Final Report

Version: 1.0

Team: INSIGHT

Sponsor: Mike Taylor

Faculty Mentor: Scott Larocca

Team Members:
Joshua VanderMeer, Michael Vertin,
Aidan Hebert, Forrest Hartley

Date: 5/9/2025

Table of Contents

1. Introduction

Overview of the project's context and motivation:
The SCA has over 120k digitized images and ten million total images. The goal of this project is to provide the SCA with a tool that allows anyone to search through images.

Target users and problem domain:
Researchers can use this tool to find images of similar historical photos in a similar context. Historians can use this tool to identify images based on a certain era or the evolution of a location over a long period of time. Students/staff can use this tool to identify unfamiliar locations and learn background information about those locations. This can be used to identify how to cite an image that a researcher comes across.

Summary of client needs and the solution provided:
This system needs to allow users to have access to all the images in the SCA's collection. This was accomplished by using AI to help users target the type of image they want to find.

Summary of intended impact and vision of the product:

- Help people find sources related to a specific topic.
- Get information about an object or location.
- Find images that were unknown to the researcher.

2. Process Overview

To successfully deliver the Reverse Image Search tool for Northern Arizona University's Cline Library Special Collections and Archives (SCA), our team followed structured processes designed to enhance productivity, maintain clear communication, and meet project milestones effectively.

Development Methodology

Our team adopted the **SCRUM framework**, an agile project management methodology designed to facilitate flexibility, responsiveness, and continuous improvement. SCRUM allowed our team to break down the overall project goals into manageable increments, called Sprints, typically lasting two weeks each. This iterative approach helped us quickly address emerging challenges, gather timely feedback, and make adjustments based on evolving project requirements.

Each Sprint began with a planning session where team members identified key tasks, clarified objectives, and set clear expectations for the Sprint period. Regular stand-up meetings throughout each Sprint enabled team members to discuss progress, identify bottlenecks, and swiftly adjust workloads, ensuring consistent momentum and alignment with project deadlines.

Tools and Platforms Used

- **GitHub:** We utilized GitHub for version control and collaborative code management. This platform enabled seamless integration of code changes, effective handling of pull requests, and comprehensive tracking of development history. Additionally, GitHub facilitated transparent collaboration and simplified code reviews among team members.
- **Jira:** For project tracking and issue management, our team used Jira. It provided an organized framework for defining tasks, setting priorities, tracking progress, and managing Sprint backlogs. Jira's dashboard offered real-time visibility into ongoing tasks, team member responsibilities, and Sprint timelines.

Team Roles and Responsibilities

Our team structured roles clearly to optimize productivity and leverage individual strengths:

- **Joshua Vandermeer:** Team lead, Front-end developer, and API developer; coordinated team activities and was responsible for API integration and database management.
- **Michael Vertin:** Machine learning developer; handled integration with PyTorch and HuggingFace libraries and managed backend processes for image uploads.
- **Forrest Hartley:** Front-end developer and tester; ensured an intuitive user interface, seamless user experience, and rigorous frontend testing.
- **Aiden Herbert:** Project manager; oversaw SCRUM implementation, coordinated sprint planning sessions, and managed quality assurance activities.

This division of labor allowed each member to contribute effectively within their area of expertise, ensuring a cohesive and high-quality final product.

Work Organization and Communication Structure

Clear, consistent communication was fundamental to our project's success. Our team established routine communication protocols to maintain alignment, swiftly resolve issues, and support collaboration:

- **Weekly Sprint Reviews:** Comprehensive reviews conducted at the end of each Sprint, providing opportunities to demonstrate completed work, discuss challenges, gather stakeholder feedback, and make informed adjustments for subsequent Sprints.
- **Regular Communication Channels:** Discord was used for day-to-day communications, ensuring quick clarifications, efficient problem-solving, and ongoing collaboration.

