Further to our goal of exploring new technologies and solutions, our project includes the development of a prototype system and a modular framework that enables future iteration and refinement. The remainder of this document discusses the research and design considerations that have informed the development of this prototype, addresses the implementation of the prototype itself, and then discusses directions and recommendations for future iterations of this project.

Semester Schedule

Sprint	Dates	Description	Milestone
0	**9/8-9/14**	Project summary, scope, and team definition	**Preliminary research and scope assessment**
1	**9/15-9/21**	Backlog and sprint planning and initial partner meeting	**Solution design and timeline planning**
2	**9/22-9/28**	Roadmap development and project planning	**Delivery plan and roadmap finalization**
3	**9/29-10/5**	Partner coordination and access, final product design and proposal	**Technology selection and midterm solution proposal**
4	**10/6-10/12**	Progress report and early development, Palmetto access	**Technology access and begin development**
5	**10/13-10/19**	RAG pipeline and virtual environment setup	**Proof-of-concept RAG system and environment documentation**
6	**10/20-10/26**	RAG improvements, faculty meetings, continued development	**Connecting with faculty consultants and progressing with RAG**
7	**10/27-11/2**	Adding to the pipeline and working with code examples	**Specialized LLM pipelines for final design sub-tasks**

8	**11/3-11/9**	Generating and interpreting COBOL and implementing a database, Second LLM integration	**Pipeline refinement and benchmarking material development**
9	**11/17-11/23**	Code examples and outputs, pipeline improvements, and begin benchmarking	**Performance assessment, accuracy testing, and system refinement**
10	**11/24-12/7**	Final fine-tuning and bug-fixing	**Design finalization feature-freeze and final report conclusion**
–	**12/13**	Final presentation to industry partners	**Organize presenting material and final demo for partner and stakeholders**

Project Discovery

Understanding the Problem

The first stage of this project has centered around understanding the problem and how it affects the wider industry as well as Michelin specifically. Indeed, the day-to-day operations of many large companies are deeply integrated with complex legacy systems that were designed and implemented many years or potentially decades ago. In that time, these systems have been modified and amended by different teams to meet evolving business needs. As a result, the programming logic of these old systems often encodes valuable business logic and other information necessary for standard business functions. In Michelin's case, a mid-1970s legacy COBOL system plays a crucial role in the company's information flow.

Although approximately 80% of this system's business logic has already been documented for replication with newer technology, finding and documenting the remaining 20% would take a significant amount of time and effort to extract manually. The resource requirements for migration are especially high in this case because finding the required talent to understand outdated or uncommon languages such as COBOL can be challenging, creating an additional barrier to migration.

Determining a Solution

In light of these considerations, Michelin is looking for an AI-based assistant that can accelerate migration tasks by making it easier for the technical team performing the migration to identify and understand the nuances of the legacy system's behavior. Additionally, this AI assistant should be accessible to non-technical employees, meaning some users may be IT engineers while others may be accountants who need general information about certain details of the business's operations. The assistant will take plain-text user queries and answer them using relevant information extracted from the legacy system's implementation files. To maximize the relevance and accuracy of the assistant's answers, the assistant should not answer with any

outside information that does not apply strictly to Michelin's use case. Additionally, the finished system needs to safely handle proprietary or otherwise sensitive information specific to Michelin's legacy system as part of the answer generation process.

Initial Research

Learning COBOL

Given that the subject of this Project is Michelin's COBOL-based legacy mainframe ERP system, a rudimentary understanding of COBOL is beneficial for contributing to this project. However, a thorough understanding of the language is not necessary. We have had success using AI tools like ChatGPT and other online resources as provisional guides for understanding the finer points of COBOL syntax as the need arises. Using these techniques will allow quick knowledge acquisition and comprehension of the basics. This should empower team members to look at the existing code from Michelin and make it easy to work with. Additionally, the system's functionality should remain language-agnostic, meaning the development and testing of the project can be done using a more common language such as C or Java. Then, once the system's general performance is satisfactory, options such as few-shot learning and model fine-tuning can be explored to improve the system's aptitude for COBOL specifically. Before exploring such options, however, methods should be devised to objectively assess the system's performance such that performance regressions and improvements are readily identifiable. Few-shot learning, fine-tuning, and performance measures are discussed in the Recommendations for Future Research section.

Faculty Advice

In light of the unprecedented nature of this project, the expertise available from Clemson University faculty is an especially valuable resource. To date, we have interviewed four faculty members for their advice and insights regarding this project. While they are not affiliated or otherwise obligated to participate in this project, they have each expressed interest in learning more about the project's progress and are willing to meet with future teams to offer advice as needed. The faculty members we've interviewed for this project include:

Dr. Carl Ehrett
Dr. Ehrett is the Director of Applied Machine Learning at the Watt Family Innovation Center and is very knowledgeable on the subjects of machine learning and LLM implementation.

Dr. Paige Rodeghero
Dr. Rodeghero is an Assistant Professor in the School of Computing. She has industry experience as a software engineer and has co-authored numerous peer-reviewed articles about developer productivity.

