THE OHIO STATE UNIVERSITY

Computer Science and Engineering

CSE 5914: Capstone Design: Knowledge-Based Systems

Instructor: Eric Fosler-Lussier

The Arbiter: A Retrieval Augented Generation (RAG) Chatbot for Board Games

Brittaney Jin

Samridhi Kaushik

Matthew LeHoty

Adam Saleh

Jake Sutter

April 26, 2024

Contents

List of Figures

1 Purpose

When playing board games, there are a few common problems that always come up. For one, many times players can get into arguments about the correct way to play. Arguing about the rules can be tiresome and take the fun out of playing board games. Similarly, looking through a lengthy board game manual when you are unsure about a rule can be very time-consuming, especially if the board game is particularly complicated. Lastly, the learning curve for new players can be very steep, which makes complicated board games very daunting, and many people don't want to go through the effort of spending hours learning to play a new game. These are all problems we aimed to solve using our web application, called the Arbiter.

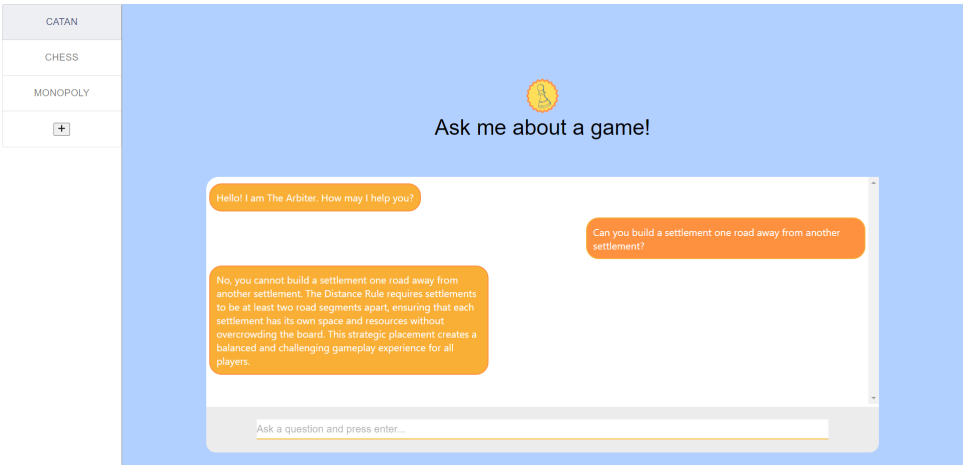


Figure 1: The homepage of The Arbiter with a chat interface.

Our project was to make a website with a chatbot that can answer ruling questions about board games quickly, easily, and accurately. In order to accomplish this task, the team decided to make a retrieval-augmented generation (RAG) application[3, 9]. Plain generative AI was not great for our uses because the answers could be wrong, hallucinated, or irrelevant. However, generative AI is great for engaging in human-like conversation. RAG is a method of improving the accuracy and relevancy of generative AI models by

feeding the model the correct and relevant facts from an external source. This is perfect for our application, because we want the answers to user's questions to be based on fact and pulled directly from a manual, but we also want the chatbot to be conversational. For our RAG application, we decided to use Flask[1] with Python for the backend server, Elasticsearch[16] through Docker[14] for our document store, React[23] with typescript for our frontend, and GPT 3.5 Turbo[24] API for making our answers human-like. Our plan for the Arbiter was for it to be very simple to use: simply go to a website, upload a board game manual in PDF format, and begin asking questions to the chatbot.

2 Project Timeline

2.1 Research and Skeleton Phase

The time period in between Checkpoint 1 (2/7) and Checkpoint 2 (2/28) is when we will be trying to do most of our research as well as laying out a skeleton for the project. The first step of this phase will be to concretely decide on what will be on the front end. This means determining precisely all of the components that users will interact with. During this time we will also draw up a layout for the website that includes the positions of the different components and generally what the "aesthetic" will be (What's our color scheme? What's our font? Etc.) In order to do this, we plan on using Figma[19] (or some other service that is good for mapping out a layout). Once we determine what will be on the front end and what we want it to look like, we want to create it with React. Considering our project is probably just one page with a simple chat bot, this seems feasible within a one month period. That being said, we do not plan on it being fully functional and connected to the backend quite yet.

On the backend side of things, our plan during this phase is to convert a manual into a json file so that it can be used with Elasticsearch. At this point we will probably just be using a dummy manual for testing, rather than an actual game manual. After testing that it works, we would also like to connect our backend to the ElasticSearch querying so that we have everything we need to begin serving things to the client. Finally, during this phase we would also like to do research on what the two different possibilities (generalizing vs. specializing) look like and have a good idea of where we think we're going to end up going with that. We would also like to research LLMs during this time so that we can have a grasp on how we might be using them throughout the project (the tech talk for LLMs should certainly help with this).

2.2 Backend Design decisions/Development Phase

The time period between Checkpoint 2 (2/28) and Checkpoint 3 (3/27) will be considered the backend design decisions and development phase. The biggest decision our team will make within this checkpoint phase is whether we want to specialize our application towards chess, or generalize the application to a variety of board games. This will have consequences for the backend and frontend structures. If we generalize our application, the front end will require a component that enables the user to select the game they would like to ask a question about. In the backend, there will have to be support for storing multiple documents that correspond to different games, and the AI agent will need to know how to select documents for information retrieval. These changes will have to be implemented once the decision is made.

Other tasks that will be completed in this phase of the project include connecting the frontend to the backend fully. Site navigation will be fully set up, and our server will load the correct information on each site page. This also means that commands or prompts entered by the user via the frontend will be handed off to the question-answering pipeline. This includes the process of sending the user's question to the server, processing the question into a more useful form, utilizing our AI agent to retrieve the correct information from the document store in our database, and reformatting and returning the relevant information in a readable format. The structure required for this process will be fully set up, and the AI agent will have to be devised and implemented. The core tasks performed by our AI agent will be retrieving the correct information that answers the question, and formatting it in an easy-to-read textual response. Lastly, our team would like to have a "FAQ" section on our site that displays the most frequently asked questions and answers. This prevents users from having to query our chatbot for commonly asked questions. The backend logic and frontend component for this feature will be implemented.

2.3 Final Touches Phase

There isn't too much to say for the time period in between Checkpoint 3 and Checkpoint 4. During this period we will be tying up loose ends/bug fixing and maybe adding in additional features if we have time. If we are behind on schedule, then this will be when we need to catch up. Otherwise, the bulk of the work will simply be working on our presentation for our project and properly documenting everything we need to document.

3 Design

3.1 Overview - How To Use

Our application, the Arbiter, has several different functionalities, which will be outlined in detail now.