By following this structured yet adaptable approach, our team maintained high productivity levels, effectively addressed unforeseen challenges, and delivered a robust tool aligned with the goals of the Special Collections and Archives

3. Requirements

3.1 Gathering Requirements

Before requirements were formally gathered, a basic design of the system was created to identify the necessary components of the system. The requirements that INSIGHT initially proposed consisted of the functionality of each component in the design. During discussions with the client, modifications were made until everyone approved of the requirements.

3.2 Functional Requirements

- **Embedding Generation**
    - A pretrained AI model will be used to generate embeddings for images. The AI model is located on HuggingFace, which can be used at any time.
    - The AI model uses calculations to generate an embedding for an image. To effectively compare embeddings, there should be a basic understanding of how the AI calculates embeddings. This information can be used to determine how to search for similar images, dissimilar images, or use multiple images to improve control over the search results.
    - The AI needs to support at least .jpg and .jp2 images. More image types should be supported as users may want to search using images of other types.
- **Storage**
    - Before the project is handed to the client, data for all images currently in the SCA should be saved in the vector database.
    - After INSIGHT passes control of the system, the SCA will need to keep the database up to date as images are added to the SCA's collections. Given files containing metadata in the format of the one provided at the beginning of the semester, a script on the EC2 instance must be able to perform the insertion process for the data in those files.
    - This process should be simple and easy to adjust, such that the SCA does not need a deep understanding of the system to expand it.
- **Search for Images**
    - The search process is the main functionality of the system that researchers interact with. Given the embedding of an input image, the vector database must support

    searching for embeddings similar to the input and return the metadata associated with those embeddings.

- **Metadata Content**
    - The vector database must support metadata for at least 500k images. To support interaction between the AI model and ContentDM, each of these items must contain an embedding for searching for similar items and a ContentDM ID that can be used to access a thumbnail image from ContentDM.
- **Website**
    - The website will be accessible to everyone through https://insight.library.nau.edu.
    - To perform a search, a user will select an image from their device and press search. This will send a search request to the EC2 instance with the image to search for. After the search is completed, a list of URLs will be returned, and the images in those URLs will be displayed to the user.
    - Images will be displayed in a grid format, with a quantity of results ranging from 3-15 images.
- **SCA Database**
    - To minimize memory usage in the system, the SCA database will be responsible for containing the images in the SCA's collections.
    - In order to generate embeddings and display search results, any image object must be attainable from its ContentDM ID. Using ContentDM's api, the object's ID can be used to generate an image link that can be sent to the user interface and rendered to the searcher.
    - Additionally, there should be a way to generate the URL for the ContentDM web page for any ID. For example, the webpage for item 15662 can be found at https://cdm16748.contentdm.oclc.org/digital/collection/cpa/id/15662. This web page can contain any metadata that the SCA wants researchers to know about the item.

3.3 Non-Functional Requirements

- Time requirements
    - The insertion process should be capable of inserting data for at least 1000 per hour.
    - Users will expect near immediate results when performing searches. For images under 10MB, the search should take no longer than 5 seconds.
- Database requirements
    - The vector database must be capable of storing at least 500k images. During the handoff, the database is expected to have the 120k images in the SCA's initial collection. As the system scales, the EC2 may require upgrades to support more data. The database should be able to accept this new data without modifications to its implementation.

- Maintainability
    - When issues arise, documentation will provide instructions to resolve the errors. If the process is hard to execute, a script will be created to make the process easier. Adding image references of a new format should require minimal code adjustments. If future metadata files are in a different format, adjusting to that format will require minimal adjustments to the code.
- Accessibility
    - The website should be active at least 99% of the time.
    - The website's user interface should be intuitive to use for people who are unfamiliar with the project's development or design.