Professor Dan Wooster
Professor Wooster is a Professor of Practice in the School of Computing. He has decades of industry experience, including some with COBOL, and he is enthusiastic about the use of AI to solve problems.

Dr. Russell Purvis
Dr. Purvis is a Professor in the College of Business. He has offered insights from a business perspective and has provided good advice on the development of use cases.

Reports documenting the initial interviews with these faculty members can be found in the project repository on GitHub.

Current Solution Architecture

RAG Introduction

RAG is an emerging generative AI technique designed to improve the factual accuracy and depth of answers generated by LLMs. Using this approach allows us to specialize pre-trained models to answer questions about topics not present in the dataset that was used to train the model. It does this by using a search engine-like system to retrieve relevant facts from a database of custom information. The LLM is instructed to only answer using information from the database. Using this approach to supply an LLM with information about the legacy system, RAG becomes a suitable method for creating a user-friendly application capable of analyzing the legacy system to accurately answer plain-language questions about its business rules and functionality. For this project, we've chosen RAG as the most suitable method for meeting the requirements of this system. This section of the document describes the organization of an implemented prototype, identifying and explaining the purpose of each component in the prototype RAG pipeline. A strong avenue for future iterations of this project is to identify how this prototype can be extended and made more suitable for this application.

Our proposed solution is composed of several components that work together to achieve the desired functionality by implementing a RAG pipeline designed to extract business rules from the legacy system as well as answer other targeted technical questions about its behavior. These components include the elements of a traditional RAG pipeline as well as extensions necessary for our specific use case. The below diagram shows the overall architecture of our existing RAG pipeline and each of its constituent components at the conclusion of this semester.

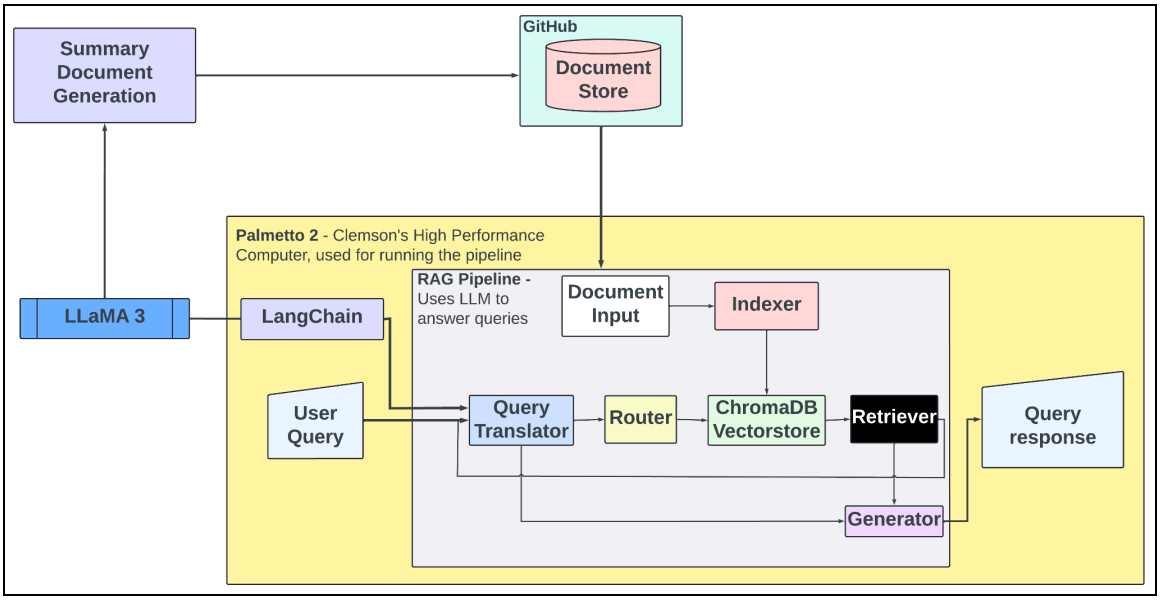

Illustration of proposed system design

The rest of this section is dedicated to explaining the components listed in the above system architecture diagram.

Summary Documentation Generation: This component describes a custom AI agent we developed to generate supplementary documentation to the COBOL code included in the legacy system. This serves to provide more relevant context to the LLM running the RAG pipeline so it does not have to infer it at the same time it is answering queries.

Query Translator: The query translator is responsible for rewriting, iterating, and modifying the abstraction level of user-provided queries. This is necessary because user queries can be ambiguous, leading to inconsistent context retrieval performance. At present, our query translator uses the same Llama3 model as the generator. Since the vector store contains mostly code, retrieval will only be successful if done with code. Therefore, we prompt Llama to convert user-provided questions like "What does function foo do in this program?" into example code like "`int foo (void) { … return bar; }`" to increase the likelihood of the retriever returning the correct parts of the codebase.

