Software Design Document

Version: 1.0
Team: INSIGHT
Sponsor: Mike Taylor
Faculty Mentor: Scott Larocca
Team Members:
Joshua VanderMeer, Michael Vertin,
Aidan Hebert, Forrest Hartley

Date: 1/7/2025

Table of Contents

1 Introduction

Northern Arizona University has long prided itself on its vast and diverse history. Ensuring that this history is not only documented and preserved, but readily accessible for any interested parties is an integral part of historical preservation. Northern Arizona University's Cline library has long acted as this intermediary for these interested parties and desired information, facilitating any access to persevered historical information. In the recent decade, this endeavor and responsibility have expanded into the digital era of preservation, forcing documents and images to be scanned and digitized for optimal and secure preservation. This task has largely been delegated and maintained by the SCA (Special Collections & Archives) within the Cline Library, who store, maintain, and facilitate access to any digitized content. While this step is beneficial to the longevity and accessibility of this historical information, it is merely a single step towards this goal. Being able to traverse, sort, and efficiently locate information that has since been digitized is arguably an equally important step toward this goal. Ensuring that a proper tool is in place to assist with the daunting task of large-scale preservation will greatly improve the accessibility, accuracy, and overall knowledge regarding what is persevered.

To achieve this lofty goal, our team along with our sponsor Mike Taylor, aim to devise, develop, and implement an image similarity search tool for traversing and sifting through the SCA archives. This tool aims to retrieve an image from our end user, process and compute said image, leaving only similar images for the user to see. This heavily differs from the existing approach as this is not reliant on manually tagged images that could be subjected to inaccurate tagging, missing tags, or not being tagged altogether. This new result aims to bring more accurate and concrete information to a user who may be searching for something specific while mitigating instances of information being missed due to slight variations in the manual tagging process.

Overall, we feel our newly devised method for accurately searching and retrieving information from the SCA will greatly benefit any interested individuals who would like to access historical information. We feel these improvements will be accomplished and perceived by our users through our implementation of some recent technologies. These technologies are our use of image embeddings or "Vectorization", which helps us quantify and understand the contents of an image the end user provides. This then provides us with a baseline for comparison. This comparison is then possible through our use of cosine similarity, which will help us curate our "list" of similar images to then be presented to our end user cleanly and intuitively.

2 Implementation Overview

2.1 Introduction

In this section, we aim to introduce the general idea of our implementation and begin to understand how this implementation will be built and utilize existing technologies. Additionally, we will cover more specifically how each of these existing technologies will serve as a fundamental pillar for our implementation.

2.2 General Implementation

For our implementation, we intend to develop an image similarity searching tool to search through the SCA's large historical collection. Through this development and implementation of said tool, we intend to greatly improve ease of use, accessibility, and interactions with accessing historical information from the SCA's vast collection. To ensure these qualities and improvements are met, we intend to leverage some existing technologies and approaches. This begins with how the user will search for content. With this tool, the user will provide an image of something that they wish to see more of. Once provided, we will process the image and compute what is called a vector embedding. This embedding will give us a baseline for future comparisons and allow us to compare and subsequently retrieve other images that possess similar vector embeddings. Once this comparison has been completed, the user will then be shown any images that have been deemed to be similar to the SCA.

2.3 Technologies & Approaches

User Input

Our implementation begins with our user and ensuring a clean process for collecting the initial image. This will be accomplished by restricting what types of files and data can be uploaded and subsequently interpreted by our tool. This is done mainly to ensure incorrect or malicious files are not uploaded as they will not yield any useful results. Additionally, we feel this approach will allow for the vectorization and comparisons to be more consistent and hopefully more accurate as it only must account for a specific type of comparison.