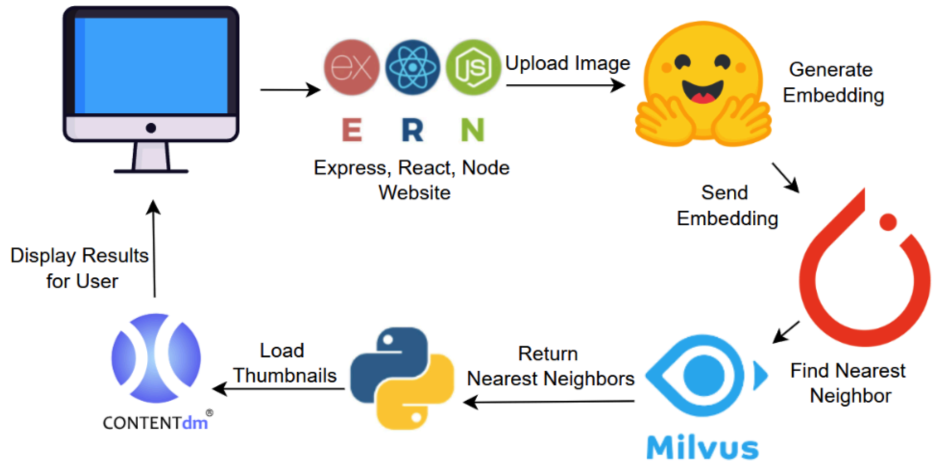
3.4 Meeting the Requirements

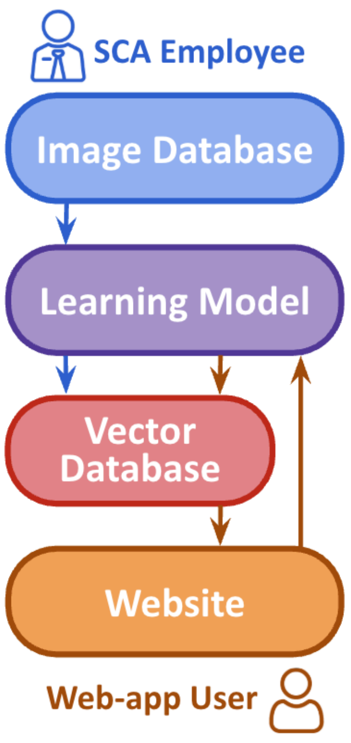
The majority of the requirements were a description of the functionality of the different components to make the entire system work. These requirements were met naturally by successfully implementing the insert and search functionality described in the requirements document. For optimization, several methods were tested to find which options worked best, and the design of the system was iteratively refined to support these new methods.

4. Architecture and Implementation

4.1 High-Level System Architecture

High-Level Flow of Data for System:





High-Level System Architecture:

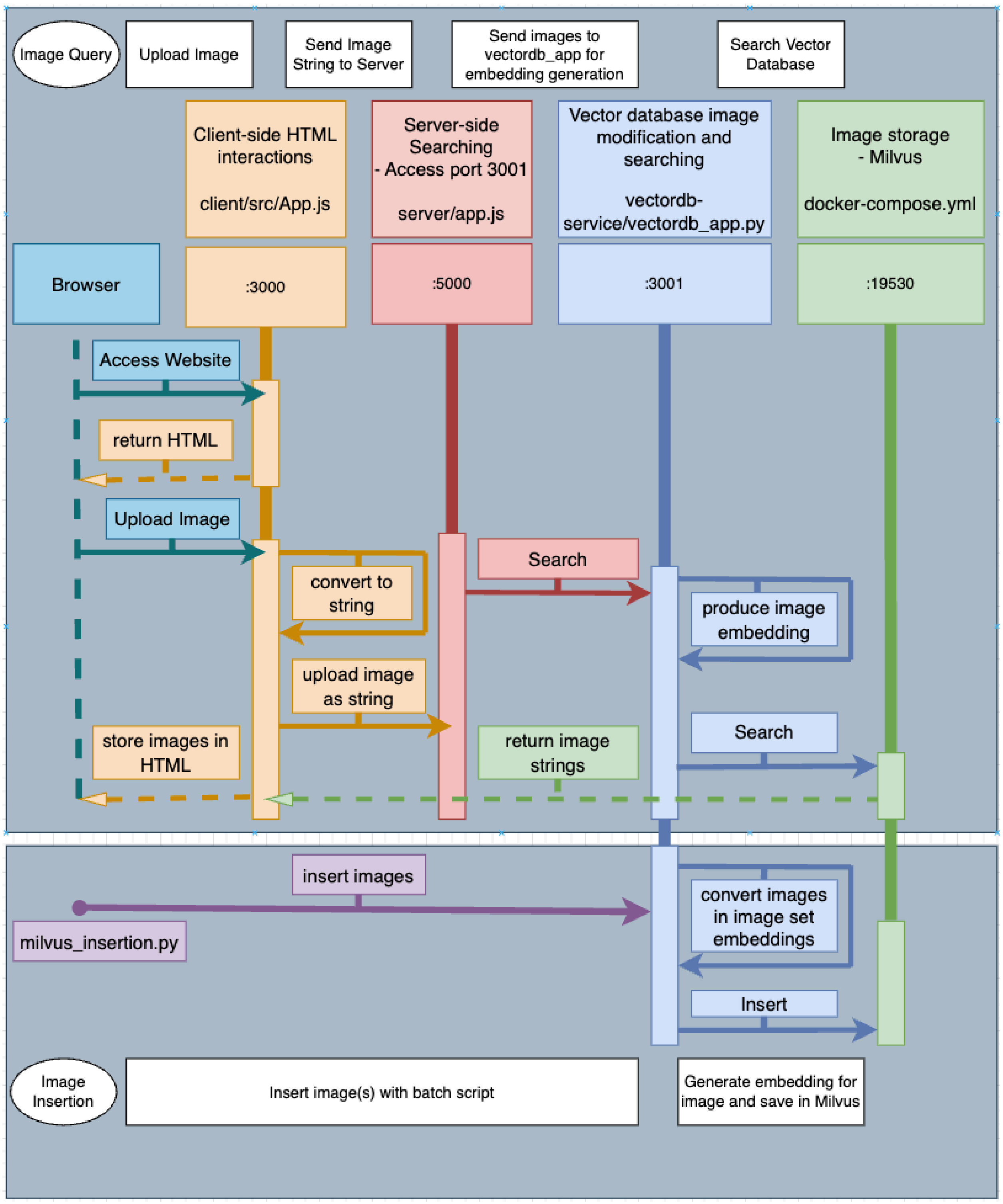


Image Query
Upload Image
Send Image String to Server
Send images to vectordb_app for embedding generation
Search Vector Database
Client-side HTML interactions client/src/App.js
Server-side Searching - Access port 3001 server/app.js
Vector database image modification and searching vectordb-service/vectordb_app.py
Image storage - Milvus docker-compose.yml
Browser
:3000
:5000
:3001
:19530
Access Website
return HTML
Upload Image
convert to string
upload image as string
store images in HTML
Search
produce image embedding
Search
return image strings
insert images
milvus_insertion.py
convert images in image set embeddings
Insert
Image Insertion
Insert image(s) with batch script
Generate embedding for image and save in Milvus

High‐Level View

Overview of System Components

1. **Browser (User Interface)**
   - Acts as the client-side interface.
   - Facilitates interaction with users to upload images and view search results.
2. **Front-End (React - client/src/App.js, Port 3000)**
   - Provides a user-friendly UI for image uploads.
   - Converts images to Base64 format for data transmission.
   - Displays search results received from the back-end.
3. **Back-End (Node.js - server/app.js, Port 5000)**
   - Manages communication between the front-end and vector database service.
   - Receives and forwards Base64 image strings from the front-end to the vector database service.
   - Relays search results back to the front-end.
4. **Vector Database Service (Python - vectordb_service/vectordb_app.py, Port 3001)**
   - Handles image embedding creation and database interactions.
   - Utilizes Hugging Face models to generate vector embeddings from images.
   - Performs similarity searches in the vector database (Milvus) using cosine similarity.
   - Supports batch insertion of new image sets through scripts like milvus_insertion.py.
5. **Milvus Database (Docker - docker-compose.yml, Port 19530)**
   - Stores embeddings and related metadata.
   - Efficiently performs similarity searches using specialized indexes.
6. **Batch Insertion Script (milvus_insertion.py)**
   - Enables bulk insertion of images by converting them to embeddings via the vector database service.
   - Inserts these embeddings into Milvus.

Key Responsibilities and Features

- **User Interface (Browser)**: Uploading images, displaying results.
- **Front-End**: User interaction, data formatting for transmission.
- **Back-End**: Data flow management, communication relay between front-end and vector database service.
- **Vector Database Service**: Image embedding generation, searching functionality, batch insertion.

- **Milvus**: Embedding storage, rapid similarity search functionality.
- **Batch Insertion Script**: Streamlined batch processing for large datasets.

Communication and Data/Control Flow

- **User Interaction Flow**:
    - Browser → Front-End (upload image).
    - Front-End → Back-End (Base64 image string).
    - Back-End → Vector Database Service (forward image).
    - Vector Database Service → Milvus (query embeddings).
    - Milvus → Vector Database Service (return search results).
    - Vector Database Service → Back-End (return image references).
    - Back-End → Front-End (relay results).
    - Front-End → Browser (display results).
- **Batch Insertion Flow**:
    - Batch Insertion Script (milvus_insertion.py) → Vector Database Service (bulk image set).
    - Vector Database Service → Milvus (store embeddings).

Example Use Cases

1. **Image Search**
    - A user uploads an image via the browser.
    - The front-end converts it into Base64 and sends it to the back-end.
    - The back-end forwards the request to the vector database service, which generates an embedding and searches Milvus for similar images.
    - The resulting similar image references are returned to the user.
2. **Batch Image Upload**
    - An administrator runs milvus_insertion.py to insert large batches of new images.
    - Images are processed through the vector database service to create embeddings and stored in Milvus.

4.2 Component-Level Design

4.2.1 Client (Frontend)

- **Purpose:** The main React component handles user interactions for uploading images, selecting search modes, setting result limits, viewing search results, and interacting with individual results. It orchestrates communication between the user interface and the backend search services.
- Key Classes/Files:
    - **App.js:** Main React app component, routes, and state management.

        - **index.js:** Entry point to the React app.
        - **logger.js:** Frontend logger that sends messages to the backend.
        - **App.css, index.css**: Styling.
- Important APIs:
    - /upload (POST)
        - Sends a base64-encoded image, the search mode (similar / non-similar), and the desired result limit to the backend for search.
    - Frontend Logging (via log_message)
        - Sends logging information about search actions and metadata quantities to the backend logging service.
- Interaction:

Aspect	Details
Image Upload	The user selects an image, which is read as Base64 and posted to /upload.
Search Mode Selection	Users can toggle between "similar" and "non-similar" searches.
Result Limiting	The user can drag a slider to control how many search results are retrieved.
Explore Mode	Clicking a result loads the clicked image as a new query (optional for users).
Result Display	Displays image search results, provides fallback images if loading fails.
Logging	Internal events (like a successful metadata set) are logged to the backend.

4.2.2 Server (Backend Express Server)

- Purpose: The Express backend server acts as a middleman between the React frontend (client/src) and the Python vector database service (vectordb-service). It handles image uploads, logs frontend events, and forwards requests to the Python service for searching.
- Key Classes/Files:
    - **app.js:** The core Node.js Express application for backend routing and API handling.
- Important APIs:
    - POST /upload

        - Receives a base64-encoded image and search parameters from the frontend, then forwards the search request to the Python service
    - POST /log_message
        - Receives logs from the frontend and records them via logger.js
- Interaction:

Aspect	Details
Image Upload Handling	Uses multer (in memory) to accept image uploads, focusing on base64 data sent as part of the request body.
Proxying Requests	Forwards image search requests to the Python Flask app at localhost:3001.
CORS Policy	Allows frontend React app (running on localhost:3000) to access backend services during development.
Logging Middleware	POST /log_message is used by the frontend to send structured logs.