Router: Routers direct queries to the appropriate vector store. In cases where there is more than one type of data in need of retrieval, it is necessary to determine to which data store each query should be directed to find relevant information to provide to the generator. Often, another LLM is given information about the types of data available and asked to decide which data source to use. In the current solution, there is no need for routing because all documents come from a single source and are handled by the same retriever. Translated queries are piped directly to the retriever to find relevant context.

Vector Store: Vector stores are used to store the RAG documents. We will use vector stores to convert unstructured information like documentation and source code to their

respective numeric representations, to be stored for retrieval by comparing them with queries for documents with similar information. Queries are embedded into the same space as the stored context where the retriever does a nearest-neighbor-like search to find the parts of the stored context most relevant to the semantic meaning of the query. We are presently using ChromaDB as the vector store; replacing it with other options is straightforward with LangChain's abstractions.

Indexer: The indexer is responsible for storing documents for future retrieval. RAG relies upon comparing queries with the provided documents. Indexing is the process of converting the documents to a form where semantic comparisons can easily be made between the documents and queries during retrieval. Our current model handles indexing using ChromaDB's built-in indexing functionality.

Retriever: The retriever is responsible for fetching and assessing the quality and relevance of retrieved information. The core of RAG is the ability to look up relevant pieces of the provided documents as context for the LLM. The retriever performs similarity searches between queries and embedded documents to provide relevant context to the generator. At present, the system only uses a single retriever to pull context from a single data source (a codebase). In the future, separate retrievers should be used for different data sources. For example, one retriever should handle the source code, another should handle documentation, etc. This allows tailoring each retriever toward the type of information it handles. Retrieving is currently handled by ChromaDB.

Generator: The generator is the component of the RAG pipeline responsible for using the LLM to generate answers based on the provided query and the retrieved context. The system uses a locally hosted Llama3 model acquired from HuggingFace as the generator. The original user question and the context from ChromaDB go into the generator and combine with the generator's parametric knowledge to produce a final answer to the question.

Frameworks, Tools, and Technology

This section introduces and briefly describes the technologies used to implement the RAG-based application outlined in the above diagram. In some cases, it also highlights promising avenues for future technologies to incorporate into the existing system.

LangChain: A framework for creating applications that integrate with large language models. LangChain provides integrations with popular large language models and provides tooling to easily integrate these models with other ancillary components needed to build more complex LLM-based applications including most or all of the components necessary to implement a RAG pipeline. LangChain supports both Python and JavaScript – we will use Python. Access to most of the components in our system will be through LangChain's interfaces.

LangSmith: LangSmith is a companion call-tracing and debugging platform featuring tight integration with the LangChain framework. We recommend using LangSmith's integration features to test, debug, and monitor the behavior of our application during the development process to make understanding and debugging the system easier.

Llama3: The core of a RAG pipeline is a large language model that produces answers based on the query and the retrieved context. We are using Meta's Llama3 as our primary LLM because it is open-source, comparable to popular proprietary models, and can be run locally. These factors are favorable because they provide more freedom to make necessary adjustments. In our existing implementation, we primarily use meta-llama/Llama-3.1-8B Instruct and meta-llama/Llama-3.2-3B Instruct because they offer a good balance between performance and resource usage.

ChromaDB: RAG relies on decomposing reference information into easily retrievable chunks to insert into a generator's context window. ChromaDB is a popular open-source vector database for embedding documents for retrieval in a RAG pipeline. Our decision to work with it stems from it being open source, well-supported within LangChain, and commonly used for our use case. LangChain provides integration for ChromaDB which makes it easy to incorporate it into a Python script.

Palmetto: Compute-intensive applications like those running one or more LLMs benefit from higher resource availability. Palmetto is Clemson University's high-performance computing platform. Many nodes are equipped with NVIDIA Tesla GPUs, enabling us to work with more sophisticated models and higher parameter counts, increasing application performance and output reliability. The compute resources available on Palmetto permit much larger models than 3B and 8B Llama models, but we choose to test with smaller models for their faster load times.

Multi-Query Translation: The performance of a RAG application depends on the retriever's ability to find documents relevant to the user's query in order to provide context to the generator to produce an answer. Since queries are written by users, multi-query translation is a technique for improving answer consistency. This is accomplished by using an LLM to write multiple versions of the original query to reduce ambiguity. Retrieval is done with each rewritten question, and the total retrieved documentation is combined when provided to the generator. At present, our implementation does not support multi-query translation. In future, retrieval could be made more reliable by using this technique to normalize the translator's behavior: user queries can be translated into multiple code snippets, retrieval can be done with each snippet, and the unique set of context from all retrievals is given to the generator to help us answer the original query.

Logical Routing: In cases where a RAG application includes multiple data stores, it becomes necessary to determine to which data store each question should be routed based on its content. This is often done by prompting an LLM with appropriate context to

route the question. Since this application retrieves information from system documentation and source code alike, it is necessary to have separate data stores and route questions accordingly. Should multiple retrievers and data sources be added to the system, this is an appropriate approach for routing queries between the different data sources.

Prototyping and Proof of Concept

After researching and arriving at the above-described system architecture for our RAG pipeline, we completed the below steps to arrive at a working prototype model capable of answering questions about specific codebases and other custom bodies of information.

RAG Pipeline: Our initial RAG pipeline was a barebones attempt made simply to be functional for testing and comparison. We used the video "RAG from Scratch," which explains the basics of RAG and how to build such a pipeline. As time went on, we updated the LLM from running a local Ollama server to hosting and connecting to the model via HuggingFace. This allowed us to use different versions of Llama which were trained with more or fewer parameters, increasing performance. From here we began implementing modularizations that we developed.

Query Translation: We developed a custom agent to assist in the query translation stage of our RAG pipeline. This worked to assist the model in comparing plaintext queries to code by converting the query to code and comparing it to documents in our data store. We then implemented this feature into our RAG pipeline so it is done seamlessly when users send queries to the pipeline.

Summary Generation: We also built an agent to generate more plain-text documentation for the RAG pipeline to use. Due to its current runtime, it needs to be run on the documents and have its output manually included in the document store. It takes in documents and uses an LLM to generate two (or as many as desired) documents about each file. Currently, it creates one overall summary of the file and a breakdown of each major section. The prompts that it uses to generate the documents can be adjusted to create any sort of output that is desired by developers.

Front-end Development: We created a UI with React that currently permits users to enter a query. Until it is connected with the backend, it will continue to provide an auto-generated response. The front-end also documents the entered queries to create a list of the most common queries that the user can access in future. The user can also create, delete, and clear query groups.

Roadblocks and Solutions

This section outlines various difficulties we encountered while working on this project. In some cases, these issues have been resolved and are only presented here for future

reference. Others remain active challenges that must be addressed as the project continues. We recommend that all future team members read this section especially carefully.

Project purpose and research vs. implementation: Perhaps the most significant challenge we encountered this semester was finding an appropriate balance between investigating different avenues for solving this problem and using this research to inform the development of a software system capable of applying these techniques to solve the problem in an applied, real-world scenario. In simpler terms, given the relatively short development period allowable within a single semester, we needed to weigh the value of conducting more research against applying this research toward an implemented product. Taken to an extreme, spending too much time on research without any development leads to intangible or otherwise unsatisfactory results.

Proposed solution: Indeed, we have found development to be an important part of the research process because it offers an avenue for testing the applicability of different techniques. While further development of our proof-of-concept system will certainly require further research, we caution against pursuing avenues of research that contradict or otherwise warrant the abandonment of the system developed this semester. Rather, we had the best results this semester once we chose a single path to pursue at length. We very strongly recommend that any future research conducted by later teams be done with the intent of further refining the existing direction of the project rather than starting over with an entirely new direction: we arrived at this conclusion after going through this process several times ourselves. The opinions and recommendations outlined in this document are the culmination of our collective understanding of the project as well as advice from faculty and industry experts alike. Finally, we want to emphasize the importance of development-validated research: as the system at the heart of this project is further enhanced and new ideas are explored for improving various aspects of its performance, it is important to implement these ideas and objectively test their efficacy.

Scope definition and deliverables: This capstone project's scope stands apart from others. While most are intended to be single-semester projects, this project is meant to span multiple semesters to continuously develop, improve, and rework ideas from previous project iterations. As the project’s first team, our first significant challenge was determining an appropriate direction for the project. Further, once we decided on an appropriate direction, our next challenge was to decide the scope for this semester and plan what we could reasonably implement toward the overall project goal with our remaining time.

Proposed solution: To overcome these obstacles, we, as a team, deliberated with partners from Michelin, Professor Russell, and faculty members to understand what was needed, what was expected of us, and what we felt was possible to do. We decided on the solution documented in this report and laid a set of goals we thought possible to complete in a semester. We then continued development keeping in mind that future

teams need to know what we did, how we did it, and how to use it. We took this approach to ensure the continuation of this project rather than its resuscitation.

COBOL query translation and similarity lookup: From our initial research and our discussions with Clemson faculty members, one of the greatest challenges we anticipated this semester was finding and implementing a query translation scheme for converting English user queries into a form that can be compared against the embedded source code to retrieve relevant snippets from the vector store.

Additionally, any routers later added to the system need to determine when to direct queries to the vector store for documentation and when to direct them to the vector store for source code. This is especially challenging because we anticipate many instances where queries bound for different data stores share a high degree of similarity, and there may be instances where retrieval is required from both data stores.

Proposed solution: A substantial portion of the work done this semester revolved around building a query translation and routing system suitable for our needs, but we eventually arrived at the solution described for our Query Translator in the Current Solution Architecture section. Solving the routing problem remains an outstanding issue that needs to be solved at the time the system is extended to draw context from multiple data sources. However, a promising approach is the use of an LLM and information about the available data to make retrieval decisions. For example, when the router LLM receives input containing mostly language syntax, it would choose to retrieve from the vector store containing source code. Similarly, it might retrieve from a documentation vector store for mostly natural-language input.

Response accuracy and performance testing: Since this system needs to deliver highly accurate responses due to its critical role in business rule extraction, it is paramount that this system consistently produces responses that are highly accurate descriptions of the legacy system's actual behavior. Due to the hallucinatory nature of large language models, we expect difficulty attaining complete factual accuracy in responses from our early systems. We did not have sufficient time this semester to implement a testing framework for our system. This will be one of the most important tasks for future teams.

Proposed solution: There are several well-known approaches for improving the accuracy of LLM output, especially in the context of an RAG system. The first and most promising approach is CRAG, or Corrective Retrieval Augmented Generation, in which the retriever plays a more active role by using its own LLM to assess the relevance of the retrieved information concerning the original query. We plan to extend this idea further by using more LLMs to verify the accuracy of generator output and further support this effort with manual test cases made from business rules known to be valid. Essentially, the idea is to create a suite of unit tests for the system. For each test, we have a human-verified question-response pair where the question is satisfactorily and

accurately answered. To run the tests, we ask the system each question and save its response. To grade the model's performance, we compare its answer to the correct answer, either manually or automatically with another LLM; the model's score is the number of passed test cases as a fraction of the total.

Summary generation and runtime: One of the problems our assistant faces is that, in a vacuum, the only relevant documentation it has, or that exists, is almost entirely source code, most notably COBOL with sparse or non-existent inline documentation. This is to be expected from a COBOL-based mainframe system. However, by providing more context, we can improve the output of the model. We have done this by feeding documents to an LLM and having it generate extra documents to use as context. This approach will be particularly useful for the final product because it allows users to upload their own documents and source code as context for the assistant. However, a significant problem we encountered is that even with 32 CPU cores and 2-3 A100 GPUs, the runtime of this program is unsatisfactory for an interactive system, especially since COBOL files tend to be several hundred lines long.

Proposed solution: We have experimented with different storage methods while developing this functionality including Pandas dataframes and torch datasets. We also considered hosting a database and having the LLM pull from there. The problem is that regardless of how it is adjusted, the file content is being passed as one large block of text. What is needed is a way to pass it in chunks and have it answered as a single query. The time constraint of this semester did not allow us to fully explore this problem but it will need to be continued in later semesters.

Recommendations for Future Research

Summary of current project state

Currently, our RAG pipeline uses our custom query translation to answer questions about arbitrary codebases from users but remains a prototype. We also have a separate program that generates summary documentation. It runs before the RAG pipeline to generate additional documentation to provide as context to the LLM. As of now, no testing has been done regarding incorporating this context into the RAG system itself. Recommendations for this are provided in another section.

For example, to use the RAG pipeline on a set of source files, you would feed that source code to the summary generator and then embed its output in the vector space used by the RAG pipeline. The reason for this separation is that the summary generator takes too long to run to be included in the pipeline. The front-end currently serves only as a visual representation of what the end product might look like. It has basic I/O functionality but doesn't interface with Palmetto or the pipeline itself. Once all desired context has been provided to the RAG pipeline, interaction is done directly through the same CLI used to run the pipeline.

Outstanding tasks for future teams

- **System benchmarking:** Before making any further adjustments or additions to the RAG pipeline itself, it is paramount to the continued success of this project that future teams establish appropriate methods for objectively assessing the overall performance of the existing RAG pipeline and each intermediate step within the pipeline.

    - **Query translation:** one of the most important parts of the pipeline is its ability to translate English queries into example code snippets to facilitate the retrieval of relevant actual-code snippets from the vector database to provide the model as context for answering the question. While current results are acceptable, our current method for doing this is rudimentary and inconsistent. Future iterations of this project should work to refine this process such that it yields more consistent results. Within the existing codebase, it is trivial to disable all generator functionality and examine only retrieval results due to LangChain's modular nature. Since high consistency and high accuracy are needed at this step, work should be done to automatically evaluate retriever performance against a suite of manually constructed unit tests.

      For our use case, we can treat the query translator and the retriever as a single combined black box when evaluating its performance: queries go in; relevant context snippets come out. Unit tests for this component should consist of known correct input-output pairs: each unit test should contain an English query and the set of context snippets from the vector store that should be returned for this query. Using this approach, testing is conducted as follows: for each unit test, give the translator-retriever system the English query and save the returned context snippets. Then, compare the returned context snippets to the expected context snippets and grade the unit test based on the size of the intersection of the two sets. Using field-specific terms, this approach uses the IOU (intersection over union) metric because it examines how many of the retrieved splits overlap with what we manually decided the retriever should have pulled for that query assuming ideal operation.

    - **Overall system:** Regarding the evaluation of the overall system's performance and its ability to accurately answer questions about a provided codebase, we have yet to identify a suitable approach for automating this process since it requires expert knowledge of the system itself to verify the accuracy of the RAG pipeline's claims about the system. However, since many of the business rules in the legacy system have already been identified through Michelin's manual migration efforts, the use of these known aspects of the system presents a promising avenue for assessing the system's performance by grading it on how accurately it identifies these already-known rules on its own. Barring this, a strong alternative is using another well-known codebase. For example, using an existing well-documented codebase whose functionality is already

well-understood. A good candidate project might be the group project for an early software development class since documentation is such an important part of such classes.

Once appropriate measures are in place for grading the system's performance, the system can be expanded and refined with different techniques. Establishing objective measures for the system's performance is a prerequisite to further augmenting the system because otherwise there is no objective way to measure how such augmentation impacts the overall system. Further, having objective metrics on system aspects like accuracy makes the project's results more compelling.

- **Few-shot learning:** Few-shot learning is an approach for increasing the reliability and specialty of LLM output by including a small number of examples in the prompt that create a pattern for the LLM to follow when generating a response. In the context of this project, this approach is applicable anywhere an LLM is involved. For example, the query translation could be improved by including a set of examples showcasing a variety of pairings between natural language prompts and their pseudocode snippet equivalents to prime the model for producing a comparable snippet with greater reliability.

- **Fine-tuning:** Fine-tuning involves partially retraining an existing AI model to make its inherent knowledge more suitable for application-specific tasks like translating natural language queries to example code snippets or summarizing a set of code snippets to answer a specific natural language query provided alongside such snippets. Because of the resource requirements needed for fine-tuning, this should be considered as a last resort. Additionally, fine-tuning makes models less useful for general applications, meaning it is less likely that fine-tuned models can be reused for different parts of the RAG pipeline. This means different parts of the overall pipeline require their own models. This is not ideal because running multiple models increases the already-high performance requirements of the overall system.

- **Adding generated context:** A considerable amount of time this semester was spent developing a summary generation agent to provide additional context to the model. Although we have tested its ability to generate summary documents, we have not yet fully implemented this output into the RAG pipeline. This is because we were developing this agent in tandem with our query translator. When implementing the two into the RAG pipeline we found it easier to have the retriever retrieve documents based on the code snippet generated by our query translator. Future iterations of this project should extend the RAG pipeline to retrieve documents from both the generated code snippets and plain text user queries. This process should be a straightforward matter of injecting the summary information directly into the generator prompt as additional context. However, we excluded this step from this semester due to time constraints.

- **Connecting the front-end:** At present, the front-end is a completely separate entity from the RAG pipeline itself. Our demos and presentations use hard-coded inputs and

outputs to demonstrate how the front-end would use the RAG pipeline to assist users, but we have not made any connection between the two.

    - Ideally, the RAG pipeline should provide a simple RESTful API to allow connecting the front-end or other applications. API operations should include sending user query requests to Palmetto and returning the output of the model.

    - As the pipeline advances and potentially moves off of Palmetto, the API would have to grow and adapt to meet the needs of the new configuration. But we believe a simple API version now would make this process easier as future teams would need only to upgrade the old API instead of starting from scratch at a later stage of development.

- **Improving performance:** As AI grows more popular and is more common in the workplace and at home, user expectations increasingly align more closely with the performance of popular models like ChatGPT and Gemini. For our assistant to be useful, it needs to generate queries quickly and accurately.

    - During our initial development, we focused more on the accuracy of our model than efficiency. We found it easier to build something that works and improve the performance afterward. Our summary generator can run for several minutes depending on the length of files and the compute resources available.

    - Ideally, any broadly adopted versions of this application have access to sufficiently powerful hardware, but we do not wish to rely on such assumptions. We have tried improving performance by limiting CPU usage, analyzing COBOL files in blocks, and using existing datastore frameworks like ChromaDB, pandas, and torch. Each step brought us closer to the performance we had hoped for, but there is still plenty of room for improvement. The beauty of this project is that other teams with new and fresh ideas will be able to work on and improve the performance of our model over time and bring it to its full potential.

Finally, we want to stress the importance of proactive documentation. We documented everything: if we researched anything, we wrote reports on it, even if it wasn't implemented (see Ontology and Context Engine reports) so that the next team could have that information for later reference. In GitHub, you will see four generations of the summary generator script so that new team members can see what changed as we learned. All this is to show that the most important thing you can do while working on this project is to write down what you did. If it failed, why didn't it work? If it worked, explain why it does.

Further, any step in the process is worth sharing with Michelin, typically in the form of Teams posts. Even in seemingly trivial cases like "I had this cool idea, I tried it, but it didn't work at all," these results are worth sharing. In other words, lessons from failures are still valuable. Michelin

is actively involved in the learning process and wants to be involved in the project's progress at each step.

Research Collection

Below is the compilation of reports and documentation produced this semester. Some detail various aspects of our research progress while others explain components of the project. Others are research and inquiries done on potential solutions that remain unimplemented or were abandoned.

AI Analysis for Small COBOL File: **AI Analysis for COBOL File.pdf**
Sample output from inputting a COBOL file into ChatGPT

COBOL Summary: **COBOL Summary.pdf**
A summary of the COBOL language and the basics of reading it

LangChain and Rag Summary: **LangChain and RAG Summary.pdf**
An introduction to RAG and LangChain, a framework used in development

Rag From Scratch: YouTube link
An intensive explanation of how RAG works. Helpful for determining places for improvement. Most of the actual work shown here is done by LangChain.

Ontology Report: **Ontology Report.pdf**
A dive into ontologies and how they may help the project

Palmetto 2 Environment Setup: **Palmetto 2 Environment Setup.pdf**
An instructional document on how to set up the environment we used in Palmetto

Potential Large Language Models: **Potential Large Language Models.pdf**
A list of the LLMs we considered using with their pros and cons

Query Translation: **Query Translation.pdf**
An explanation of query translation and how we planned to leverage it

Setting Up a GPT: **Setting Up GPTs.pdf**
A setup guide for using a GPT left over from when we considered GPTs as an alternative to RAG

Consensus on Using GPTs:
Using an existing solution can always have its benefits. GPT stands for Generative pre-trained transformer. GPTs are the technology at the center of popular modern AI tools like ChatGPT and GitHub Copilot in use in the industry. GPTs are an LLM technology originally created by OpenAI. Many GPTs are open-sourced and easy to integrate into an environment using Python scripts. They are excellent for fine-tuning and restricted datasets, allowing users to determine what they want the AI to evaluate. Without additional tooling, GPTs don't have direct internet access and are not useful for current events that require searching for outside information that is not already available from its training dataset. However, GPTs exist as a possible alternative to the primary Llama models used by our current implementation. They may also be used as helper models for other parts of the overall RAG pipeline.

Summary Document Generation: **Commentary on doc_reader.pdf**

History of the summary document generator, explaining the iterations it went through and how it was improved throughout the semester

Context Engines: **Context Engines Summary.pdf**

A dive into context engines and how they may be useful in our pipeline

Benchmarking: **Benchmarking Ideas.pdf**

A dive into how to benchmark a RAG pipeline

First proposal: **Original Project Proposal**

Our initial presentation explaining what we decided to pursue

Developer Guide

In this section, we outline the process of rebuilding the entire project from scratch to facilitate the onboarding of future teams by explaining each aspect of the current system. Below are the documents from the Research Collection section which outline the necessary steps to continue this project. We will also outline the path we took in the project, what we decided, and what we learned along the way.

These two documents outline the core information required to understand what we are working to build and how to set up the environment we have been using to develop in Palmetto.

LangChain and RAG Summary: **LangChain and RAG Summary.pdf**

Palmetto 2 Environment Setup: **Palmetto 2 Environment Setup.pdf**

These documents summarize the purpose and history of our methods for improving the RAG pipeline by creating supplemental context to improve its question-answering capabilities.

Summary Generation: **Commentary on doc_reader.pdf**

Query Translation: **Query Translation.pdf**

Development goals and rationale

In line with the goals outlined in the Project Scope section, this project is an AI assistant that answers questions about an arbitrary user-provided codebase. The long-term goal for this system is to assist the legacy migration process by accelerating the extraction of business rules and customer logic. When considering our system's implementation, it is important to recognize the research-centric nature of this project: what we are doing is akin to a large "brainstorming session." While our system design has an implemented prototype, this implementation exists as a proof of concept for our research and as a testing ground for new ideas aimed at improving this system's performance for eventual adaptation in an industry setting.

As a result, Michelin encourages exploring different approaches and avenues for improving this system. The end goal is to arrive at a finished design refined by extensive prototyping and testing that is suitable for use as something that can be adopted internally by Michelin.

Development outcomes

Overview

In the context of Michelin's problem, our project is a RAG-based AI assistant that answers queries by retrieving context from a legacy COBOL system and any available documentation. For example, if a certain customer has a seasonal discount that is automatically applied by the COBOL system, that functionality needs to be reflected in the new system. To find this information, a user might ask the assistant "Does customer A have any special discounts?" To answer this query, the RAG pipeline would find the COBOL source code and documentation relevant to that customer to analyze for seasonal discounts. The results of this analysis are summarized by the system's response to the user, streamlining the identification of these rules and their subsequent replication in the new system.

We chose RAG for this application because it allows us to give the LLM custom context while being more reliable and less time-consuming than fine-tuning or retraining. It also avoids the up-front compute cost needed for training a fine-tuned model. Our work this semester revolved around identifying and learning tools and frameworks suitable for implementing a modular, customizable RAG pipeline to refine iteratively toward our envisioned final product.

Palmetto access

Because LLMs like Llama3 require high-performing hardware to run acceptably, development for this project is done on Palmetto 2, Clemson University's high-performance computing resource. We recommend starting the Palmetto onboarding process as early as possible to avoid development delays. You must complete the Palmetto onboarding course provided by CCIT Research Computing & Data before you can use Palmetto. It takes about an hour to complete.

Development environment

Development this semester was done in a Python 3.10.15 virtual environment required by the LangChain modules necessary for implementing the system. Because Palmetto packages an outdated version of Python, we built Python 3.10.15 from source after determining that this version was appropriate for our use case.

Almost all of our project files use some version of Llama3 for natural language processing. Originally, we used LangChain's integrations for Ollama, a tool for streamlining the use of chat-based applications based on Llama models. However, Ollama requires a separate server process that must run alongside programs using Ollama as an LLM provider. We found this process tedious and instead migrated to HuggingFace-provided Llama3 models. HuggingFace models do not require a separate server process, are generally more flexible, and provide greater variety than Ollama. However, HuggingFace Llama models have gated access, meaning those wanting to use the models must first request access and obtain an access key. We recommend starting this process as early as possible.

Once on Palmetto, the above documentation explains how to reproduce our development environment.

Once we had a working development environment, our first task was implementing a basic RAG pipeline using LangChain, Llama 3.1-8B, and ChromaDB. The initial RAG pipeline was similar to the one found in the “RAG from Scratch” video above and was implemented as a starting point for future iterations and as an exercise to familiarize ourselves with our development tools.
We then set out on two avenues for improving this RAG pipeline. The problem we faced was that most of the documentation Michelin has to work with is raw COBOL, which is not surprising. However, we need varied context so the LLM can more easily access relevant information and create better output. To this end, we augmented our basic RAG pipeline in two important ways. Namely, we wrote custom code for query translation and supplementary document retrieval as detailed above.

Aside from the RAG pipeline itself, we also began work on a mock front-end using React for later use with an API allowing external apps to interface with the RAG system on Palmetto. As of now, this API remains unimplemented.

Another important aspect of our development process was our regular meetings and interviews with faculty members. These meetings helped us gain fresh insight into the problem and helped us maintain that the approach we were taking was logically sound.

User Guide

For setting up the development environment on Palmetto, see the above documentation for details about the necessary steps to set up the virtual environment. Before starting, you will also need to ensure that you have access to both the GitHub repository. This contains the RAG pipeline and additional resources useful for getting started with this project.

Palmetto usage
If set up correctly, the virtual environment in Palmetto should start automatically after logging in.

RCD is the team responsible for the Palmetto cluster and is available to answer any questions or problems you may run into while using Palmetto. A ticket can be submitted to them via Clemson’s IT department (CCIT) at IThelp@clemson.edu. Be sure to include Palmetto, RCD, or something similar in the subject line so they know to forward it to RCD.

If using Ollama

If you are using Ollama in your implementation, there are two things to do. First, when running the pipeline, you will need to run the appropriate commands to have an Ollama server running **while** the pipeline runs. Instructions on how to do this are in the environment setup guide. Further, if you are adapting previous code you will need to adjust the connection lines to work with Ollama instead of HuggingFace. An example of doing this via a subprocess can be found in doc_reader_gen1.py. In summary, the relevant code is as follows:

```
result = subprocess.run(
            ["ollama", "run", "llama3.2:1b"],
            input=prompt,
            capture_output=True,
            text=True,
            check=True
)
```

Job submission

It is absolutely necessary to have a Palmetto job running when executing any code. RCD provides documentation here about job submission and how to view and cancel current running jobs: https://docs.rcd.clemson.edu/palmetto/.

We have found that when working on a single script you can reuse the same job command since you will need the same resources each time. The RAG pipeline currently runs using this job submission:

```
salloc --nodes 1 --ntasks-per-node 1 --cpus-per-task 32 --mem 64G --time
4:00:00 --gpus a100:1
```

CPU usage and VRAM are the two limiting resources when running our code. We have also at times used 64 CPUs rather than 32 and 2 or even 3 A100 GPUs. This is the usual safe range we operate in depending on what we were running and how large the files we were reading were. Keep in mind that you are not guaranteed these resources and they are on a first come first serve basis. If you know you'll need a large resource pool, then consider submitting the job at a low-traffic time to have your job approved faster. Also, note that the above job is requested for 4 hours. This is because we would often leave Powershell open while developing for long periods. If you are only testing or demonstrating your project we recommend only submitting a 1 hour job.

Front-end

The front-end is a program written in React with the visual components being written in CSS. The program is split between three files: App.jsx, CommonQueries.jsx, and App.css. App.jsx is the primary file which includes functions such as addNewQG, commonQueryClick, and newInputEntered. The App.jsx creates the container for the page and integrates all of the components. The App.jsx uses the functions to handle the majority of user interaction with the interface. Navigation is used to connect the App.jsx to the CommonQueries.jsx. The CommonQueries.jsx consists of a function that retrieves and displays the ten most common queries. App.css establishes the visual display. The goal was for everything to have a clean and

cohesive look. Extra elements such as hovering effects were added to enhance user experience.

When a user starts to use the interface, they must first create a new chat. Within this chat, they can enter new queries which will produce the message “Auto-generated Response.” When the front-end and back-end are connected, the response will be the generated output. The user can then clear or delete their chat. The “Common Queries” button can be used to open a display that shows the ten most frequently posed queries. If a user clicks on one of these queries, a new chat is created with that query automatically sent to the system. At any point, a chat can be created, deleted, or cleared.