Image processing & storage

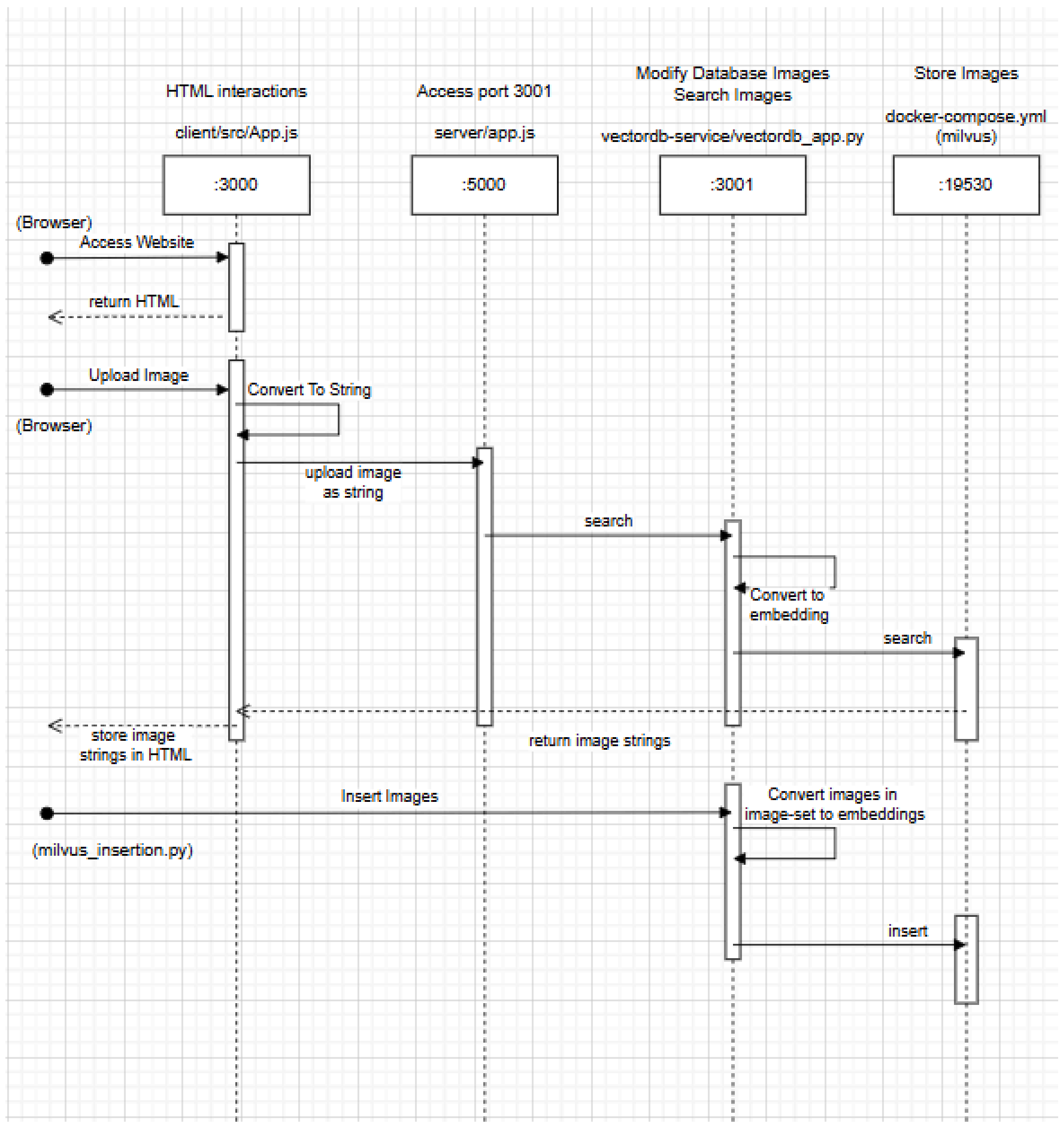
Once we have successfully collected our input image from our user, we can begin the process of processing and comparing the image. This is accomplished through a process known as vectorization. This process utilizes a technology known as TensorFlow. TensorFlow will yield what is known as vector embedding. This embedding can be understood as a computer's interpretation of an image and its contents. Once we have this embedding, we are able to begin the comparison process. The comparison process involves a technology called Milvus and a mathematical concept called cosine similarity. Milvus acts as the database where the already computed vector embeddings representing SCA's images are stored. These stored embeddings will allow us to efficiently compare images and return only images deemed similar. Cosine similarity is how this comparison between vector embeddings will occur.

Final Results & UI

Once the interpretation and comparison process has been completed. We now must serve the computed results to the user in a meaningful and intuitive manner. This will be accomplished through the implementation of a simple yet powerful frontend powered by popular Web 2.0 frameworks React and Express.

3 Architectural Overview

Diagram for High-Level System Architecture:



High‑Level View

Sequence of Interactions

1. **User (Browser)**
    - Accesses the website (front‑end served on port 3000).
    - Uploads a Base64 image to the server.
    - Displays returned search results as image references within the web page.
2. **Front‑End (client/src/App.js, Port 3000)**
    - Provides a UI for uploading images and viewing results.
    - Converts uploaded images to Base64 before sending them to the Node.js server (app.js).
3. **Back‑End (server/app.js, Port 5000)**
    - Receives the image string from the front‑end.
    - For a search request: pass the image string to the vector database service (port 3001).
    - Receives the search results as image references from the vector database service, then relays them back to the user's browser.
4. **Vector Database Service (vectordb_service/vectordb_app.py, :3001)**
    - Accepts incoming requests from the Node.js server (search, insert, etc.).
    - **Convert to embedding**: uses Hugging Face, a machine learning model to transform the uploaded image into a vector representation.
    - **Search**: compares the new vector against existing embeddings in Milvus using a cosine similarity measure.
    - **Insert**: supports a batch insertion mechanism (triggered by scripts such as milvus_insertion.py) to convert image sets into embeddings and store them in Milvus.
5. **Milvus (docker‑compose.yml, Port 19530)**
    - Stores vector embeddings and associated metadata.
    - Performs similarity searches using specialized indexes.
    - Returns nearest neighbor results as references to the vector database images.
6. **Batch Insertion (milvus_insertion.py)**
    - Used to insert large sets of new images into Milvus.
    - Passes images to vectordb_app.py, which converts them to embeddings and then calls Milvus to store the new vectors.

Architectural Components

A. Front‑End (User Interface)

- **Responsibilities & Features:**
  - Provides a web interface for accessing the system.
  - Enables users to select and upload images, which are then converted to Base64 for network transfer.
  - Renders search results in a user‑friendly manner, displaying image thumbnails of the most similar/dissimilar results.
- **Communication Mechanisms:**
  - Communicates with the Node.js server via HTTP requests with Axios.
  - Receives HTML/CSS/JS content, then dynamically updates the UI with the returned image references.

B. Node.js Server (Primary API Gateway)

- **Responsibilities & Features:**
  - Acts as the main gateway between the front‑end and back‑end services.
  - Routes requests for searching or inserting images to the appropriate internal endpoint in the vector database service.
  - Can perform preliminary validation or formatting before passing data onward.
- **Communication Mechanisms:**
  - Runs on port 5000, listening for requests from port 3000 (front‑end).
  - Uses RESTful routes or similar patterns to forward data to the vector database service.

C. Vector Database Service (Embedding Generation & Search)

- **Responsibilities & Features:**
  - **Embedding Generation**: Converts Base64 image strings into vector embeddings via Hugging Face. This ensures a consistent representation across all images.
  - **Searching**: Compares query embeddings to those in Milvus using similarity metrics. Returns top matching image references to the Node.js server.
  - **Insertion**: When triggered (e.g., by mulvus_insertion.py), it batch‑processes new images, generating embeddings and inserting them into Milvus.
- **Communication Mechanisms:**
  - Runs on port 3001, and utilizes RESTful endpoints such as search, insert, and clear to access the existing image embeddings for images from the SCA database
  - Interacts with Milvus over its native API (port 19530) for various operations on vector embeddings.

D. Milvus (Vector Store)

- **Responsibilities & Features:**
    - Serves as the primary database for vector embeddings.
    - Implements high‑performance nearest neighbor search with Inverted File (IVF) indexing to handle large‑scale vector data.
    - Allows for scalable storage and fast retrieval of image embeddings.
- **Communication Mechanisms:**
    - Exposes a service port (19530) for direct interaction with vectordb_app.py.
    - Runs inside a Docker container as defined in docker_compose.yml, making it straightforward to deploy alongside other services.

E. Batch Insertion (milvus_insertion.py)

- **Responsibilities & Features:**
    1. Provides an offline or on‑demand mechanism for adding large sets of new images such as those which will be expected from the SCA database
    2. Calls endpoints in vectordb_app.py to generate embeddings and store them in Milvus.
    3. Helps keep the vector store in sync with newly acquired images without manual user intervention.

Architectural Styles & Influences

1. **Client‑Server & Service Orientation**
    - The user interface is the client, communicating with a Node.js server, which in turn routes calls to a separate Python service. Milvus forms its own service for vector data operations.
2. **Modularity**
    - The front‑end, Node.js server, and vector database service operate as independently deployable components, each handling a distinct set of responsibilities. Milvus runs separately as a dedicated vector DB system, reinforcing modularity and scalability.
3. **Layered Approach**
    - **Presentation Layer**: The React front‑end (port 3000) and Node.js gateway (port 5000).
    - **Application Logic Layer**: Python service (port 3001) for embedding generation and searching logic.
    - **Data Layer**: Milvus (port 19530) for persistent storage of vectors and references.
4. **Data Pipeline**

- The system follows a clear flow: user uploads → server receives → vector database service embeds → Milvus searches → results return to the user. This same pattern is applied in bulk insertion to embed and store new images in the vector DB.

4 Module and Interface Descriptions

4.0 Introduction

This section provides a detailed breakdown of the key modules and interfaces that make up the system. Each component plays a crucial role in the overall functionality, from storing and retrieving images to processing and searching through the database. The modules are designed to work together seamlessly, ensuring efficient image retrieval and user interaction. The following subsections will describe the structure, responsibilities, and services provided by each module, including the image database, vector database, client, server, and user interface.

4.1 Image Database (ContentDM)

Description

The image database serves as a data source for the system. Specifically for our system, the image database is hosted in an EC2 bucket, containing a large set of images from the SCA's collection. During the initial launch of the system, the images in the image database will be sent to the server, where they will be stored as embeddings. As images are added to the SCA's collection, those images will also be represented in the vector database. At any time, the images that the searchers view will come from the SCA's database.

UML Diagram

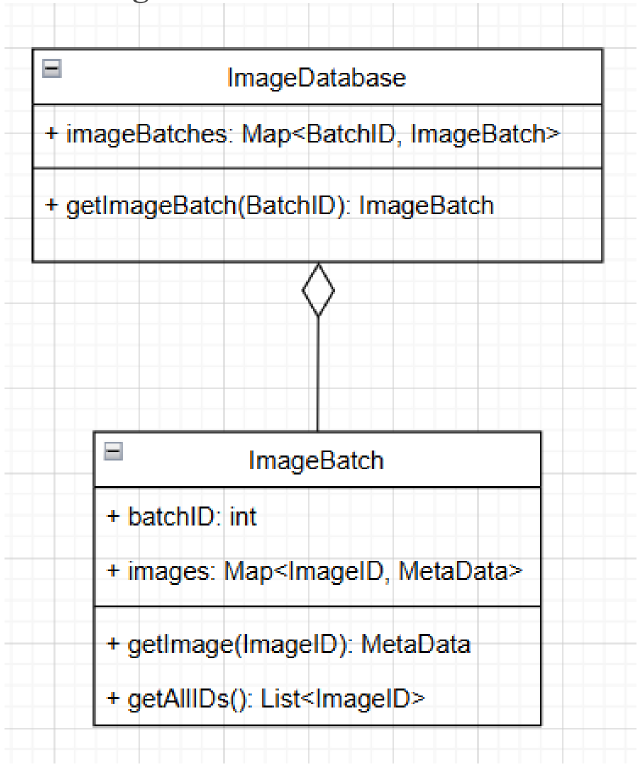

ImageDatabase
+ imageBatches: Map<BatchID, ImageBatch>
+ getImageBatch(BatchID): ImageBatch
ImageBatch
+ batchID: int
+ images: Map<ImageID, MetaData>
+ getImage(ImageID): MetaData
+ getAllIDs(): List<ImageID>

Image Database Details

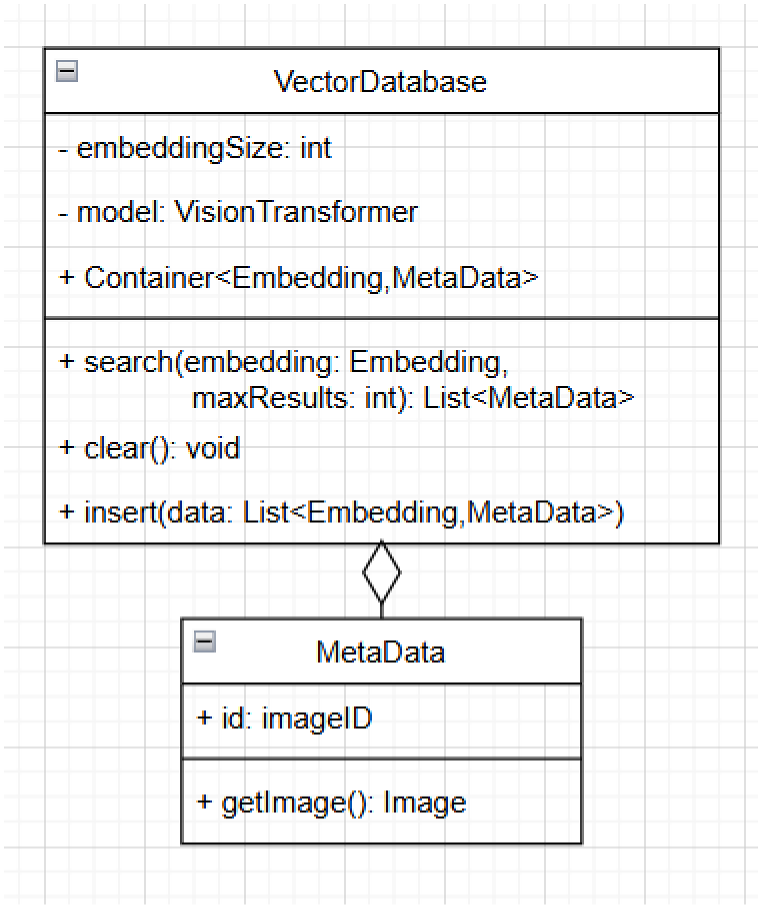
1. **getImageBatch:**
    - **Description:** This method is used to access a batch of images from the SCA's database associated with the given ID.
    - **Return Type:** ImageBatch
    - **Service:**
        - Take the given ID
        - Find the batch of images associated with the ID
        - Return an ImageBatch object representing the batch of images
2. **getImage:**
    - **Description:** This method is used to get data representing an image from an image batch given the ID for the image.
    - **Return Type:** MetaData
    - **Service:**
        - Take the given ID
        - Search through metadata in each image of the batch
        - Select the metadata with an ID that matches the inputted ID
        - Return the metadata as an object
3. **getAllIDs:**
    - **Description:** This method is used to access the identity of every image existing in the image batch
    - **Return Type:** List<ImageID>
    - **Service:**
        - Search through all available metadata in the batch
        - Extract the IDs associated with each metadata object
        - Return all IDs that were found in the process

4.2 Vector Database (Milvus)

Description

The vector database is the part of the system that contains the AI interpretations of the images in the image database. Its role is to receive embeddings and metadata from the image database and perform searching algorithms to quickly identify images similar to the ones it is provided.

UML Diagram


VectorDatabase
- embeddingSize: int
- model: VisionTransformer
+ Container<Embedding,MetaData>
+ search(embedding: Embedding, maxResults: int): List<MetaData>
+ clear(): void
+ insert(data: List<Embedding,MetaData>)
MetaData
+ id: imageID
+ getImage(): Image

Vector Database Details

1. **search:**
   - **Description:** This method is used to search the vector database. Given an embedding, it identifies the closest nine embeddings and returns the data associated with those embeddings.
   - **Return Type:** List<MetaData>
   - **Service:**
     - Before runtime, a predetermined searching algorithm is defined
     - Takes the given embedding and requested number of items to find

    - Uses the searching algorithm to identify up to X embeddings near the inputted embedding
    - Returns the metadata associated with the embeddings found

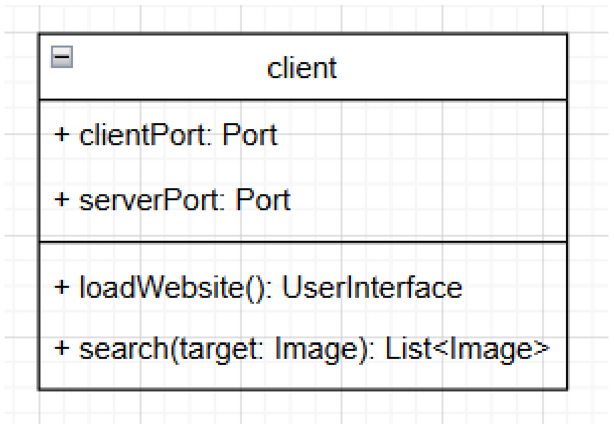
2. **clear:**
    - **Description:** This method is used to remove all data and reset all settings in the vector database
    - **Return Type:** void
    - **Service:**
        - Removes all items in the database
        - Deletes the database from memory
        - Marks the database as uninitialized
3. **insert:**
    - **Description:** This method is used to insert embeddings into the vector database, and each one with a metadata object.
    - **Return Type:** void
    - **Service:**
        - Takes a list of embeddings alongside associated metadata
        - Inserts the embeddings into the database to maximize efficiency in the search algorithm
        - Links each new embedding to the associated metadata

4.3 Client (Javascript)

Description

The client is the part of the system responsible for connecting the user interface to the server. The functionality of the client determines the ways that researchers can interact with the system. This includes the ability to load the website from a browser and search and retrieve similar images to the ones they input.

UML Diagram


client
+ clientPort: Port
+ serverPort: Port
+ loadWebsite(): UserInterface
+ search(target: Image): List<Image>

Client Details

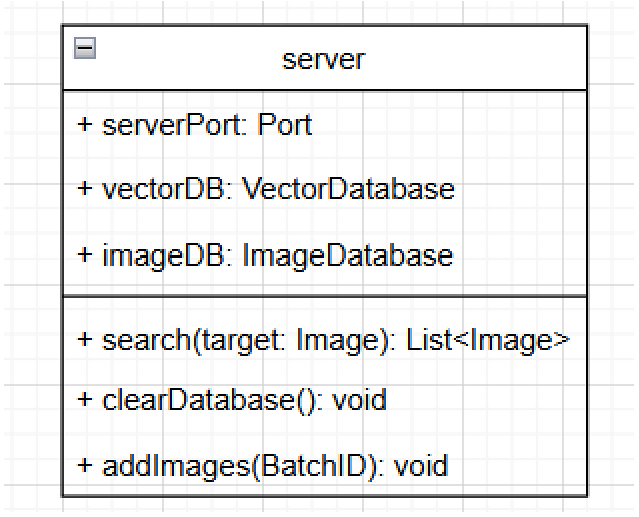
1. **loadWebsite:**
    - **Description:** This Method is used to initialize a user interface for researchers, which can access the system through the server.
    - **Return Type:** UserInterface
    - **Service:**
        - Called by users by typing the link into their browser
        - Initializes an application, allowing the user interface to interact with the server
        - Returns a HTML file containing the user interface
2. **search:**
    - **Description:** This Method is used to send a search request to the server
    - **Return Type:** List<Image>
    - **Service:**
        - Takes an image file
        - Converts the file into a string
        - Passes the string to the server
        - Waits for a list of image results from the server
        - Stores the image results into the user interface to be viewed visually

4.4 Server (javascript/python)

Description

The Sever has one of the most important jobs in this whole system. It is the network that connects all the Pieces. The front-end User Interface will interact with this to send data from the front to the back to receive similar images. This Server will connect the Image Database to the Vector database and allow for any data sent by the user to be processed and used in the server code to generate the images needed using a search engine written into the server. The server is where all the processing happens, and the image sent by the user is converted into a usable vector to compare with the database to receive the correct images and send them back to the front to be displayed to the user.

UML Diagram


server
+ serverPort: Port
+ vectorDB: VectorDatabase
+ imageDB: ImageDatabase
+ search(target: Image): List<Image>
+ clearDatabase(): void
+ addImages(BatchID): void

Server Details

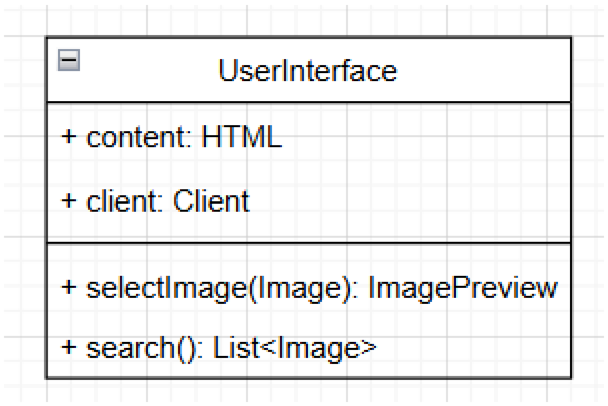
1. **Search:**
    - **Description:** This Method is used to search the vector database using the given image from the user that was converted into a vector that can use cosine similarity to find the X most similar images in the database and return them.
    - **Return Type:** List<Image>
    - **Service:**
        - Take given Vector
        - Compare to the Vector Database
        - Select the nine most similar images
        - Return the list of these nine images
2. **clearDatabase:**
    - **Description:** This Method is used to clear all the data from the vector database and make sure it is clean and empty to be repopulated with images.
    - **Return Type:** Void
    - **Service:**
        - Goes through the Vector Database and deletes all data
3. **addImages:**
    - **Description:** This Method is used to help populate the vector database with images from a selected folder/location. This method goes through all the images in this location, converts them to a vector, and sends them to the database to be stored.
    - **Return Type:** Void
    - **Service:**
        - Convert images from a given location to vectors
        - Send them one by one to the Vector database
        - Close the folder and finish populating the Vector database.

4.5 User Interface(html/css/javascript)

Description

The User Interface is the part of this system that the user interacts with and sees. This is built with HTML, CSS, and JavaScript. With this, we can see how the user interacts with the product through two main features the upload and the return of images. This is handled by many parts of the product but it is visually seen on the User Interface. This is the most important part as without this the user can not use the product.

UML Diagram



UserInterface Details

1. **selectImage:**
    - **Description:** This is a method that takes an image name/location and selects it to be shown in the website window. This is useful as this is how the product returns and shows similar images to the user.
    - **Return Type:** ImagePreview
    - **Service:**
        - Locates the image in the Image Database
        - Pulls image location to frontend/UserInterface
        - Displays Images on the website window for users to interact with
2. **search:**
    - **Description:** This is the entry point for the long chain of searching function calls that goes from the User Interface to Client to Server to Vector database and back up. This is the function that uses the image the user input to find the nine most similar images.
    - **Return Type:** List<Image>
    - **Service:**
        - Sends image to Client for passing and processing in lower levels
        - Waits for a response from the Client
        - When the response is received show a message of error or success
        - If successful, display images

5. Implementation Plan

5.0 Intro

For our implementation plan, we aim to begin with the overall structure we intend on following in regards to our timeline. Laying out each major chunk of development in the form of Phases. Each phase will provide a brief overview of what should be accomplished prior to the next phase.

5.1 Development Timeline

Phase 1: Semester 2 Setup & Prep (Weeks 1-2)

- Finalize system requirements and confirm architecture.
- Begin to understand codebase utilizing key technologies (TensorFlow & Milvus)

Phase 2: Core Functionality Implementation(Weeks 3-9)

- Implement image upload functionality and input validation.
- Interface with AWS bucket to retrieve needed information
- Set up Milvus database to store generated embeddings from AWS Bucket
- Integrate the front-end with the backend for seamless communication.

Phase 3: UI & User Experience Improvements (Weeks 10-12)

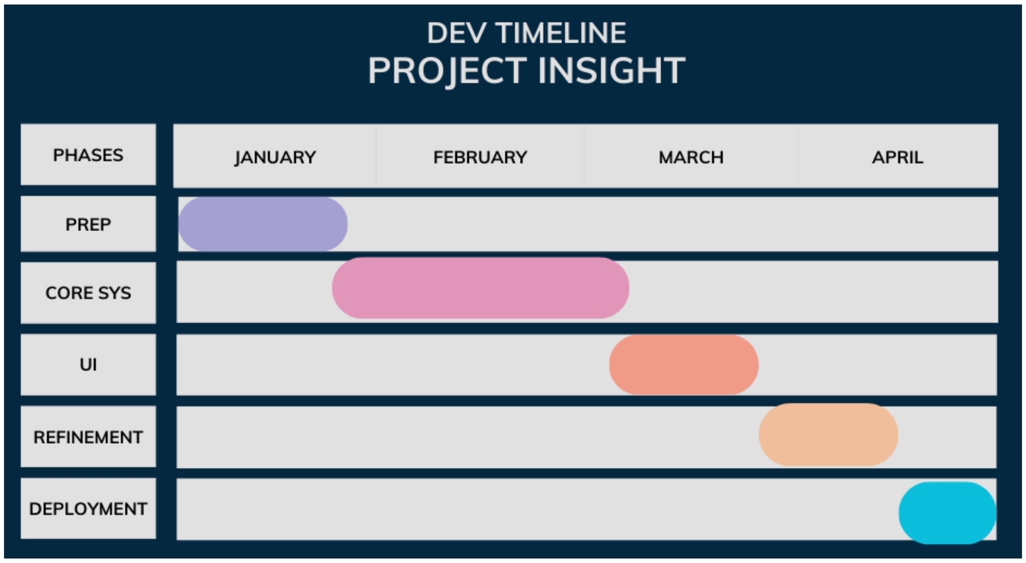
- Implement and further refine React-based UI for searching and displaying results.
- Improve error handling and user notifications.
- Conduct internal testing for usability and responsiveness.

Phase 4: Testing & Optimization (Weeks 13-14)

- Conduct unit testing for vectorization, search, and database interactions.
- Optimize search performance and refine cosine similarity calculations.
- Perform system-wide integration testing.

Phase 5: Deployment & Documentation (Weeks 15-16)

- Deploy the live system post client demo
- Write user and developer documentation.
- Conduct final presentation and submit project deliverables.



5.2 Milestones & Deliverables

1. **Setup Complete (Week 2)** – The development environment is understood and configured.
2. **API & Backend Ready (Week 9)** – Search functionality working with vectorized images.
3. **Usable UI Prototype (Week 12)** – Basic front-end facilitating image searches and results.
4. **Testing & Refinements (Week 14)** – Bug fixes, security checks, and performance optimizations.
5. **Final Deployment (Week 16)** – System live, documentation completed, and final review conducted.

6. Conclusion

With all this said, this project's main goal is to provide users with a sleek and well-rounded product that allows them to quickly and easily search for images in the Special Collections Archive using image-based searching. Utilizing many main tools of Javascript and Machine learning techniques like Milvus we can build a top-to-bottom application that allows for all this to be possible. We covered the modules that allow for all this to happen starting at the CONTENTdm database that stores all the images hosted by the SCA. The Milvus Vector database allows for cross-referencing the image given by the user with the contents of the SCA database. This is done with Milvus and Vector cosine similarity calculations. From here we have the Server and Client that build the actual web frame that hosts all this. The server handles the routing and calculations that allow all this to be possible. This is then handed to the Client, and the front-facing user interface takes the user's input and returns similar images.

By implementing all these things we can ensure that we will both follow the outline of our development plan and can deliver a fully working and complete project. With this as our goal, we are determined to make this project a reality and are excited to see it used by the students and staff of Northern Arizona University.