4.2.3 Service Control Layer (Python Flask Mini-Service)

- Purpose: The Service Control Layer acts as a health monitor and manager for the system's major services (Milvus vector database, the Python vector database API server, and the Node.js/React client-server stack). It enables status checks, automatic resets if services go down, manual resets, and controlled "kills" and "unkills" of services, ensuring system stability without manual intervention.
- Key Classes/Files:
    - **service_control_app.py:** Flask app exposing API endpoints to control and monitor system services.
    - **service_controller.py**: Defines the logic for monitoring, killing, resetting, and reviving services via a thread-based background system.
- Important APIs:
    - POST /get-status:
        - Returns whether a service is alive or manually killed.
    - POST /service-reset:
        - Triggers a reset (restart) for a given service.
    - POST /service-kill:
        - "Kills" a service for a specified duration, preventing it from automatically restarting.
    - POST /service-unkill:
        - Manually re-enables resetting for a previously killed service.
- Interaction:

Aspect	Details
Monitoring Services	Services automatically check their health periodically and self-reset if needed.
Manual Control APIs	You can send manual reset, kill, or unkill commands through simple POST requests.
Service Definitions	Milvus_SC, VectorDB_SC, and ClientServer_SC handle specific services with customized reset logic.
Threaded Auto-Monitoring	Each service controller runs a background thread that monitors and repairs its service.

4.2.4 Vector Database Service (Python Flask App + Milvus Integration)

- Purpose: This is the core intelligence of the system: it takes user-uploaded images, computes their embeddings using a deep learning model, stores and manages these embeddings in a Milvus vector database, and performs similarity searches to retrieve relevant images based on the input. It also provides endpoints for managing the database (insert new images, clear the database, etc.).
- Key Classes/Files:
    - **Vectordb_app.py:** Flask app that defines and handles API routes like /search, /insert, /clear, and /log_message.
    - **Vector_database.py:** Interface layer to Milvus: handles insertion, search, and deletion of vector data in the database.
    - **image_database.py:** Utility to load images from a local folder for bulk insertion.
    - **embedding_model.py:** Loads the Hugging Face ViT (Vision Transformer) model and generates image embeddings.
    - **Image_reference.py:** Abstract and concrete classes for different image formats/sources (local file, base64 string, AWS S3, etc.).
    - **logger.py**: Handles structured logging inside the service.
- Important APIs:
    - GET /test
        - Returns success if the Flask server is alive.
    - POST /log_message ‘
        - Receives and records logs (e.g., insert success, search complete).
    - GET /insert
        - Reads all images from image-set/, generates embeddings, and stores them in Milvus.
    - POST /insert-images-by-metadata

        - Allows insertion of specific images described by their metadata.
    - POST /search
        - Receives a base64 image (or path), generates an embedding, and performs a similarity search in Milvus.
    - GET/POST /clear
        - Deletes all records from the vector database.
- Interaction:

Aspect	Detail
Image Input Handling	Supports images via file path, Base64 string, AWS S3, and ContentDM collections.
Embeddings Generation	Uses a pre-trained ViT model to convert images into 768-dimensional embeddings.
Database Operations	Insert, clear, search, and manage records through Milvus.
Logging and Monitoring	All major operations (insertion, search, errors) are logged via logger.py.

4.3 Design Rationale and Style

Architectural Styles

- **Client-Server Architecture:** Clearly separates client-side responsibilities (user interaction, data presentation) from server-side operations (data processing, business logic).
- **Microservices Architecture:** The system is structured into distinct, independently deployable services (front-end, back-end gateway, vector database service, and database), allowing focused, modular development and easier maintenance.

Rationale for Architectural and Tech Stack Decisions

- **Flexibility and Modularity:** Utilizing microservices enhances flexibility, allowing independent scaling and updating of components without impacting the overall system.
- **Scalability and Performance:** By employing Node.js and Python services optimized for their respective tasks, the system achieves improved performance and scalability.

- **Maintainability and Extensibility:** The clear delineation of responsibilities simplifies debugging, enhances maintainability, and allows straightforward integration of additional services or features.
- **Specialized Data Management:** Choosing Milvus as a dedicated vector database ensures optimized storage and rapid query responses for vector similarity searches, crucial for the application's core functionality.

Data Flow Management

- The structured flow ensures that user-uploaded images undergo systematic processing: encoding, embedding generation, similarity search, and results retrieval, maintaining consistency and reliability throughout the process.

4.4 As-Built vs. As-Planned

Our product was created to help users and researchers find images in the special collections and archives. To best figure out what we needed to build, we spent time with our client to determine what features were needed. You read about them in section 3 above, and here is what was built compared to the original plan. We built a web application that was to be hosted on an AWS EC2 instance that the SCA and our client owned. In this EC2 instance, we developed a small web application that allowed a user to upload an image and alter the search parameters to search for an image first. Initially, we planned to make it so the user could upload multiple images to help refine the search. Still, we ultimately decided for use on mobile and desktop, it would be more usable to have only one upload per search. On top of this, one upload, though we developed multiple features to help refine the search for the user. We implemented a Similar or Non Similar searching option that the user can toggle on every search. This allows the user to refine their search to images that are closely related or as far off as possible. Next, we developed an option for toggling the number of image results the user will receive, in case the default of 9 was too much or too little for their search. This gave the user the option of 3, 6, 9, 12, or 15 images as a result grid. Lastly, we added the most valuable feature, “explore mode.” This mode allows users to use an image result as the query for the next search to help them explore the collection and find the resources they need.

We learned some lessons during this development and the evolution of our applications' architecture. These lessons helped us make informed decisions and improve the application to help it be useful to as many users. We learned that it doesn’t need to be flashy or complex to have a very solid working application that will last the tests of time. This application turned out to be very sleek and usable and had some solid user testing reviews, which will be discussed in the next section.

5. Testing

User testing is a very important and valuable part of software development because it is the first time to test all the components together and then get some real-life user feedback. This is why we focused heavily on testing to ensure this application would be usable and useful for as many users as possible.

5.1 Unit testing

Unit testing is an important part of testing software because it tests the smallest and most delicate parts of the software. Unit testing also ensures that new features and old functionality still work well after changes are made. Using Pytest and Jest to test the functionality of different components of the software, we were able to ensure components worked as expected. Pytest was used for any Python-related components and used a class-based testing system. Jest is very similar but was used to test the frontend web application files written in JavaScript.

5.2 Integration testing

Integration testing is the next step after unit testing, as after unit testing is done, it is clear that the individual components work, but integration testing ensures they work together. Integration testing, if done correctly, can give the developer confidence that all the aspects of the software are working together and will not have any unexpected issues or difficulties. This is why we use Jest and Pytest to ensure that all the JavaScript and Python functionality work together, and the flow of data and retrieval from the backend to the frontend is working as expected. Some of the main things tested in integration testing were the ability for the uploaded user image to be sent to the backend and processed into an embedding. A separate test was run to test whether this embedding was able to be used to cross-reference the vector database. There was then a test to retrieve the data from that cross-reference and send it back to the back end. Finally, we have a test to display an array of images retrieved from the vector database on the front end. All of these tests were run with all the different search options to ensure the usability of all search features.

5.3 Usability testing

Now that the application was thoroughly tested on the software side of things, the next step was to get real user feedback. To do this, we asked peers to try it out and give any feedback. We also sent the working application to our sponsor and the SCA department to get their feedback, as they have a more technical view of these databases. We received feedback from the users to update the layout of the website to feel more modern and less cluttered. We also received feedback that the application was not user-friendly on mobile devices and had to spend some time to ensure it would work on every device. We also received positive feedback about our searching features and how the different options opened the door to a variety of searching options on this application.

6. Project Timeline

5.1 Development Timeline

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

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

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

- Implement image upload functionality and input validation.
- Interface with the provided metadata file to generate APIs for ContentDM.
- