Computer Science Capstone (SIM3)
Task 2: Design and Development

Cristian Sotomayor

ID #001216131

WGU Email: csotom2@wgu.edu

Date: 12/21/2023

Computer Science Capstone - C964

Part A: Letter of Transmittal

12/21/2023

Senior Managers and Executives
Construction Tech Ventures
123 Hackerway, Los Angeles, CA 91234

Subject: Implementation Proposal for Advanced Data Product in Construction Code Interpretation and Research

Dear Esteemed Members,

I am writing to propose Archivy, a web application aimed at resolving a significant challenge in our industry: the complex interpretation of construction codes (building law). This challenge leads to inefficiencies and compliance risks, impacting project timelines and legal adherence.

Archivy simplifies code interpretation and research by providing a user-friendly platform that quickly delivers accurate and easy-to-understand building code information. This solution will significantly enhance compliance accuracy and operational efficiency for construction industry professionals, streamlining their decision-making processes.

Our project requires a budget of $400K to $500K, covering the development, data acquisition, and operational costs. The initial phase will focus on the 2010 Americans with Disabilities Act Design Standards, ensuring a solid, compliant foundation. We plan to complete the MVP development by Q2 2024, followed by a phase of testing and refinement.

Ethically, we are committed to the highest standards of data protection and legal compliance. The application will only host legally adopted building code sections, ensuring adherence to copyright laws and regulatory standards.

My expertise in software engineering, my experience in the architecture industry, and the team's dedication to innovative solutions will position us well to lead this project. Archivy promises to deliver a transformative tool for the construction industry, enhancing efficiency and compliance.

I look forward to discussing how Archivy can benefit the wider industry.

Warm regards,

Cristian Sotomayor,
Chief Technology Officer
Archivy.io

Part B: Project Proposal Plan

Project Summary

Problem Description:
Professionals in the construction industry are often hindered by the complexity of building codes, leading to inefficiencies and risks in compliance. This challenge slows down project timelines and compromises adherence to legal standards, necessitating an innovative solution.

Client Needs:
Our target clients, including architects, contractors, and construction professionals, require an efficient, accurate tool for interpreting these complex codes. Unlike current market solutions, which offer basic PDF-like navigation, our clients need a more intelligent, time-saving, and reliable method for code interpretation and search.

Deliverables:

1. A Python and Flask-based web application, Archivy, enhanced with ChatGPT models, a Retrieval-Augmented Generation (RAG) system, and a Pinecone vector database, ensuring high-accuracy responses.
2. Comprehensive user guide and documentation covering the application's functionalities, accuracy, implementation, and accessibility.
3. Regular updates and maintenance protocols to ensure ongoing efficacy and relevance.

Benefit Justification:
Archivy will provide precise, easily accessible information through a natural language user interface, dramatically enhancing the accuracy and efficiency of code interpretation. This transformation will significantly reduce the time spent on design compliance research and mitigate risks, thereby boosting operational efficiency and compliance in the construction sector.

Data Summary

Data Source and Collection:
The MVP will utilize the Americans with Disabilities Act Design Standards for its simplicity and uniformity. As the project progresses, we will expand to include a comprehensive database of California's construction laws and codes.

Data Processing and Management:
Our approach will use a combination of ChatGPT models, a Retrieval-Augmented Generation (RAG) system, and a Pinecone vector database, ensuring the delivery of relevant and accurate interpretations. Data will be continually updated and refined to maintain accuracy and relevance.

Data Justification:
This strategy ensures access to current and precise information, crucial for accurate code interpretation. Data anomalies will be identified and addressed through continuous testing and updates.

Ethical and Legal Concerns:
We are committed to ethical AI usage and data protection, hosting only legally adopted sections of building codes and adhering to copyright laws and regulatory standards.

Implementation

Methodology:
Our development strategy involves creating a bespoke LLM-based application tailored to construction code text data. We will employ an agile development framework, focusing on resolving technical and product issues in two-week sprints.

Implementation Plan:
The plan includes developing a user-friendly web application that integrates advanced LLMs like ChatGPT-4 and 3.5, combined with a RAG system and Pinecone vector database. This integration targets the specific need for precise, contextually relevant code interpretation.

Timeline

Milestone/Deliverable	Duration (Months)	Projected Start Date	Anticipated End Date
MVP Development	6	Q1 2024	End of Q2 2024
Alpha Testing with Architecture Professionals	3	End of Q2 2024	Q3 2024
Beta Release and Public Testing	3	Q3 2024	Q4 2024
Product Finalization and Launch Preparation	3	Q4 2024	Q1 2025
Official Launch	-	Q1 2025	-

Evaluation Plan

Verification Methods:
Development will include systematic requirement analysis, user-centric design approaches, and continuous integration/continuous deployment (CI/CD) pipelines, supplemented by regular code reviews and quality assurance testing.

Validation Method:
The project's effectiveness will be validated through an alpha testing phase with 100-200 professionals in the architecture industry, followed by a public beta release. This phased approach will gather valuable user feedback for iterative improvements.

Resources and Costs

1. **Hardware and Software Costs:**
    - Costs associated with server hosting, database management, and software licenses.
2. **Labor Costs:**
    - Estimated 18,000 hours of development time by a team of four, including the CTO, a principal machine learning developer, and two software developers.
3. **Environment Costs:**
    - Costs for deployment, hosting, and maintenance of the web application, RAG system, and databases.

This detailed plan aims to provide our clients with a cutting-edge solution for construction code interpretation, addressing their specific needs and enhancing efficiency and compliance in the construction industry.

Part C: Application

1. **The application functions as described.**
   See User Guide on page 31.
2. **A mathematical algorithm applied to data:**
   See attached source code in Archivy.zip.
3. **A "user interface."**:
   See User Guide on page 31.
4. **Three visualizations:**
   See Visualizations on page 25.
5. **Submitted files and links are static and accessible:**
   See attached source code in Archivy.zip and User Guide on page 31.

Part D: Post-implementation Report

Solution Summary

Problem Overview:
The architecture, engineering, and construction sectors have long battled with the challenges posed by traditional methods of accessing and interpreting building codes. For years, professionals have been mired in the laborious task of navigating through 500-page building code manuals or consulting overworked local building officials for clarification on ambiguous sections. This process is not only time-consuming but also inefficient, compounded by the gradual disappearance of experienced building code experts due to an aging workforce and high entry barriers deterring new talent.

Archivy's Solution:
Archivy stands as a groundbreaking solution to these persistent industry challenges. Integrating an advanced Language Learning Model (LLM), Archivy revolutionizes the way building codes are accessed and interpreted. Our application, developed with a Python and Flask-based web framework, employs ChatGPT models, a Retrieval-Augmented Generation (RAG) system, and a Pinecone vector database, ensuring rapid, accurate, and contextually relevant search results.

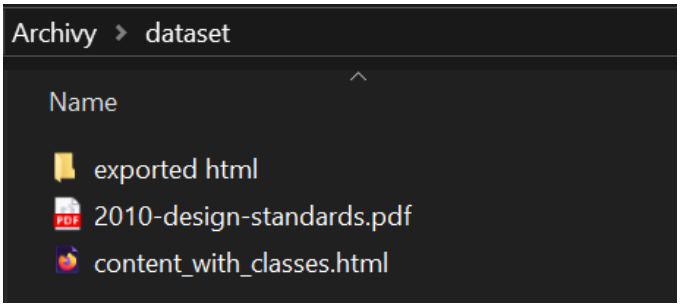
Unlike traditional PDFs and basic search applications, Archivy provides a more nuanced and efficient approach to code interpretation. Its natural language user interface allows professionals to obtain precise information quickly, drastically reducing the time spent on design compliance research. Initially focusing on the American with Disabilities Act Design Standards, Archivy is poised to expand to a comprehensive database of California's construction laws and codes, catering to a broader range of industry needs.

Moreover, Archivy's implementation addresses the looming crisis of the dwindling pool of building code experts. By simplifying code interpretation and making information more accessible, it empowers emerging professionals, enabling them to focus more on vital tasks and less on deciphering complex codes. This not only bridges the knowledge gap but also ensures a sustainable future for industry expertise.

In conclusion, Archivy is not just an application; it is a beacon of innovation for the construction industry, offering a streamlined, intelligent solution to a longstanding problem and paving the way for a new era of technological advancement in the sector.

Data Summary

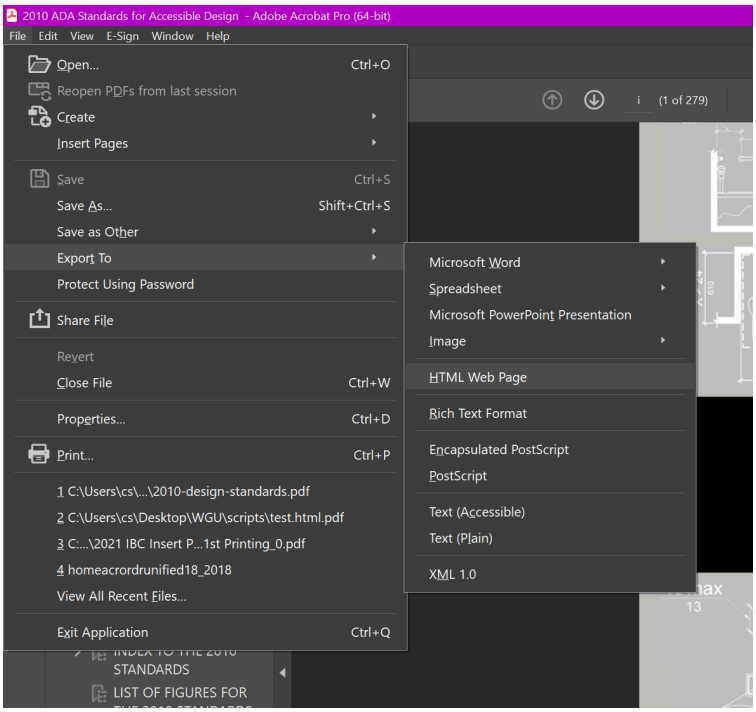
The included datasets are found within the directory *'.../Archivy/dataset'* as seen in the image below:



The PDF file *'2010-design-standards.pdf'* is the original building code file as published by the US Department of Justice at https://www.ada.gov/law-and-regs/design-standards.

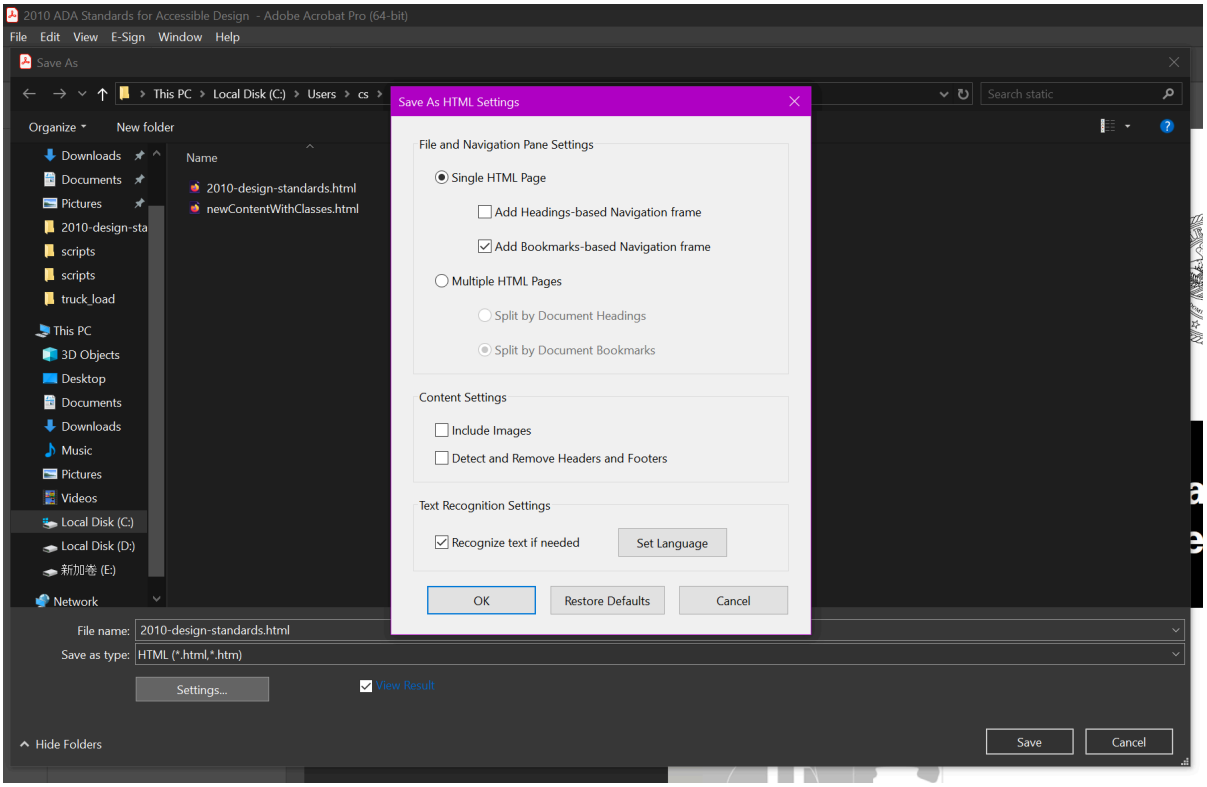
The HTML file *'content_with_classes.html'* is the curated result of processing the original PDF file within a custom script built-into Archivy in preparation for the insertion of the building code into a Pinecone vector database.

In order to curate the PDF file into the HTML file, the process begins by using Adobe Acrobat Pro to turn the PDF file into an HTML file as seen below:



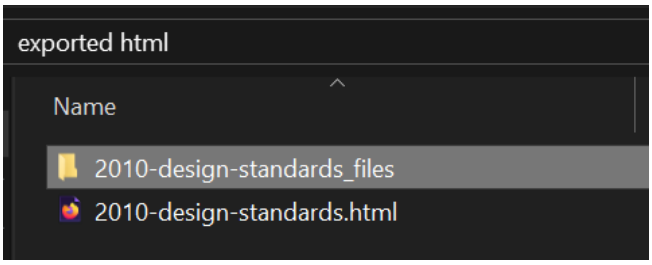
***Programmatic solutions were considered to turn the PDF into an HTML file but due to the large price required to use these APIs it was decided that using Adobe Acrobat was good enough for the development of the MVP.*

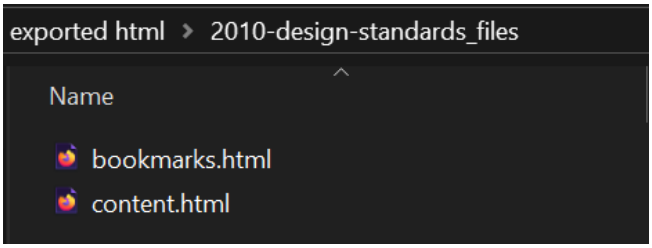
When exporting the PDF as an HTML the following settings were chosen:



***The exported files can be found in directory '.../Archivy/dataset/exported html'*

After exporting the PDF as HTML, the files below were created. However, out of all files created only two are relevant, *'content.html'* and *'bookmarks.html'*:





Once the files are created, they must be moved to directory *'.../Archivy/scripts'* to be processed by the following script found at *'.../Archivy/scripts/process_html.py'* which is used to curate the data prior to vectorization:

process_html.py > ...

```python
# Import necessary classes from BeautifulSoup
from bs4 import BeautifulSoup, Tag

# Define a function to extract sections from an HTML file
def extract_sections(html_raw, html_extracted):

    # Open the raw HTML file for reading
    with open(html_raw, 'r', encoding='utf-8') as file:
        # Parse the HTML file using BeautifulSoup
        soup = BeautifulSoup(file, 'html.parser')

    # Find all elements in the HTML file
    elements = soup.find_all()

    # Open a new file to write the extracted HTML
    with open(html_extracted, 'w', encoding='utf-8') as file:

        # Iterate over each element in the HTML
        for i, elem in enumerate(elements):

            char_num = 0

            # Check if the element is not a tuple and is a leaf Tag (no child elements)
            if not isinstance(elem, tuple):
                if isinstance(elem, Tag) and not elem.find():
                    # Check if the element has a 'name' attribute
                    if 'name' in elem.attrs:

                        elem_name = elem['name']

                        # Write closing and opening div tags with the element's name
                        file.write('</div>')
                        file.write('<div id="' + elem_name + '">')

                    # Check if the element has text and write it to the file
                    if len(elem.get_text()) != 0:
                        elem_text = elem.get_text()
                        file.write(elem_text + '\n')
        # Close the file (not necessary as 'with' statement handles it)
        file.close
```

**Image continues in next page

```python

# Define a function to add classes to the extracted HTML based on bookmarks
def add_classes(html_extracted, html_bookmarks, html_with_classes):
    # Open the extracted HTML file for reading
    with open(html_extracted, 'r', encoding='utf-8') as file:
        soup = BeautifulSoup(file, 'html.parser')
        # Find all div elements
        contElements = soup.find_all('div')

        # Open the bookmarks HTML file for reading
        with open(html_bookmarks, 'r', encoding='utf-8') as fileA:
            soupA = BeautifulSoup(fileA, 'html.parser')
            # Find all bookmark links
            bookmarksA = soupA.find_all('a', class_='bookmark')

            # Loop through each bookmark
            for bookmark in bookmarksA:
                # Extract the bookmark ID
                bookmarkA = bookmark['href'].split('#')[-1]
                label = bookmark.text

                # Loop through each div element in the extracted HTML
                for element in contElements:
                    # Check if the element ID matches the bookmark ID
                    if 'id' in element.attrs:
                        if bookmarkA == element['id']:
                            # Add a class to the element based on the bookmark label
                            element['class'] = element.get('class', []) + [label]

    # Write the modified HTML to a new file
    with open(html_with_classes, 'w', encoding='utf-8') as output_file:
        output_file.write(soup.prettify())


# Define the main function
def main():
    # Define file paths
    html_raw = 'content.html'
    html_extracted = 'content_extracted.html'
    html_bookmarks = 'bookmarks.html'
    html_with_classes = 'content_with_classes.html'

    # Call functions to process the HTML files
    extract_sections(html_raw, html_extracted)
    add_classes(html_extracted, html_bookmarks, html_with_classes)

# Check if the script is the main program and execute the main function
if __name__ == "__main__":
    main()
```

While Acrobat exports a usable HTML file, the code within the file is rather messy and contains many unnecessary elements. Therefore this script uses BeautifulSoup to parse and manipulate the exported HTML content.

The script begins with *'extract_sections()'*, this function reads the *'content.html'* file, extracts its elements, and writes them to a new file stripped of all unnecessary items. Then the *'add_classes()'* function reads this extracted content and the '*bookmarks.html'* file, and updates the classes of the extracted HTML elements with the name of the section per the based on the bookmarks HTML file, and writes the updated HTML elements to another new file named *'conten_with_classes()'*.

It is important to perform the step of adding the section name as a class attribute to the HTML elements. Once the final HTML file is inserted into the vector database the section names and bookmark numbers will be attached to corresponding section paragraphs as metadata. Doing this allows the application to retrieve relevant section information from the vector database when a user queries the application with a question

Once processed, the exported Acrobat HTML files will go from this:

```
3431          <a name="bookmark23"
3432            >Commercial facilities located in private residences.</a
3433          >
3434        </p>
3435        <p style="text-indent: 0pt; text-align: left"><br /></p>
3436        <ol id="136">
3437          <li data-list-text="(1)">
3438            <p
3439              style="
3440                padding-left: 60pt;
3441                text-indent: -18pt;
3442                line-height: 111%;
3443                text-align: left;
3444              "
3445            >
3446              When a commercial facility is located in a private residence, the
3447              portion of the residence used exclusively as a residence is not
3448              covered by this subpart, but that portion used exclusively in the
3449              operation of the commercial facility or that portion used both for
3450              the commercial facility and for residential purposes is covered by
3451              the new construction and alterations requirements of this subpart.
3452            </p>
3453            <p style="text-indent: 0pt; text-align: left"><br /></p>
3454          </li>
```

To this, which is much more easily inserted into a vector database:

```
442   <div
443     class="(b) Commercial facilities located in private residences."
444     id="bookmark23"
445   >
446     Commercial facilities located in private residences. When a commercial
447     facility is located in a private residence, the portion of the residence used
448     exclusively as a residence is not covered by this subpart, but that portion
449     used exclusively in the operation of the commercial facility or that portion
450     used both for the commercial facility and for residential purposes is covered
451     by the new construction and alterations requirements of this subpart. The
452     portion of the residence covered under paragraph (b)(1) of this section
453     extends to those elements used to enter the commercial facility, including the
454     homeowner's front sidewalk, if any, the door or entryway, and hallways; and
455     those portions of the residence, interior or exterior, available to or used by
456     employees or visitors of the commercial facility, including restrooms.
457   </div>
```

Machine Learning

Method Identification and Functionality:
Archivy incorporates a Language Learning Model (LLM) integrated with a Retrieval-Augmented Generation (RAG) system. The LLM, based on models like ChatGPT-4, excels in generating human-like text responses. The RAG system enhances this by retrieving contextually relevant information from a large dataset, in this case, building codes, to provide specific, accurate responses.

Development Process with Technical Specifics:

Data Preparation and Transformation:
Original Data Source: '2010-design-standards.pdf' from the US Department of Justice.

HTML Conversion:
Using Adobe Acrobat Pro, the PDF was converted into HTML, facilitating the subsequent processing and data manipulation.

Data Curation with Custom Scripting:

HTML Processing:
A Python script with BeautifulSoup refined the HTML data, stripping unnecessary elements and structuring the content for machine processing.

Class and ID Assignment:
This script assigned unique class and ID attributes relating to building code sections, preparing the data for effective retrieval.

Machine Learning Integration and Vectorization:

Embedding Generation:
The create_docsearch Python script used OpenAI's API to create embeddings for each text segment. These embeddings are high-dimensional vector representations of the text, capturing its semantic properties.

Pinecone Indexing:
The embeddings were indexed in Pinecone, a vector database. The command pinecone.create_index(name=index_name, metric="cosine", dimension=1536) plays a crucial role here:

create_index: This function initiates the creation of a new index in Pinecone, where the indexed data will be stored.

name=index_name: This parameter specifies the name of the new index.

metric="cosine": The cosine similarity metric is used to compare the similarity between vectors. In the context of our application, it measures the cosine of the angle between two embedding vectors, with a smaller angle indicating higher similarity. This is pivotal for retrieving the most relevant building code sections in response to a query.

dimension=1536: This specifies the dimensionality of the vectors stored in the index. Higher dimensions can capture more detailed semantic information, making the retrieval more accurate.

Retrieval-Augmented Generation System:

ChatGPT Integration:
The RAG system uses the indexed vector database to augment the responses from the ChatGPT model with precise information from the building codes database.

Real-Time Retrieval:
When a query is made, the RAG system dynamically retrieves the most semantically similar entries from the vector database, ensuring contextually relevant responses.

Technical Justification for Method Selection and Development:

Precision and Contextual Relevance:
The use of LLM with a RAG system and vector database ensures that Archivy's responses are not only accurate but contextually tailored to each query.

Efficiency in Retrieval:
The cosine similarity metric in the vector database allows for rapid and precise retrieval of information, vastly outperforming traditional search methods.

Scalability and Adaptability:
The system's scalability is underpinned by the vector database's ability to handle high-dimensional data, allowing for future expansion and updates.

Technical Innovation:
This approach represents a significant advancement in AI and ML application, specifically in handling complex legal and technical documents like building codes.

Addressing Industry Expertise Gaps:
The user-friendly interface and accurate retrieval system support professionals at all experience levels, ensuring the continuity of expertise in the industry.

In summary, Archivy's machine learning methodology is a blend of advanced NLP techniques and state-of-the-art vector retrieval systems. The technical intricacies, particularly in data vectorization and retrieval using cosine similarity in a high-dimensional space, are

key to its ability to provide accurate, efficient, and contextually relevant responses, addressing critical needs in the construction industry.

Validation

Validation Method:

To validate Archivy, we performed an analysis focused on its capacity to process specialized, technical knowledge, and their ability to accurately reference relevant building code sections, critical in fields like architecture and building code compliance. This evaluation also provided insights into the progress of AI development, particularly in handling complex, domain-specific queries, and identified current limitations and potential for future integration into professional workflows. Overall, this research offers a detailed comparison of these AI systems, underscoring their capabilities and areas for improvement, and marks a significant step in understanding and applying AI in the specialized knowledge domain of building codes.

What was tested:

Specialized Knowledge Handling:
The AI systems were tested on their ability to handle specialized, technical knowledge. The specific focus on the 2010 ADA Design Standards challenges the AIs to not only understand the query but also to retrieve and interpret detailed and specialized information accurately.

Source Referencing Capability:
A key aspect of the evaluation is the AI's ability to direct users or create citations to the correct section of the building code. This is a crucial skill, especially in professional contexts where accuracy and reliability of the source material are paramount.

Accuracy in Context-Specific Queries:
The AI systems were tested for their precision in answering context-specific queries. This includes understanding the nuances of the ADA standards and applying them to the questions correctly.

How it was tested:

The Subjects:
The testing assessed the performance of different AI systems (Archivy, ChatGPT 4, ChatGPT 3.5, Bard, and Claude) in handling queries related to the 2010 ADA Design Standards. This evaluation was crucial to understand how these AI models fare in interpreting and referencing specific sections of complex legal documents and how Archivy might benefit from using a RAG system.

The Tests:
60 Questions were compiled from the 2010 ADA Design Standards buildings codes with varying levels of difficulty. The formats were divided into multiple choice and 'fill in the blank' questions. The section from where the correct answers originate was also recorded. Every question was asked to each one of the LLMs and several things were recorded: Whether the answer was correct or not, the answer itself, whether the source was provided, and the source itself.

***It is important to understand the difference between directing and citing. Directing a user to a source material severely decreases the time spent corroborating an LLM answer. A citation, while useful, will force a user to have to research to corroborate an LLM answer.*

Metrics of Evaluation:
The number of the LLMs questions were simply measured as a percentage of the total of all questions. For example, if Archivy answered 45/60 questions correctly, its score would be 75%.

Moreover, the presence and accuracy of sources provided was measured by recording the LLMs response into the following categories;

DCS, Directed to correct source:
LLM provided a link to the specific section from source material, allowing testers to corroborate answers within application.

CCS, Cited correct source:
LLM provided a correct citation, but no link, forcing tester to search for source material to corroborate the answer.

DTGWBC, Directed to general website for building code:
LLM provided a link to an article that cites the original source material, or a link to the front page of source material, forcing testers to search for source material to corroborate the answer.

NSG, No source given:
LLM provided only an answer with no citation or link.

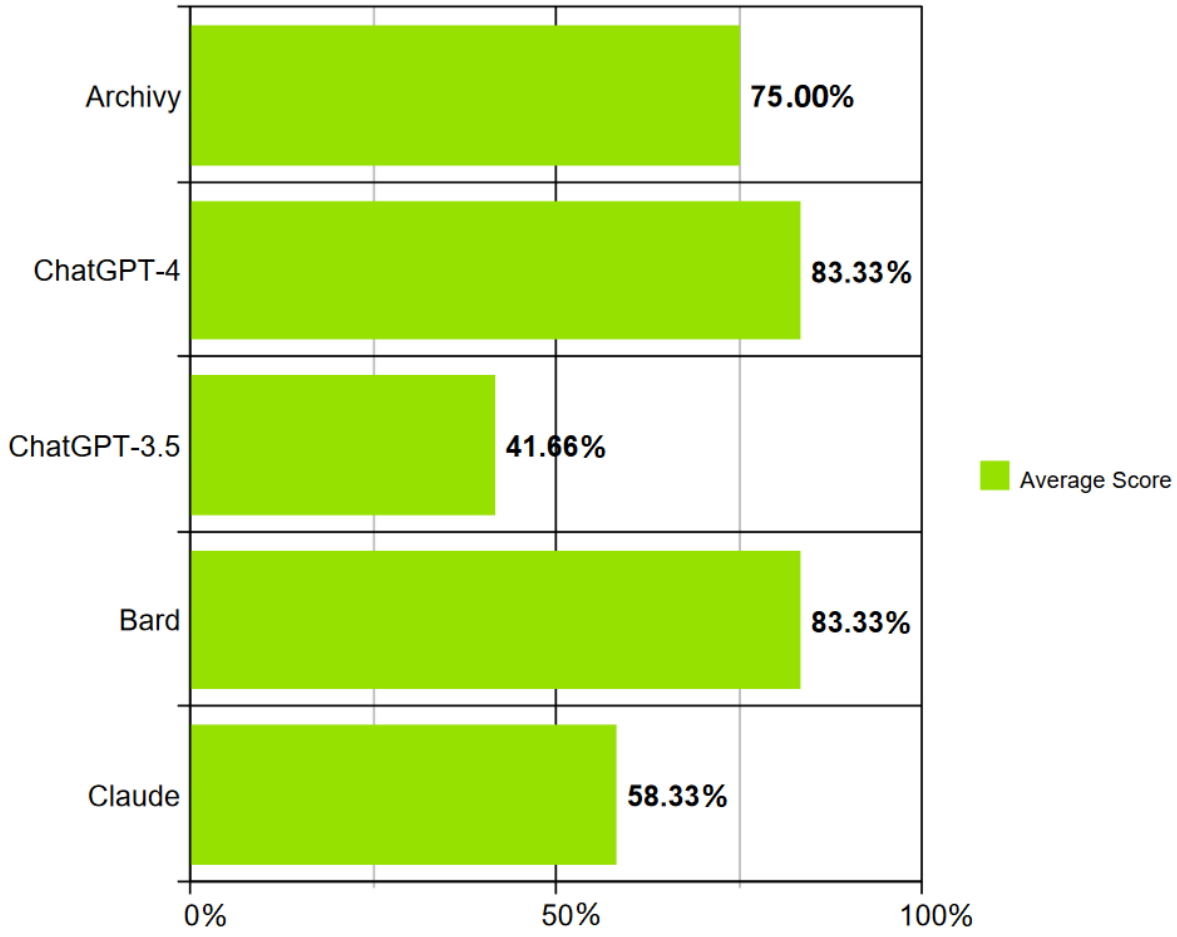
CNCS, Cited nearby correct source:
LLM provided a citation that is close only a few sections away from original source material, forcing testers to search for answers to corroborate.

CWSL, Cited wrong source:
LLM provided a citation that is wrong.

For every question asked to an LLM, its behavior was recorded depending on how it provided links or citations and was categorized into the sections mentioned above. A score would be provided for every LLM and category. For example, if Archivy provided a link to a source material for 45/60 questions that category would get a score of 75%.

Validation Results:

Answer accuracy:



To analyze the capabilities of each AI system (Archivy, ChatGPT 4, ChatGPT 3.5, Bard, and Claude) based on the provided context and data, let's consider each system individually:

1. Archivy

- **Strengths:** Archivy demonstrates a strong capability in answering questions related to the 2010 ADA Design Standards. It correctly answers most questions, indicating a good understanding of the domain-specific information.
- **Overall Evaluation:** Archivy appears to be a reliable tool for obtaining accurate answers in this specialized field.

2. ChatGPT 4

- **Strengths:** ChatGPT 4 shows a high level of accuracy in answering the questions, reflecting a strong grasp of the ADA standards.
- **Overall Evaluation:** ChatGPT 4 is a robust AI system for answering technical questions, though it may have limitations in source referencing.

3. ChatGPT 3.5

- **Strengths:** ChatGPT 3.5, while slightly less capable than its successor (ChatGPT 4), still provides correct answers to a number of questions.
- **Overall Evaluation:** A competent system, but possibly less refined than ChatGPT 4 in handling complex, domain-specific queries.

4. Bard

- **Strengths:** Bard's performance in the test suggests a reasonable understanding of ADA standards.
- **Overall Evaluation:** Bard seems to be a viable option for ADA standards-related queries but may not be the most accurate among the tested AIs.

5. Claude

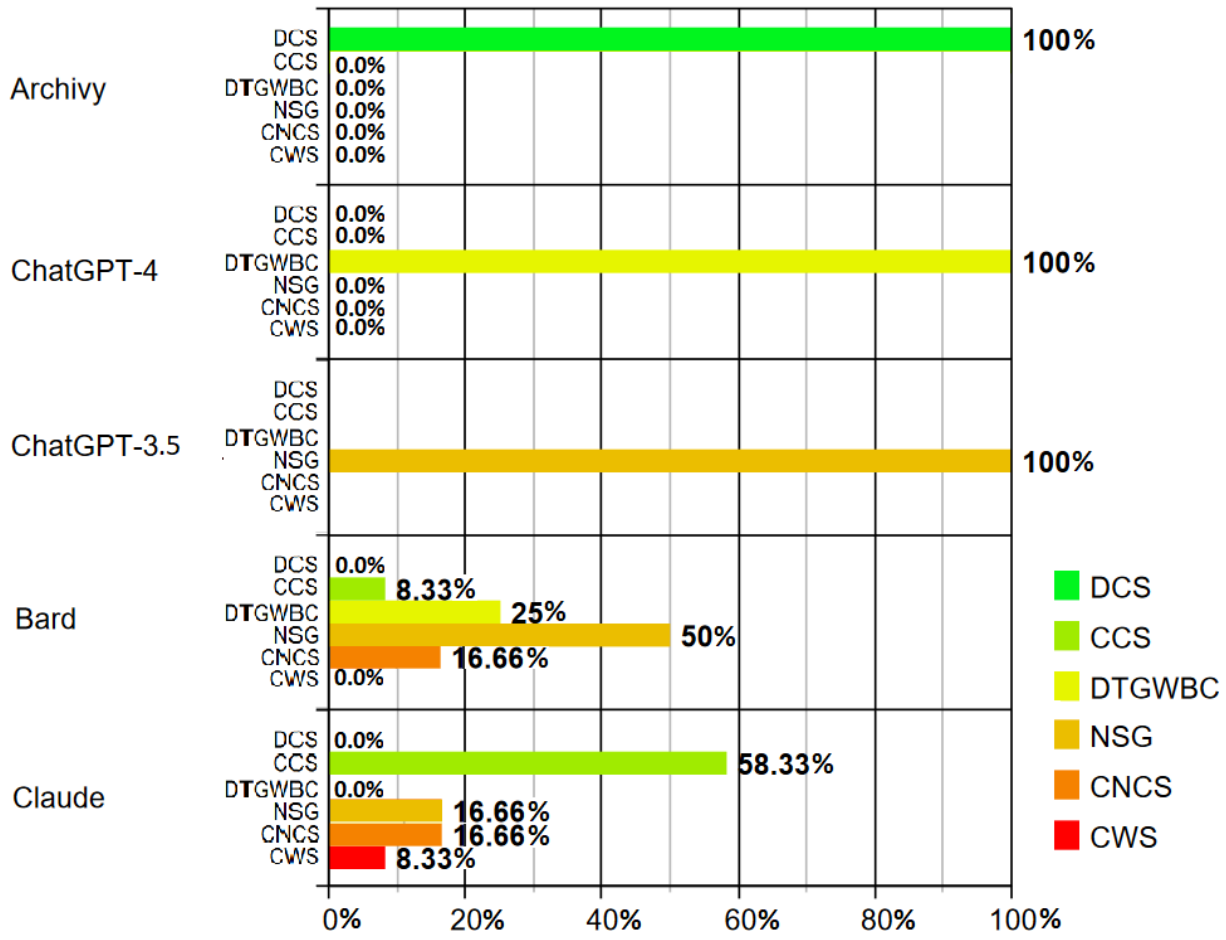
- **Performance:** Claude appears to have more limitations or challenges in accurately answering the ADA standards questions compared to other AIs.
- **Overall Evaluation:** Claude may require improvements or more specific training to match the performance of other AIs in this domain.

General Observations

- **Domain-Specific Knowledge:** All AIs show some level of capability in handling specialized, technical queries, with varying degrees of success.
- **Potential for Improvement:** Continuous advancements in AI could lead to improved performance in future versions, especially in the context of specialized knowledge domains like legal or building standards.

In summary, while each AI system demonstrates a certain level of proficiency in answering technical queries related to the 2010 ADA Design Standards, there are noticeable differences in their accuracy and potential ability to direct users to specific source material.

Directing and citation accuracy:



Archivy

- **Directing:** Archivy excels at directing to the correct section, scoring. This implies that it consistently took the user directly to the source material, which is the most desirable outcome.
- **Citing:** Archivy also had accurate citations, indicating it mentioned the exact part of the code in the answers.
- **Overall Evaluation:** Archivy stands out as the most capable AI in the test for both directing to and citing the exact sections of the ADA standards, which is extremely valuable for professionals needing to access specific regulatory information.

ChatGPT 4

- **Directing:** ChatGPT 4 did not direct users to the correct section at all. It seems to lack the ability to take the user directly to the source material.
- **Citing:** However, it was able to cite the correct general website in all cases, meaning it provided section numbers or references within its responses, which could still be useful for users.
- **Overall Evaluation:** ChatGPT 4 is effective at citing general information, but it does not direct users to the specific source material. This indicates a strong general knowledge base but suggests an area for improvement in directly accessing specific documents.

ChatGPT 3.5

- **Directing:** ChatGPT 3.5 did not score in either directing to the correct section or to general websites.
- **Citing:** This version did not cite any source at all, which suggests it was unable to provide users with any specific references or guidance on where to find the ADA standards.
- **Overall Evaluation:** ChatGPT 3.5 appears significantly limited in both directing and citing, indicating a possible area where advancements in AI, such as the improvements seen in ChatGPT 4, are particularly noticeable.

Bard

- **Directing:** Bard occasionally directed to general websites, but did not direct to the correct section.
- **Citing:** It cited the correct source about half of the time, so while it may not guide the user to the material, it often provides section numbers or references.
- **Overall Evaluation:** Bard shows a moderate ability to cite relevant sections but struggles with directly guiding users to the source material, indicating a partial capability in assisting users with finding specific information.

Claude

- **Directing:** Claude rarely directed to the correct section and occasionally directed to general websites, showing limited capability.
- **Citing:** It cited the correct source more than half the time, which means while it doesn't often provide direct access, it does frequently include the correct section references in its responses.
- **Overall Evaluation:** Claude shows a reasonable ability to cite correct sources but has room for improvement in directly guiding users to the ADA standards.

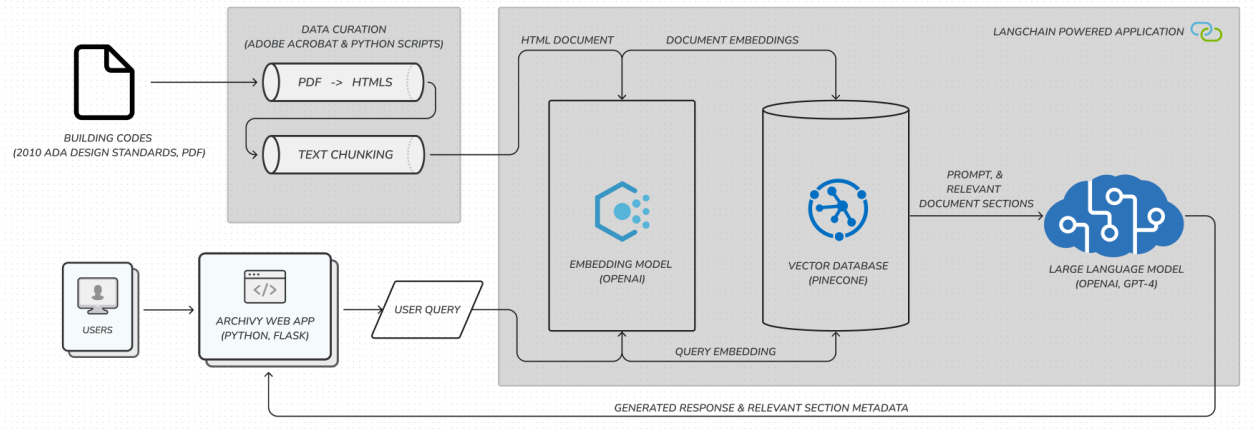
Summary of Updated Analysis

- **Archivy:** Highly proficient in both directing and citing, offering comprehensive guidance for accessing and referencing ADA standards.
- **ChatGPT 4:** Effective at citing but lacks the capability to direct users to specific source material.
- **ChatGPT 3.5:** Shows significant limitations in both directing and citing, indicating an area where newer AI models have improved.
- **Bard:** Moderately effective at citing the correct sources, with limited directing ability.
- **Claude:** More effective at citing than directing, with some capability to reference correct sources.

In professional contexts where referencing and accessing specific regulatory information is crucial, Archivy's performance indicates it could be the most valuable tool among those tested. Meanwhile, ChatGPT 4 and Claude's ability to cite, but not direct, might still serve users who are capable of navigating to the source material themselves. Bard's moderate performance and ChatGPT 3.5's limitations highlight the varying levels of maturity and specialization among AI systems in handling tasks that require precise sourcing and citation.

Visualizations

App Architecture and Data Flow



***This image depict Archivy's RAG (retrieval augmented generation) architecture and the flow of data curation, embedding, storage, interpretation and generation.*

Data Preparation and Transformation:

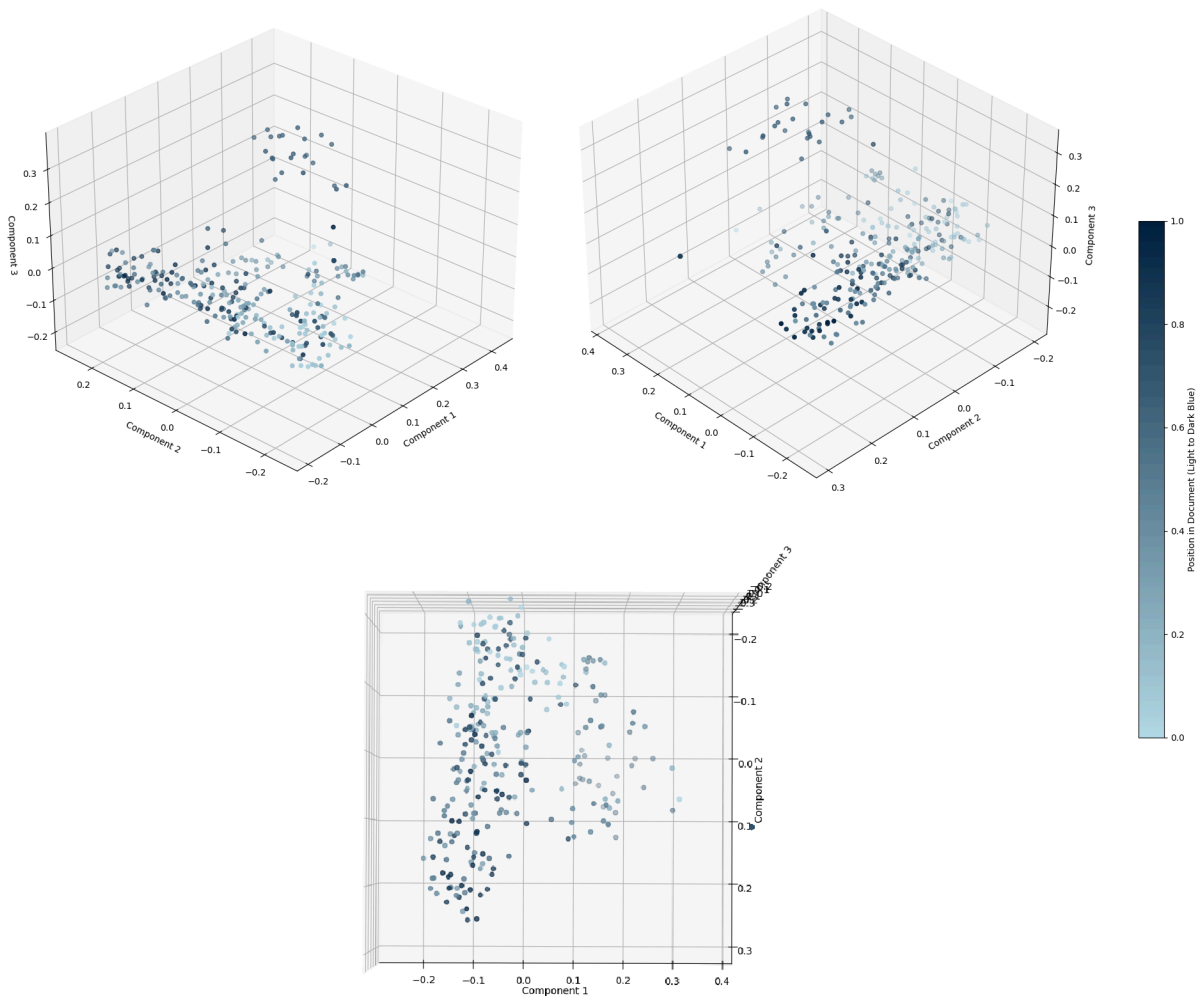
- The process starts with the '2010 ADA Design Standards' PDF document, which serves as the foundational dataset for the application. The document contains complex and detailed building codes which are not readily machine-readable.
- The transformation of this PDF into HTML format is a critical step. By using Adobe Acrobat Pro, the application converts the document into a series of HTML files. HTML is a more accessible and manipulable format for web-based applications and allows for the next stage of processing to be more effective.

2. **Data Curation with Custom Scripting**:
   - Once in HTML format, the data undergoes a curation process. A specially crafted Python script, utilizing the BeautifulSoup library, parses the HTML. This parsing is not a trivial task; the script must intelligently strip away unnecessary HTML elements that do not pertain to the core content, such as stylistic elements or scripting tags, which might interfere with data processing.
   - As the script parses the HTML, it performs a crucial operation called 'Text Chunking.' This operation involves breaking down the text into coherent, standalone sections that align with the different parts and provisions of the building codes. This chunking is vital for creating distinct data entries that the application can later retrieve individually.
   - The Python script goes further by assigning unique class and ID attributes to these chunks. These identifiers are meticulously correlated to the building code sections. This step ensures that each section of the building code can be individually indexed and retrieved, forming a bridge between the raw data and the retrieval system.

3. **Machine Learning Integration and Vectorization**:
   - The data, now curated and structured, is ready for vectorization. The create_docsearch Python script interacts with OpenAI's API to transform each

text chunk into a numerical vector, known as an embedding. These embeddings are crafted to encapsulate the semantic meaning of the text, distilling the complex legal language of the building codes into a form that a machine learning model can understand and utilize.
    - The embeddings are indexed in Pinecone, a vector database engineered for such high-dimensional data. The indexing process involves the create_index command, which is parametrized to use cosine similarity. This choice is deliberate, as cosine similarity excels at measuring the orientation (and thus the similarity) between vectors in a high-dimensional space.
4. **Retrieval-Augmented Generation System**:
    - At this stage, the application is prepared to handle user queries. When a query is inputted through the web app interface, the system uses the same embedding process to convert this query into a vector.
    - It then performs a real-time search against the indexed vectors to retrieve the most relevant building code sections. The cosine similarity metric ensures that the search results are semantically aligned with the user's query.
5. **User Interaction**:
    - The interface for user interaction is a web app built using Python and Flask. Users submit their queries through this interface, and the back-end system, powered by the data curation and vectorization processes, provides accurate, relevant, and contextually rich responses.

In summary, the data preparation and curation stages are foundational to the system's operation. They transform the raw building codes into a refined, structured, and query-able dataset, enabling the application to deliver precise information from the vast building codes database in an efficient and user-friendly manner.



***This image was produced with script 'Visualizer.py' found in directory ...\Archivy\vector_db_visualizer\visualizer.py*

The process of dimensionality reduction is a cornerstone in analyzing high-dimensional data, such as textual content from documents. A prime example of this is the visualization of text segments from the '2010 ADA Design Standards' PDF in a 3D scatter plot. This method maps the intricate structure of the document into a simplified three-dimensional space, thereby offering a clear and concise view of the content's complexity.

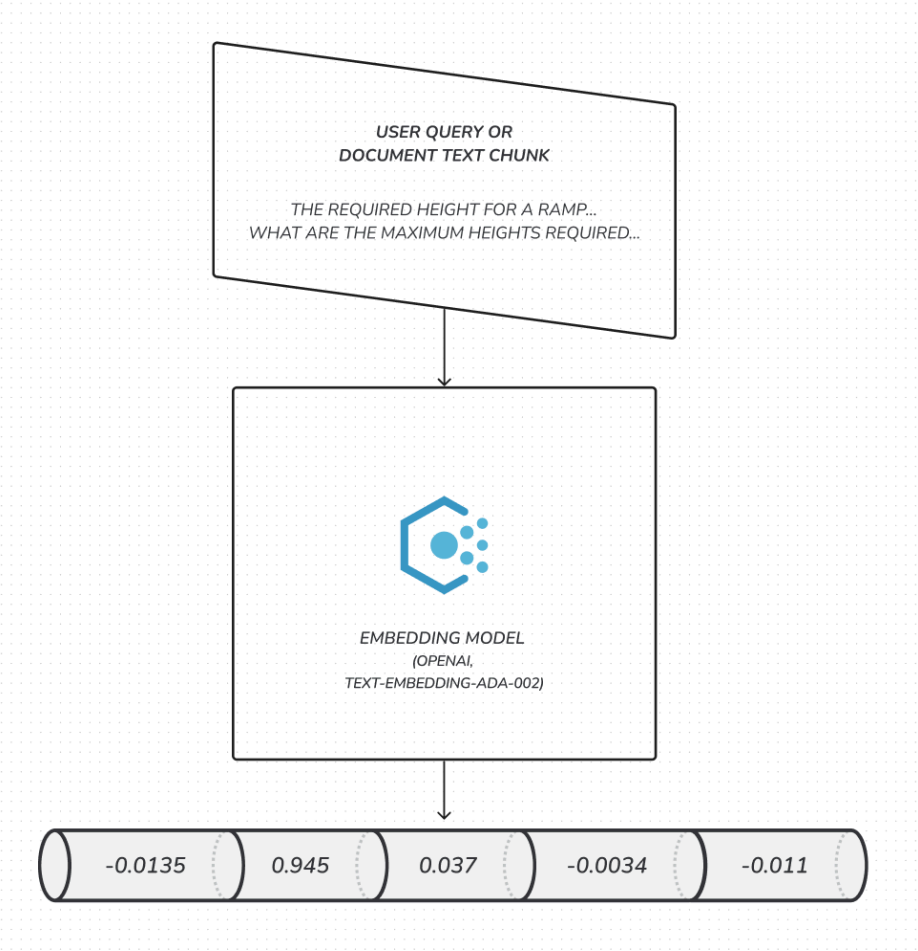
In this scatter plot, a color gradient ranging from lighter to darker shades represents the sequential flow of the document. The lighter colors typically mark the beginning, while the darker hues denote the end. Clusters within this plot indicate sections with semantically similar text, perhaps due to the recurrence of specific technical terms, themes, or standardized language.

On the other hand, isolated points or sparser areas might highlight unique content or segments that diverge from the main themes. This visual representation is instrumental in swiftly identifying patterns, trends, or anomalies, enhancing the efficiency of search and retrieval processes.

The initial step in this analytical journey involves the meticulous exploration and preprocessing of the text. This stage includes cleaning the data, normalizing text, and converting it into numerical vectors that encapsulate its semantic essence. Techniques like TF-IDF or word embeddings are common for such vectorization. Subsequently, algorithms like PCA (Principal Component Analysis) are applied to reduce the dimensionality of the data. This reduction is crucial in maintaining a balance between a manageable representation of the data and the preservation of vital information. It brings to light the relationships and distinctions within the document's content, serving as a tool to detect topic clusters or outliers, which might indicate unusual or unique elements.

For enhanced vectorization and to improve search and retrieval capabilities in the future, more sophisticated techniques such as neural embeddings can be employed. These methods delve deeper into linguistic contexts. Additionally, incorporating algorithms that consider non-linear relationships, like t-SNE or UMAP, could offer more intricate representations of textual data. This approach is particularly beneficial for complex documents where linear models like PCA might not adequately capture the nuances of the content.

The Embedding Process



The TEXT-EMBEDDING-ADA-002 model stands as a sophisticated embodiment of the latest advancements in the field of embedding models, which are integral to the functionality of Retrieval-Augmented Generation (RAG) systems. This model translates words into vectors through a process that encapsulates the multidimensional nature of language into a format that can be processed computationally with remarkable efficiency.

The process begins with the assimilation of a corpus—a substantial and diverse collection of text data—from which the model learns the contextual co-occurrence of words. Through a series of neural network layers, TEXT-EMBEDDING-ADA-002 analyzes the corpus and identifies patterns in the usage and placement of words. The core of this model's learning algorithm involves predicting words in a context or finding words with similar contexts, which teaches the model about the subtle relationships between different terms in the language.

Each word is then assigned a vector in a continuous, multidimensional space. The dimensions of this space are not arbitrary; they are constructed in such a way that they capture a range of semantic and syntactic properties. For instance, words that are often used in similar contexts or

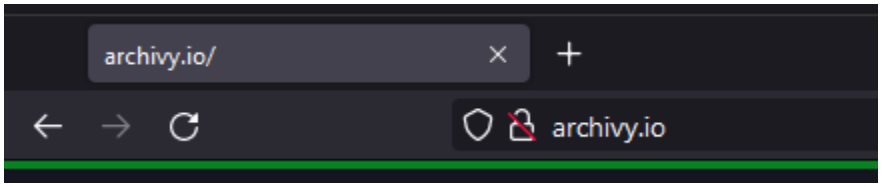
share semantic similarities will have vectors that are closer together in this vector space. The high-dimensional vectors that traditionally represent words (often thousands of dimensions corresponding to the size of the vocabulary) are reduced to a more manageable size—typically in the hundreds—while preserving these linguistic relationships.

The model's architecture, possibly involving transformer mechanisms, allows each word's vector to be influenced by the entire context in which it appears, rather than just local usage, enabling a deep understanding of language structure and meaning. The vectors produced by TEXT-EMBEDDING-ADA-002 thus embody not just the individual words but also their interconnectedness with the broader language framework.

In RAG systems, the vectors generated by TEXT-EMBEDDING-ADA-002 are utilized in the retrieval process to match queries with the most relevant information in a database. The numerical nature of the vectors allows for sophisticated similarity calculations, such as cosine similarity or Euclidean distance, to determine the relevance of database entries to a given query. This translation from words to vectors is not a mere transformation but a rich encoding of language that supports complex information retrieval tasks.

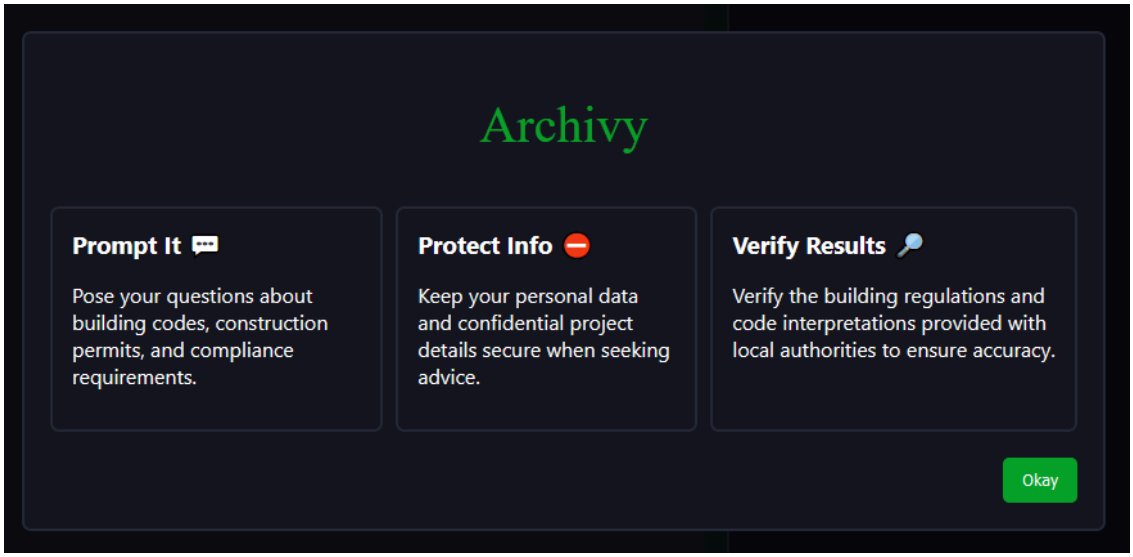
User Guide

1. With Archivy there is no need for downloads or installations, simply insert this url into your browser http://archivy.io:



***Note that your browser might warn you of an insecure connection. This is normal as we are still working on obtaining an https safety certificate*

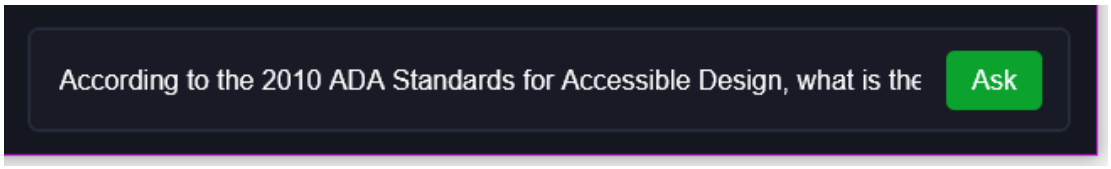
2. You will be prompted with a warning about using AI products safely. Read it carefully and if you agree. Click OK.



3. At this point you can begin interacting with Archivy's chatbot. To begin you can ask it a simple question like this:

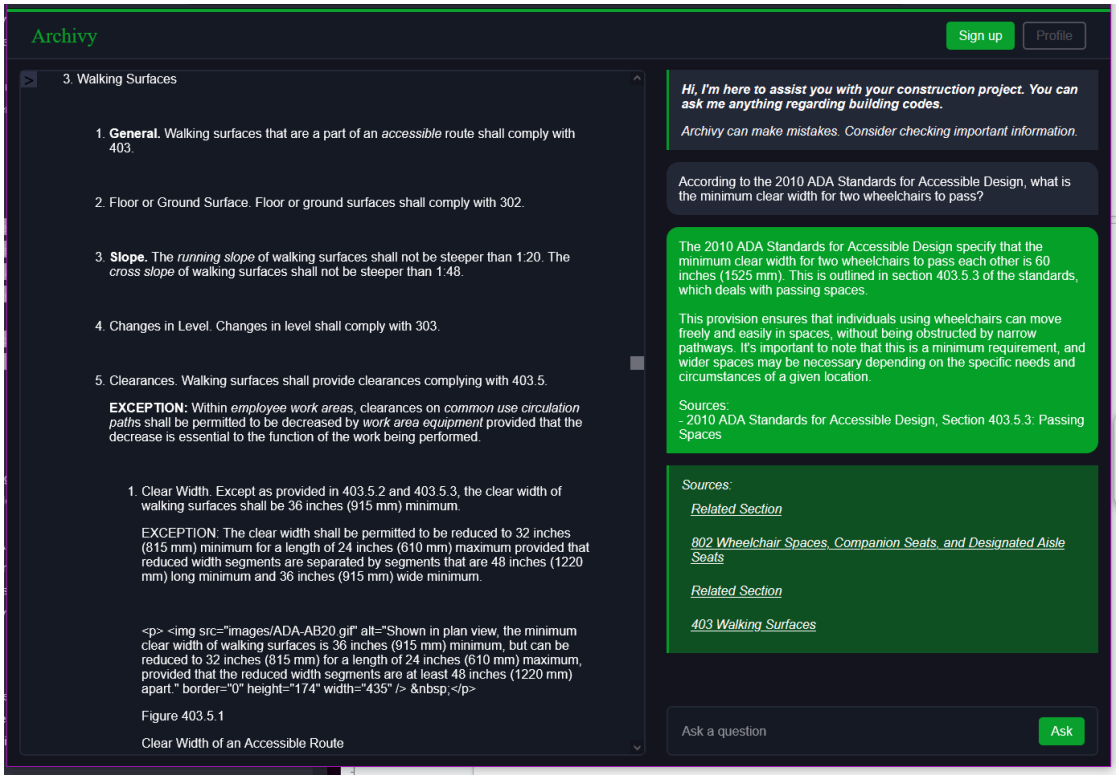
   According to the 2010 ADA Standards for Accessible Design, what is the minimum clear width for two wheelchairs to pass?
   Answers: a) 42 inches, b) 48 inches, c) 60 inches, d) 72 inches
   Correct Answer: c

   You can copy the question and the options and pasted them into the text box and press 'Ask':



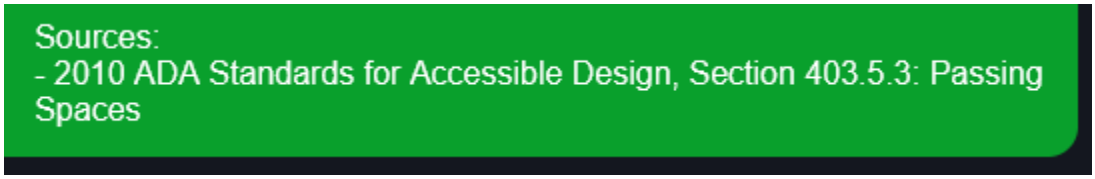
   ***Make sure not to include the correct answer when copying the question. If you do this there is a chance the LLM will know the answer from your prompt.*

   When you receive your answer your screen will look like this:



   At this point you can read your answer and check if Archivy answered the question correctly.

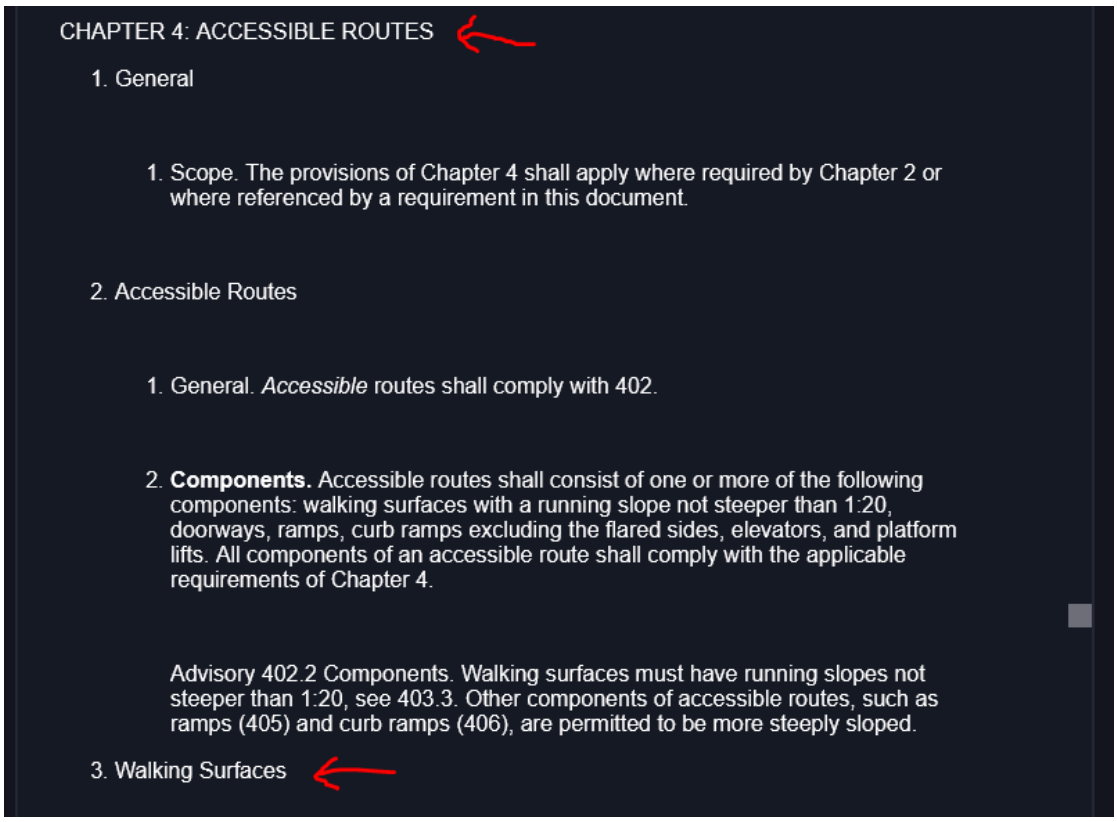
4.Furthermore, if you would like to view the answer within the embedded text. Look at the citation in your response.:



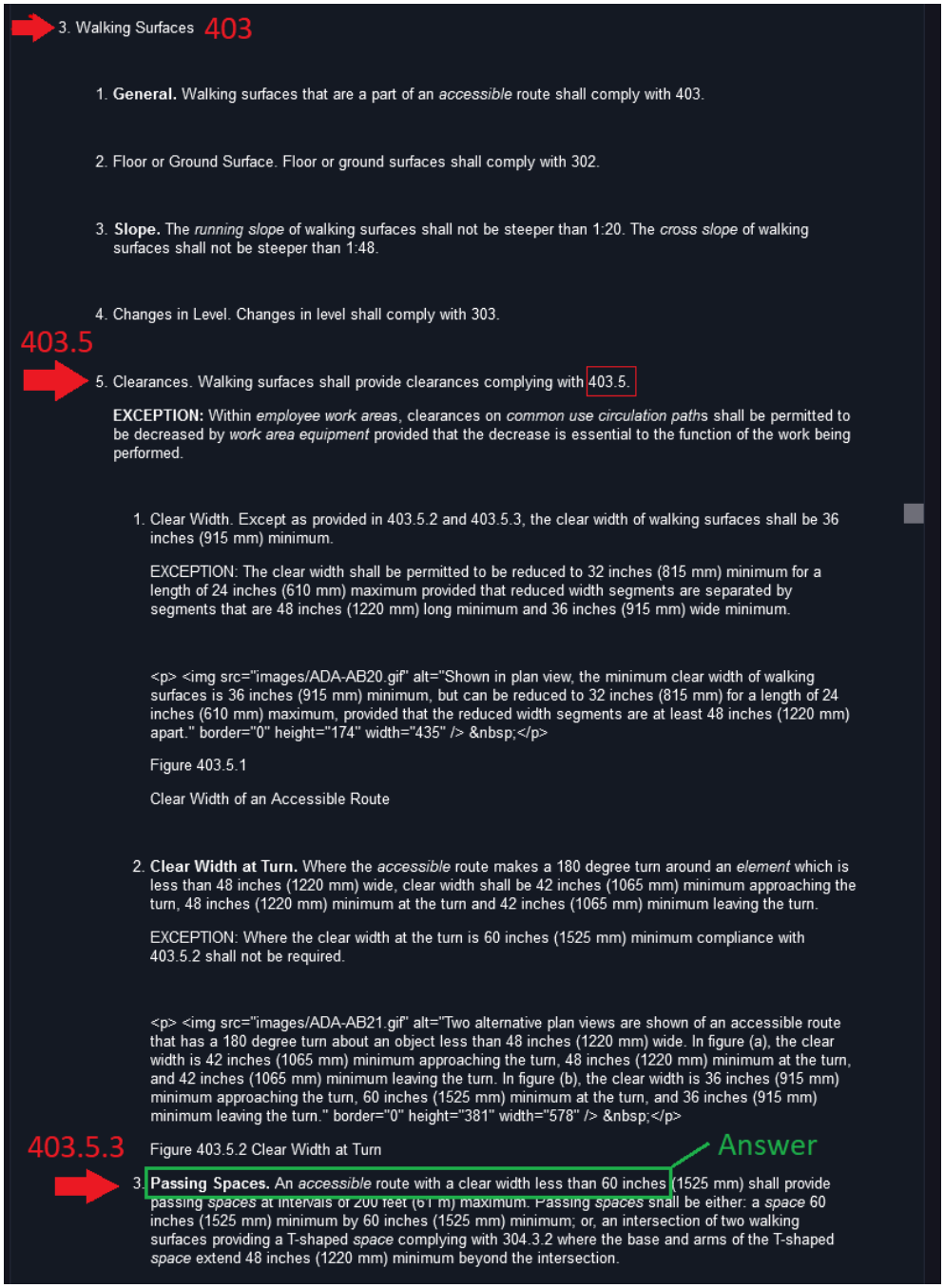
**Section 403.5.3*

Based on the section cited, go to the linked sections below and click the most relevant:

This will take you to the top of section 403 in Chapter 4:



Once here use the number 403.5.3 and scroll down to find your relevant subsection and you will find the answer. Every number after a dot is a nested subsection:



That is all! After this feel free to ask the chatbot more questions.