The first thing that a user might do is upload a manual in PDF format of the game that they would like to ask questions about. The user would do this by clicking on the "+" button on the left sidebar. A pop-up window will appear for the user to select a file from their computer. Then, the user should give the game a name by entering it into the provided textbox. This name will appear on the left sidebar after the PDF has been uploaded. The chatbot also has some game manuals preloaded into it such as game manuals for chess, Root, etc since they are popular games that a user might ask questions about.

The second and main functionality of the chatbot is allowing the user to ask a question about the rules of a certain board game. To do this, the user should click on a game on the left sidebar to select it. Then, the user can begin to ask any question pertaining to the rules of that game by typing a question into the textbox at the bottom of the screen. When a user types a question and presses enter, the chatbot will think for a bit, then give the user a response that answers their ruling question.

The last thing a user can do is switch games. To do this, they can simply click on a different game on the left sidebar, and then immediately start asking questions about the new game that they have selected.

3.2 Workflow

Every user function above will now be described in terms of the high-level, client-server interactions. The first action that the user can do is upload a PDF. As mentioned before, the user can press the "+" button on the left side of the screen to upload a PDF of a game

manual. After the "open" button is pressed, the PDF will be collected from the frontend as a variable and passed from the client to the flask server, where the Flask server will make the appropriate calls to functions to add the game information to the Elasticsearch database. Then, the server will return to the frontend client the name of the index that the new game information was just added to. After the "open" button is pressed, the file pop-up window will disappear and a new window will appear with the option to give a name to the game associated with the PDF manual. After the user either enters a name or leaves it blank and presses enter, a new tab will appear on the left side-bar labeled with either the user-entered name or the name of the PDF file.

The second user interaction is switching game tabs. If the user wishes to ask questions about a different game, they must click a tab on the left-side of the screen to switch games. When the user clicks one of the tabs, the user will be able to ask questions about the game or PDF listed on the tab they click. When the user clicks on the tab of a particular game, the frontend client sends to the Flask server the name of the index associated with the game information in the elasticsearch database. This index name was passed to the frontend when the user uploads a PDF. This index name allows the Flask backend to query from the correct location in the elasticsearch database when the user asks a question.

The last user interaction is asking a question to the chatbot. After the user selects a game from the tabs on the left, the user may enter into the textbox any question pertaining to the game. After the user types in a question, the user should then either press enter on their keyboard or press the submit button on the screen, which causes the user's question to show up as a text bubble on the right.When this happens, the question in the form of a string gets sent from the client to the Flask server. Then, the Flask server calls a function that queries the elasticsearch database with the user-entered question. The answer returned from that function gets fed into another function which calls the OpenAI GPT-3 API and translates the answer into a human-like response. While all of this is happening, the chatbot displays a thinking face as its loading bubble. This human-like response eventually gets

returned to the client and shows up as a text bubble on the left side of the screen.

3.3 Frontend Design

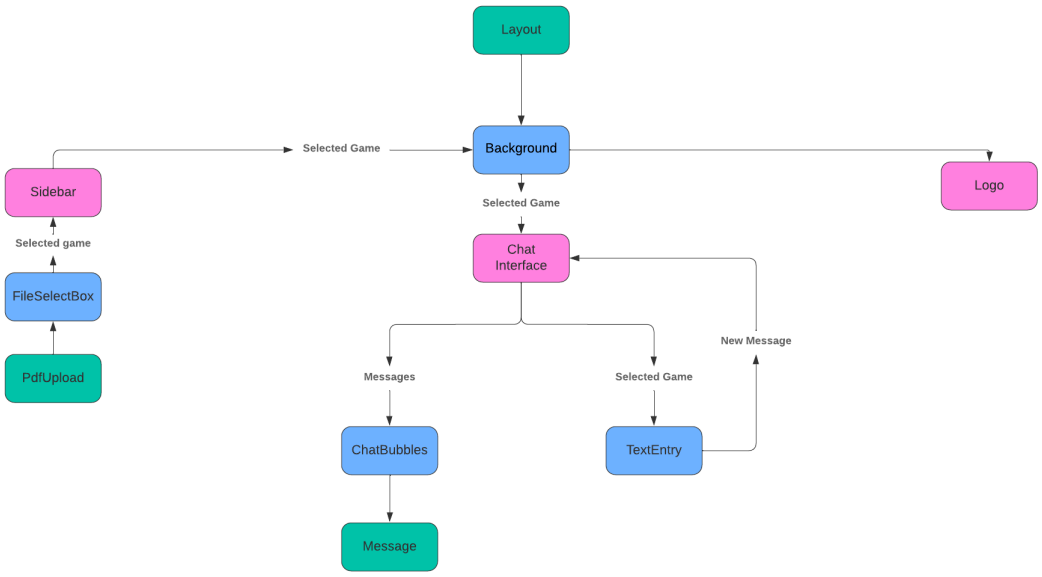


Figure 2: Frontend class hierarchy.

The front end is built from scratch using React.js along with some helper libraries. The team also used Google's MUI Materials[26] for nice looking components, FontAwesome[20] for icons, and Canva[13] to make a simple logo for our chatbot. The components are structured in a class hierarchy that allows for reusable, highly maintainable code, as shown above.

The main chat interface is made without any third-party chat library. Initially, we used an external service called TalkJS[25] for the styling, but after initial implementation, we realized that there were many problems with TalkJS, and we rebuilt the whole chat interface from scratch instead. In the implementation of the chat interface, a list of all messages make up the current chat. Each message only contains a string and an ID. When a message is sent or received, it simply gets added to this list which automatically re-renders

the page. The ID determines the styling of whether the message is from The Arbiter or the user.

The PDF upload functionality uses simple MUI Materials components. The sidebar contains toggle buttons of which only one can be pressed at a time. Each button represents a game. When a message is sent, the selected game is checked on the front end and the corresponding game index is sent to the backend so that the server knows which game in the Elasticsearch database to query from.

3.4 Backend Design

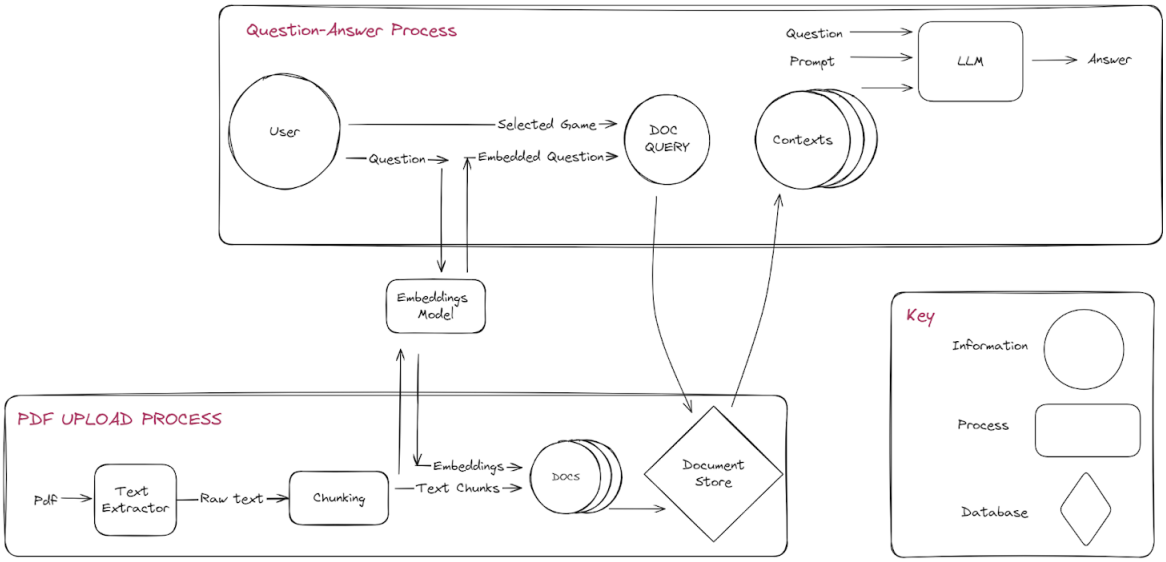


Figure 3: Complete application execution diagram.

3.4.1 Overview

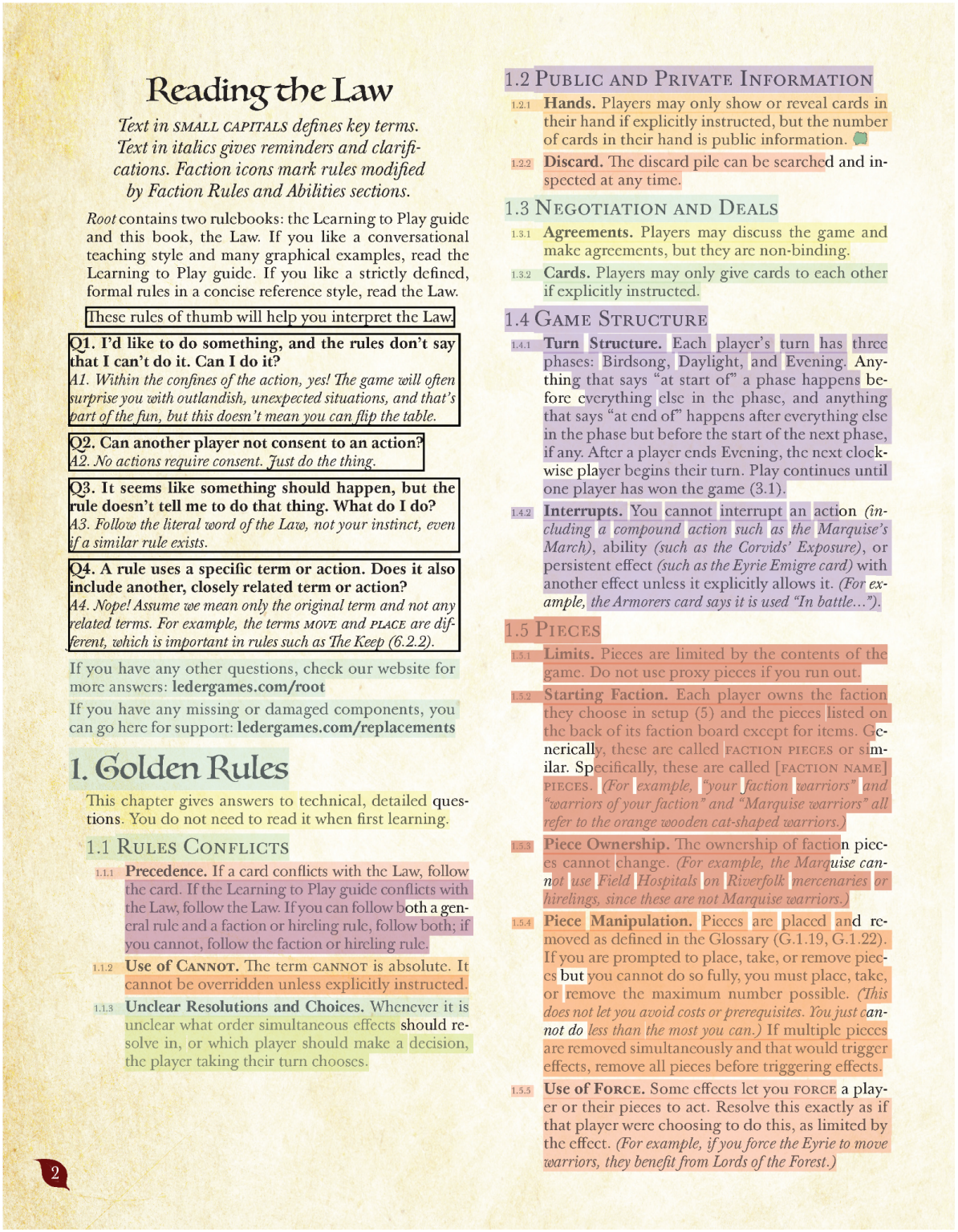
The server is built with React in Python and has 2 main routes or endpoints to respond to. The first is the /getAnswer route which takes in POST requests with parameters index and prompt. These are then passed through the Elasticsearch queries as the index to search

in and the question to search against the contexts. The returned contexts are then passed through the GPT 3.5 Turbo model and finally a complete answer is returned as a response. The other route is /uploadPDF which takes in a file upload, parses through the text with pypdf, chunks the text with langchain text_splitters, embeds it with the aforementioned model, and then uploads it into a newly generated Elasticsearch index. This index ID is passed back to the frontend as a confirmation of success and for future question referencing.

3.4.2 PDF Upload and Processing

When a user passes a game manual PDF to the server, several steps occur in order to prepare it for querying by the user. The first step is to extract the text from the file using the PyMuPDF python library[10]. This process is done in blocks (as visible in figure 4) because it allows for the text to already be grouped by general location. The issue that arises with this method is that some blocks get cut off in the middle of a sentence (visible in the bottom left blocks of text of figure 4). After some initial text processing (cleaning up newlines, hyphens, and special characters), we then search for and store the coordinates of the text to be used for highlighting later. Then, we run a simple set of checks to attempt to combine those broken sentences, such as finding the location of the last period character and checking if the next block begins with a lowercase character. This fails if the next block starts with a capitalized proper noun or a number. We conducted some tests on the benefits and trade-offs of different combining methods, but ultimately deemed it was sufficient to keep the chunks smaller at this stage and let later stages clean up sections that were still separated. The success of this process can be seen in figure 4 from the text titled "1.2 Public and Private Information" through "1.3.2 Cards."

The next steps begin the semantic chunking process. This is a method designed by Greg Kamradt[22] that involves grouping sequential chunks, vectorizing them, and comparing their semantic similarity/difference. Once all the blocks have been compared, we can graph the differences of sequential blocks and create chunk breakpoints where the values pass over



Reading the Law

Text in SMALL CAPITALS defines key terms. Text in italics gives reminders and clarifications. Faction icons mark rules modified by Faction Rules and Abilities sections.

Root contains two rulebooks: the Learning to Play guide and this book, the Law. If you like a conversational teaching style and many graphical examples, read the Learning to Play guide. If you like a strictly defined, formal rules in a concise reference style, read the Law.

These rules of thumb will help you interpret the Law.

Q1. I'd like to do something, and the rules don't say that I can't do it. Can I do it?
A1. Within the confines of the action, yes! The game will often surprise you with outlandish, unexpected situations, and that's part of the fun, but this doesn't mean you can flip the table.

Q2. Can another player not consent to an action?
A2. No actions require consent. Just do the thing.

Q3. It seems like something should happen, but the rule doesn't tell me to do that thing. What do I do?
A3. Follow the literal word of the Law, not your instinct, even if a similar rule exists.

Q4. A rule uses a specific term or action. Does it also include another, closely related term or action?
A4. Nope! Assume we mean only the original term and not any related terms. For example, the terms MOVE and PLACE are different, which is important in rules such as The Keep (6.2.2).

If you have any other questions, check our website for more answers: **ledergames.com/root**

If you have any missing or damaged components, you can go here for support: **ledergames.com/replacements**

1. Golden Rules

This chapter gives answers to technical, detailed questions. You do not need to read it when first learning.

1.1 Rules Conflicts

1.1.1 **Precedence.** If a card conflicts with the Law, follow the card. If the Learning to Play guide conflicts with the Law, follow the Law. If you can follow both a general rule and a faction or hireling rule, follow both; if you cannot, follow the faction or hireling rule.

1.1.2 **Use of CANNOT.** The term CANNOT is absolute. It cannot be overridden unless explicitly instructed.

1.1.3 **Unclear Resolutions and Choices.** Whenever it is unclear what order simultaneous effects should resolve in, or which player should make a decision, the player taking their turn chooses.

1.2 Public and Private Information

1.2.1 **Hands.** Players may only show or reveal cards in their hand if explicitly instructed, but the number of cards in their hand is public information.

1.2.2 **Discard.** The discard pile can be searched and inspected at any time.

1.3 Negotiation and Deals

1.3.1 **Agreements.** Players may discuss the game and make agreements, but they are non-binding.

1.3.2 **Cards.** Players may only give cards to each other if explicitly instructed.

1.4 Game Structure

1.4.1 **Turn Structure.** Each player's turn has three phases: Birdsong, Daylight, and Evening. Anything that says "at start of" a phase happens before everything else in the phase, and anything that says "at end of" happens after everything else in the phase but before the start of the next phase, if any. After a player ends Evening, the next clockwise player begins their turn. Play continues until one player has won the game (3.1).

1.4.2 **Interrupts.** You cannot interrupt an action *(including a compound action such as the Marquise's March)*, ability *(such as the Corvids' Exposure)*, or persistent effect *(such as the Eyrie Emigre card)* with another effect unless it explicitly allows it. *(For example, the Armorers card says it is used "In battle…").*

1.5 Pieces

1.5.1 **Limits.** Pieces are limited by the contents of the game. Do not use proxy pieces if you run out.

1.5.2 **Starting Faction.** Each player owns the faction they choose in setup (5) and the pieces listed on the back of its faction board except for items. Generically, these are called FACTION PIECES or similar. Specifically, these are called [FACTION NAME] PIECES. *(For example, "your faction warriors" and "warriors of your faction" and "Marquise warriors" all refer to the orange wooden cat-shaped warriors.)*

1.5.3 **Piece Ownership.** The ownership of faction pieces cannot change. *(For example, the Marquise cannot use Field Hospitals on Riverfolk mercenaries or hirelings, since these are not Marquise warriors.)*

1.5.4 **Piece Manipulation.** Pieces are placed and removed as defined in the Glossary (G.1.19, G.1.22). If you are prompted to place, take, or remove pieces but you cannot do so fully, you must place, take, or remove the maximum number possible. *(This does not let you avoid costs or prerequisites. You just cannot do less than the most you can.)* If multiple pieces are removed simultaneously and that would trigger effects, remove all pieces before triggering effects.

1.5.5 **Use of FORCE.** Some effects let you FORCE a player or their pieces to act. Resolve this exactly as if that player were choosing to do this, as limited by the effect. *(For example, if you force the Eyrie to move warriors, they benefit from Lords of the Forest.)*

2

Figure 4: Stages of the PDF Parsing process composited together.

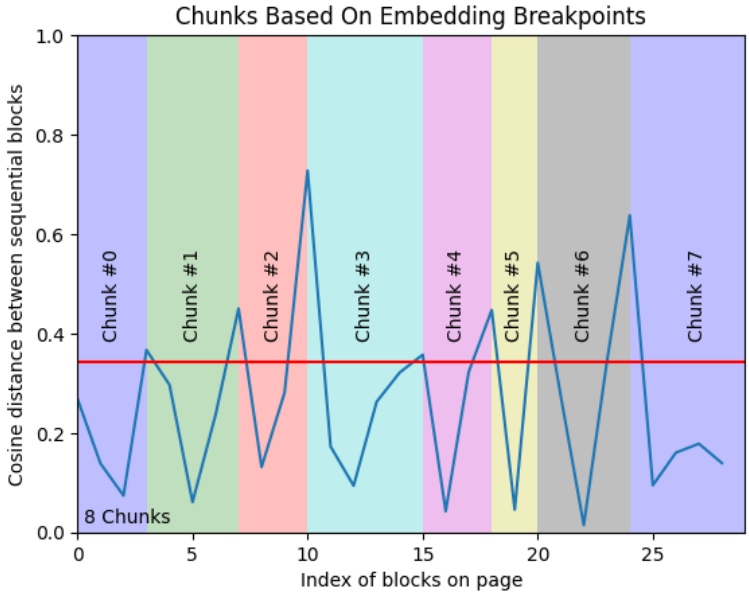


Figure 5: Differences of sequential blocks for a page in the semantic chunking process.

a certain threshold. This is the process visible in figure 5. Theoretically, these breakpoints are when the blocks seem to be about a new topic. The result of this is the large purple block headed with "1.4 Game Structure" in figure 4.

We chose to alter the original semantic chunking design slightly by separating this process by page, as conducting it with the whole document would not allow the threshold to be distributed equally across all pages. That is to say some pages may have outliers or such large values that would disrupt the desired behavior of practically every other page.

The final stage of data prepping involved actually creating those chunks based on those breakpoint indices, and then adding one more safeguard of information catching by overlapping these chunks by one block ahead and behind itself. This allows for more than one chunk to hold information on a topic in case it would have otherwise been miscategorized and creates a sort of gradient effect as visible in the remaining bottom right section of figure 4.

With the full text of the game manual, the server then begins creating the final vector

embeddings. In this and the above process, we use the HuggingFace all-MiniLM-l6-v2[18] dense vector embedding model. Note that the same embedding model must be used across the project or at least between a single game and subsequent queries for that game. To finish up the process the server generates a unique ID for the new Elasticsearch index which is based on a timestamp of the current time, and then inserts the documents with an index mapping of the text vector, the text itself, the coordinates of the text, and the page(s) the text is located on. Finally, the ID of the newly created index is returned to the client so it can be passed back to the server for reference in future user queries.

3.4.3 Answering a User Question

When the server receives a question from the frontend, a series of steps are performed to query ElasticSearch for relevant text and formulate a coherent answer. First, the server sends the question along with the corresponding game index to an ElasticSearch function. This function utilizes the HuggingFace all-MiniLM-l6-v2 dense vector embedding model to embed the question into a numerical vector representation. This is the same embeddings model used to embed the text of the game manuals. Next, the ElasticSearch database is queried for the text of the board game manuals that match the user's question the closest. This is done via the K nearest neighbors search algorithm (KNN)[15], which is implemented by ElasticSearch. Essentially, the vector representation of each chunk of game instruction text is compared to the vector representation of the question, and only the kth closest vectors and their corresponding text is returned. The relevant pieces of text are then sent to a function that handles the call to the large language model. The human-like translation of the final response of our application utilizes a pre-existing LLM, GPT 3.5 Turbo. Finally, this human-like response is sent back to the server function that handles the question answering, and the answer is returned back to the frontend, where it can be displayed in an online chat-like view. The prompt sent to the LLM is included below:
Note that the context is included in the prompt, and the LLM is instructed to answer the

```
content = "With the following context: \n"
for index, context in enumerate(contexts):
    content += f"context {index+1}: \"{context['text']}\"\n"
content += f"Answer the following question as concisely, yet professional as possible: \"{question}\" ."
```

Figure 6: GPT-3.5 Turbo Prompt

question utilizing the context.

4 History of Design Alternatives

4.1 Early Ideas

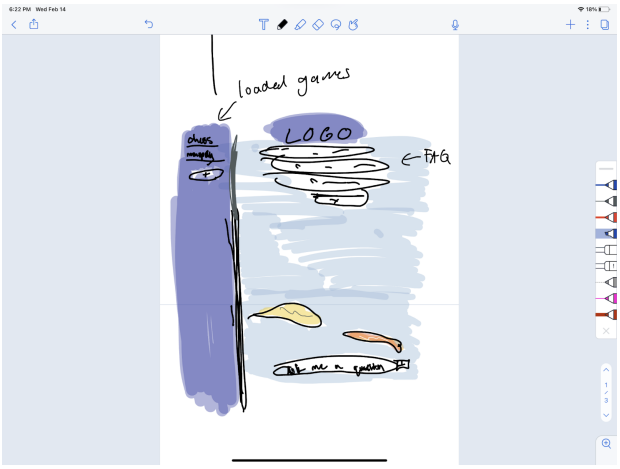


Figure 7: First sketch of the Arbiter front-end

As a team we were unsure of whether this chatbot should be generalized to any board game or specialized on a handful of board games because we are unsure of the scope of the technologies we are using. However, the team does have a plan for what to accomplish before we begin to either generalize or specialize.

As a first step, we want to make the bot function specifically for chess. Chess is a good first step because there are a ton of resources and data out there related to the rules, and it's fairly simple yet still has some rules that many people don't know. It can often be confusing to really understand what a stalemate is, when a draw happens, and when you can and cannot castle, among other things. Our hope is that the bot can resolve issues like these quickly and effectively. During the chess making period we will be doing our best to make sure that the code is never specific to chess so that it is reusable. This way, when we have a feel for the technologies we're working with and want to expand, we can do so without much refactoring.

After chess, our plan is to make a decision about whether we want to generalize or

specialize. Generalizing means that we would try to support a very large variety of board games in a way that is as simple as feeding our bot data on the board game (manuals and forums), and having it learn the rules from that. In this scenario, the user would feed our chatbot PDFs of manuals and links to official game forums of any game they want to ask questions about. Then, our chatbot would answer a user's questions based off of the user's data. Specializing, then, means that we will specialize to a select few supported board games. In this scenario, the manuals and official forums our chatbot would use would not need to be supplied by a user and would be built into our system. If our team decides to specialize, we would choose complicated board games so that the bot will have a significant reason to be used. Many members of the team are fans of the board game Root, which is notoriously complicated, has tons of expansions, and has a learning curve that will keep players asking questions even on their tenth playthrough. If we specialize, we would like to be able to support games like Root.

Even though our group is undecided on whether we want to make our application specialized to a handful of games or generalized to all games, there are still some specific use cases that we imagine will be in the final application, one of the main use cases being the "chatbot" feature. The front page will have a textbox with a prompt for the user to ask a question about a board game. The user will be able to type into the textbox a question in plain language, then hit enter. Afterwards, our application will be give an answer to the user's question in plain language, as well as provide the source that they got their answer from in the form of a link. That link would bring up a PDF of the original manual and highlight the exact section or paragraph that the chatbot got their answer from. It is important to mention that the final answer will be factually correct, as it will pull directly from the source manual or official forums. Here is an example interaction concerning chess: A user enters the question, "Every move I make results puts me into check. What does this mean?". The chatbot would give a reply similar to, "This position is called stalemate. This means the game ends in a draw".

4.2 Design Alternatives

4.2.1 Specialization vs. Generalization

At the beginning of the project, the team was conflicted whether we wanted to generalize the chatbot to provide assistance for any board game or to specialize it to only provide assistance for chess. The team has decided to generalize the chat bot. This has several implications for the overall design of the application. On the web page which has the chatbot, there is a sidebar on the left which will include a list of board game manuals that the user can use the chatbot with. If we had chosen to specialize the chatbot to just chess, this sidebar would not be needed or it would have only had a chess manual included. The frontend will need to allow users to upload a manual in PDF format. Due to this, the backend would need to have functionality to break down the text in the PDF into smaller pieces and parse them. On the backend side, ElasticSearch queries would most likely need to be tailored so that searches are specific to a certain board game manual. This might require many different search configurations compared to one or two search configurations if the chatbot was focused on providing guidance based on chess manuals only. When sending a question to the backend, for generalization, additional information will need to be sent which indicates which game the user is asking about/what game manual they want to retrieve answers from. This can be seen as a con for generalization as more info and processing of that info will be required as compared to specialization. With generalization, the team will also need to think about whether newly uploaded manuals will be stored in a database for future use or they will only be temporarily used in the current user session. If we are providing preloaded game manuals, the team might also have to differentiate between different versions of manuals for the same game. A big pro of generalization is that we can provide greater functionality to users so they can use the chatbot with a large variety of games which is lacking with specialization. In the end, the team decided to generalize the chatbot since having greater functionality is an important feature and it

would also allow the chatbot to appeal to a wider user base.

4.2.2 Frontend - React.js vs. Vue.js

When it comes to choosing a front-end framework, there are many choices available including React, Vue, and Angular. Our two main choices for a front-end framework are React[23] and Vue[27]. Both of these frameworks are similar to a certain extent, but each has its own pros and cons. Both React and Vue are used for building single-page applications and are component and prop-based. Both also use JavaScript, HTML, and CSS. The core difference is that React allows higher scalability and flexibility, meaning we can build larger, less constrained projects using React. Vue's core advantage is that it is extremely lightweight and simple, which is why it's generally used for smaller projects. Since our frontend is small – a single web page with a chatbot interface, Vue might be a good choice.

However, another difference between React and Vue has to do with the surrounding resources. Vue is newer, less popular, and therefore has less plug-ins and general support, whereas React has a host of guides, plug-ins, and general resources for those who want to make something in React. Vue is catching up to React slowly, but as of right now React is the more supported framework.

Though Vue is lighter-weight and perhaps more appropriate for our application, which has a relatively small front-end, one of the most important things to consider when picking a front end is the comfort level of the developers. On our team, almost all of us have experience with React while none of us had experience with Vue. Moreover, one of us has experience with Material-UI, a component library for React that can do a lot very quickly. Though Vue is very approachable and user-friendly, it's difficult to beat raw experience and the sheer amount of support available for React. Those are the two main reasons why we have ultimately settled on React.

4.2.3 Document Store - ElasticSearch Alternatives

A large design alternative our team has grappled with is the implementation of the question answering pipeline and how exactly we should store text from our factual document store. Initially, the plan was to use ElasticSearch to store text. ElasticSearch is a feature rich database that allows for very flexible queries. Some types of queries include exact match and fuzzy match. These searches look to match words from the query text to words within pieces of text stored in the database. Fuzzy match allows for a set number of differences in words or characters between the query text and a potential match text. ElasticSearch also has support for many different query types that are highly customizable. The problem we have run into is that ElasticSearch does not have an out of the box semantic search for text. While there is a newly released ElasticSearch neural embeddings model that can be set up to allow for semantic vector-based search, that feature is not available in the no-cost version we are utilizing. This is an issue because in our case, the query phrase we are using to retrieve relevant texts from the database is a question, and thus it will not match the exact text of documents very closely. Unless a very large amount of preprocessing work is done, it seems that using the common query types within ElasticSearch will not yield the accurate results. According to ElasticSearch documentation, ElasticSearch is capable of k-nearest-neighbor search. This means that if text is embedded separately and then loaded into ElasticSearch with its corresponding vectors, then this query type may work. However, it is not clear if this is possible with the free version of ElasticSearch yet, and a successful implementation of this search type has not been completed.

Semantic vector-based search[17, 4] via a general vector database is the alternative method our team is exploring. This approach essentially uses a model that embeds our textual information and the question text into a vector based representation. Querying the database for text in the document store that correlates to the semantic meaning of the query question could be as simple as returning the text in the database whose embedding

is closest in vector space to the vector representation of the question. FAISS (Facebook AI Similarity Search)[12] is an implementation of a vector-based database that was built solely to store embeddings and search for the best matches. It appears extremely simple to use out of the box and seems to require little customization to accomplish our task. One difference with ElasticSearch is that FAISS is much more lightweight. Instead of existing as a typical database, it is essentially a large data structure that exists in RAM. Some downsides are that FAISS doesn't have support for multiple indexes, so different instances of FAISS would be run for each board game. Also, unlike ElasticSearch, there isn't support for searching for matches based on certain metadata.

Ultimately, more research needs to be performed into each of these tools, and we would also like to test both of them out before making any decisions. We will ultimately choose the tool that yields the best results. After preliminary testing, the FAISS method seems to return better results; however, if we are able to get the dense vector nearest neighbor search running in ElasticSearch, then ElasticSearch may be the better choice with its multitude of features. It is important to consider the performance of either method will also depend on the preprocessing and postprocessing that is done with the textual data.

4.2.4 Backend - Flask vs Django

Any web application needs a backend framework to perform any processing tasks, including but not limited to database retrieval and storage, calculations, and processing requests from the front-facing web page. The backend of an application is responsible for all of the functionality of the application, and so it is important to choose the correct backend framework and the best language for the backend to be written in.

When deliberating about what language the backend should be written in, we considered mainly either a Python backend or a Javascript backend. The reasons these two languages were considered initially were because there was a lot of documentation for these kinds of backends, and members of our team had slight experience in both.

One of the main benefits of a Javascript backend is that it has the capacity for scalability in ways that Python backends cannot. That is because Javascript has multithreading capabilities (running code currently) while Python is single-threaded. Additionally, the most popular Javascript backend, Node, has event-based architecture, meaning that its inputs and outputs are asynchronous. This makes Javascript perfect for a chatbot application. However, despite these possible benefits of Javascript, we chose Python as our language. The main reason is that Python has extensive AI libraries that Javascript backends may not have the support for, which we may want to explore as options for preprocessing and post-processing data. Even though Elasticsearch can be used with any backend, machine learning and AI model libraries are not the easiest to code in other languages and we did not want that to be a barrier. Additionally, our application will overall be very small, with a limited set of possible user interactions, so scalability and efficiency from Javascript would be negligible.

After our team narrowed our option down to a Python backend, we had to decide which Python backend to work with. We deliberated between the two most popular Python backend frameworks: Flask[1] and Django[21]. Django has the most tags on Github closely followed by Flask, so the documentation and examples for both are extensive. The fundamental difference between the two frameworks is that Flask is considered a microframework while Django is a full-stack backend framework. The philosophy behind microframeworks is more "bare-minimum"; Flask has only the most basic functions needed to get a basic web application up and running. It is designed so that users only need to import what libraries they need and nothing else. This makes Flask extremely lightweight and flexible. However, it does not come with many useful built-in functions like an ORM for database communication, login capabilities, or security. Flask is designed for smaller applications. On the other hand, because full-stack frameworks have more built-in functionality, Django is designed for more robust, complicated, larger applications. Django does come with builtin functionalities such as an ORM and has a more "batteries-inclusive" philosophy.

Additionally, it is more rigid and has more conventions, which is great for large teams and for projects with a need for consistency and scalability.

Our application does need functionalities such as ORMs as we don't have a SQL database to store data, only Elasticsearch documents. Additionally, we have no need for a login system with authentication, and security is not a priority as all data being used is already public. Therefore, many of the uses of a full stack framework like Django would not be in use. Additionally, our application is very much "bare-minimum", with our planned design being a maximum of two different pages and the core functionality being only the chatbot. Therefore, Flask seems like a much better fit. Lastly, Flask's flexibility will be a benefit for our team given that we have a lot of options to explore in terms of technology.

4.2.5 Chatbot Interface - TalkJS vs From Scratch

One of the biggest design decisions we made during checkpoint 3 was incorporating TalkJS[25] into our frontend. We made this decision because it was clear that it would be very time consuming to make good looking chat bubbles from scratch, and TalkJS provides a lot of functionality for free. Using TalkJS was not without its issues, however. We had to learn how to synchronize our frontend chat bubbles with the TalkJS interface, parse a TalkJS message in the backend, and then formulate our response within a TalkJS message to send back. This required a pretty substantial amount of effort, as much of it involved a deep dive into the TalkJS API documentation. Additionally, there was no way to code in loading bubble functionality, and passing the specific game index to and from TalkJS would be extremely difficult. So, in the end, we decided to build our own chat interface from scratch in React.

4.2.6 PDF Text Extraction

The PDF text extraction process offers many interesting design decisions along each stage of the procedure. Most of the initial choices we made were done so for the speed at

which we could implement them and get to a minimum viable product. The following are some of the approaches we are considering in future iterations.

There are several ways a PDF can be created and arrive on our server for processing: digitally, such as when a document was both created and shared from a computer; scanned, such as when a document was printed and scanned into images that were placed in a PDF format; and scanned with OCR (Optical Character Recognition), where a physical document was scanned and the resulting image was parsed for text and placed in a machine-readable format by some software. The original python PDF parsing library we used, pypdf[6], does not use OCR. Thus, our application has no way to read purely scanned PDFs. However, in its documentation, pypdf brings up a few reasons why it doesn't use OCR[5], such as how raw text extraction won't ever confuse certain similarly shaped characters or rare Unicode characters like emojis. It can also obtain metadata about the pages and text like fonts and encodings. This has led us to looking for a solution that eventually involves both. We imagine this could look like using both and either comparing them with some heuristic, or using them in conjunction to cover the areas that both can't individually.

Once text has been sufficiently parsed and is ready for further processing, the system then has the choice of how to split up the text into smaller, more manageable chunks. As stated previously, the current method simply splits on 512 tokens with a 256 token overlap. One consideration we have is to refine these hyperparameters until we find an optimal size and overlap for these chunks so that they contain more precise information, i.e. chunks without several unrelated rules. Another consideration is to utilize an idea called semantic chunking, which splits the text into sentences and then groups the sentences based on embedding similarity. This seems to be the most promising, but also the most involved and potentially unpredictable approach.

4.2.7 Embedding Model

One of the last design decisions we are still working through is which model and what format to embed our text into vectors with. We currently use the all-MiniLM-l6-v2 model from HuggingFace sentence-transformers[18]. This creates a 384 length dense vector, which is a vector with all indices populated with some non-zero value. We chose this model because it was quick and free to start using. We would like to continue to search for other models and see how they perform on our typical game manual use cases. However, we have also found that Elasticsearch supports the use of sparse vectors, which are similar to dense vectors but instead many of their indices are zero values. Sparse vectors have been found to both save space in the system as well as perform operations faster. This is something we would like to take advantage of in our next iterations.

5 Ethical Considerations

When developing a new technology, especially in the AI sector, it's important to analyze the ethical concerns that may occur during development and after release. While retrieval-augmented generation isn't a completely new technology, the current version of what we've built behaves slightly different from typical systems and use cases. Many prominent cases of RAG feature a service that pulls in large amounts of a company's data, such as a knowledge base, that can then be used by a chatbot to help customers and employees solve problems specific to that company[11]. Our case, however, is a standalone website that users visit to upload a PDF that they likely don't have the creative or legal rights to grant us access to view. Furthermore, while our application was built purely for educational purposes and is not accessible on a public website, we will still examine the ethics of this project from the perspective of if this were a real product.

In late 2023, several organizations and companies, such as the Authors Guild and The New York Times, filed lawsuits against OpenAI for copyright infringement of their works[8, 7]. Specifically in the case of the Authors Guild lawsuit, they noted how their books were used, and even pirated, for training their model without permission, compensation, or notification. They also argue that the capabilities of the model were vastly impacted by their works, saying they would have a much different final product without them. Lastly, they remarked on the impact this technology is having on the market, allowing for much easier impersonation of the authors and creation of deceitful books that are sold under the guise of being written by a human.

In our project, we want to ensure we're always emphasizing the proper credit and value is being attributed to the creators and that our answers are being sourced from the original document only. We also argue that our functionality is independent from the data. While it does require a document be uploaded in the first place, it is not particular about what document, nor reliant on retaining any past documents to perform any better or worse.

Thus, the quality of the product is not dependent on the copyrighted works, in the sense that it wouldn't irreparably break if certain companies did not want their manual to be used. Lastly, we consider our product to be collaborative to the game industry and not harmful, as seen in the OpenAI cases. We couldn't identify any professions that would be negatively impacted; designers still need to create the game, develop the rules, and write the manuals. It is simply a tool to help elevate the enjoyment of the game itself.

Another couple of relevant cases to examine, which have similarities to our project, is that of the peer-to-peer application Napster[2] and the use of illegally acquired ROMs played on emulators. The ultimate demise of Napster occurred because they provided a directory for finding and retrieving copyrighted data, and maintained that database on a central server. This allowed users to easily download almost any song they wanted and they were eventually found guilty of copyright infringement. Similarly, emulators are often debated on their ethicality, because, while the emulators themselves are legal to build and download, the "requirement" is that the ROMs are also legally obtained. However, that rule is not typically followed, as evidenced by the many sources and discussion threads on obtaining these ROMs.

For our project, we differ from services like Napster because we don't offer any way of obtaining the copyrighted data. It is only used by the system to answer questions, not to provide a location to retrieve the whole document. Likewise, we evaluated the most common scenarios of users of our app and believe that a significant majority of the player base will be playing with a legally acquired game and thus a legally acquired manual, which they could either digitally scan or most likely find a free version on the games official website, as we have found in our own research.

6 Future Work

While we are pleased with the outcome of our project, we still had some ideas and functionality that we didn't get to implement due to the time constraints of the semester. If given more time or revisited later on, we would like to see the following items added to the application for increased usability.

First is the addition of optical character recognition (OCR) in the data ingest process. Currently, the PDF text extraction process only reads from the underlying file data to obtain the text on the page. This becomes a problem if a user who may not be technologically savvy wants to use our application and can only get an image or a scan of the manual without any digital representation of the text.

Another feature we considered early on in the project was a type of frequently asked questions section. The idea behind that would be for us to track some common questions users are asking about a specific game, provide them an answer, and allow them to rate the response. Finally, we would choose the top questions and answers and offer them as autofill prompts for the user, similar to search suggestions and the "People also ask" section on Google.

One feature our chatbot currently lacks that may seem abnormal when compared to applications like ChatGPT or Google Assistant is that of memory or remembering previous question contexts. For example, after getting a response from a different application, a user may be able to ask a follow-up question like "how tall is that" or "can you elaborate more on that?" Those applications can often correctly identify what the word "that" is referring to, yet our application has no such capabilities. While we might not be able to do too much work on this, we could at least try including previous contexts in the subsequent query to allow for more natural conversations.

Lastly, there is always more work that can be done on the data processing itself, whether it be the parsing, storage, or querying of it. We implemented a moderately unique solution

for parsing the PDF with the use of semantic similarity search, however that was only the third chunking process we tried. There are many other methods we could have tried and compared results from if given more time.

7 Deployment Documentation

To run the application in your own environment, you must have the latest versions of python and pip, npm, and docker already installed. Once you have those prerequisites satisfied and the repository cloned or downloaded, you can begin the installation process. Note that all of these steps are also documented in the GitHub repository's README in a more developer friendly format.

To install all the necessary python packages, first create a python virtual environment in the root directory. Then be sure to activate the virtual environment in your terminal of choice. Finally, run the command pip install -r requirement.txt.

To install all the necessary node modules, navigate to the client folder in your terminal of choice and execute the command npm install.

To set up your Elasticsearch docker instance, follow the tutorial on https://www.elastic.co/guide/en/elasticsearch/reference/current/docker.html. In summary, you need to create the elastic docker network, pull the Elasticsearch docker image, and start the Elasticsearch container. On the first time setup, the console will print some important key values, notably a Certificate Fingerprint and Elastic Password. Make sure to copy those values and place them in a new python file called settings.py in the server folder. Place them in a dictionary with the names 'CERT_FINGERPRINT' and 'ELASTIC_PASSWORD'.

Finally, you are ready to run the servers. From one terminal window, navigate to the root project directory and execute the command flask –app .\server\app.py run. This will start the flask server with a connection to the Elasticsearch instance. In another terminal window, navigate to the client folder and execute the command npm start. You can now visit http://localhost:3000 to use the application.

8 GitHub Publication

The source code and application usage instructions can be found in the following GitHub repository: https://github.com/jittaneybrin/thearbiter/tree/main

References

[1] Pallets. *Flask - User's Guide*. 2010. URL: https://flask.palletsprojects.com/en/3.0.x/ (visited on 04/26/2024).

[2] Mike Freedman. *Peer-to-Peer File Sharing*. 2014. URL: https://www.cs.princeton.edu/courses/archive/spring14/cos461/docs/lec14-p2p.pdf (visited on 02/04/2024).

[3] Patrick Lewis et al. "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle et al. Vol. 33. Curran Associates, Inc., 2020, pp. 9459–9474. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf.

[4] Elasticsearch. *Semantic Search: Bringing search experiences into the AI era*. 2023. URL: https://www.elastic.co/pdf/semantic-search-bringing-search-experiences-into-the-ai-era.pdf (visited on 04/26/2024).

[5] Mathieu Fenniak and pypdf contributors. *OCR vs Text Extraction*. 2023. URL: https://pypdf.readthedocs.io/en/stable/user/extract-text.html#ocr-vs-text-extraction (visited on 03/01/2024).

[6] Mathieu Fenniak and pypdf contributors. *Welcome to pypdf*. 2023. URL: https://pypdf.readthedocs.io/en/stable/ (visited on 03/01/2024).

[7] Michael M. Grynbaum and Ryan Mac. *The Times Sues OpenAI and Microsoft Over A.I. Use of Copyrighted Work*. 2023. URL: https://www.nytimes.com/2023/12/27/business/media/new-york-times-open-ai-microsoft-lawsuit.html (visited on 02/04/2024).

[8] The Authors Guild. *The Authors Guild, John Grisham, Jodi Picoult, David Baldacci, George R.R. Martin, and 13 Other Authors File Class-Action Suit Against OpenAI.* 2023. URL: https://authorsguild.org/news/ag-and-authors-file-class-action-suit-against-openai/ (visited on 02/04/2024).

[9] Rick Merritt. *What Is Retrieval-Augmented Generation, aka RAG?* 2023. URL: https://blogs.nvidia.com/blog/what-is-retrieval-augmented-generation/ (visited on 04/26/2024).

[10] Artifex. *Welcome to PyMuPDF.* 2024. URL: https://pymupdf.readthedocs.io/en/latest/index.html (visited on 04/22/2024).

[11] AWS. *What is AmazonQ Business?* 2024. URL: https://docs.aws.amazon.com/amazonq/latest/qbusiness-ug/what-is.html (visited on 04/26/2024).

[12] James Briggs. *Faiss: The Missing Manual.* 2024. URL: https://www.pinecone.io/learn/series/faiss/ (visited on 04/26/2024).

[13] Canva. *What will you design today?* 2024. URL: https://www.canva.com/ (visited on 04/26/2024).

[14] Docker. *Docker overview.* 2024. URL: https://docs.docker.com/get-started/overview/ (visited on 04/26/2024).

[15] Elasticsearch. *k-nearest neighbor (kNN) search.* 2024. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/knn-search.html (visited on 04/26/2024).

[16] Elasticsearch. *What is Elasticsearch?* 2024. URL: https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro.html (visited on 04/26/2024).

[17] Elasticsearch. *What is semantic search?* 2024. URL: https://www.elastic.co/what-is/semantic-search (visited on 04/26/2024).

[18] Hugging Face. *all-MiniLM-L6-v2*. 2024. URL: https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2 (visited on 04/26/2024).

[19] Figma. *Figma*. 2024. URL: https://www.figma.com/ (visited on 04/26/2024).

[20] Fonticons. *Font Awesome*. 2024. URL: https://fontawesome.com/ (visited on 04/26/2024).

[21] Django Software Foundation. *Django documentation*. 2024. URL: https://docs.djangoproject.com/en/5.0/ (visited on 04/26/2024).

[22] Greg Kamradt. *The 5 Levels Of Text Splitting For Retrieval*. 2024. URL: https://www.youtube.com/watch?v=8OJC21T2SL4 (visited on 04/26/2024).

[23] MetaOpenSource. *React - The library for web and native user interfaces*. 2024. URL: https://react.dev/ (visited on 04/26/2024).

[24] OpenAI. *Models - GPT-3.5 Turbo*. 2024. URL: https://platform.openai.com/docs/models/gpt-3-5-turbo (visited on 04/26/2024).

[25] TalkJS. *TalkJS Docs*. 2024. URL: https://talkjs.com/docs/ (visited on 04/26/2024).

[26] Material UI. *MUI*. 2024. URL: https://mui.com/ (visited on 04/26/2024).

[27] Vue.js. *Introduction - What is Vue?* 2024. URL: https://vuejs.org/guide/introduction.html (visited on 04/26/2024).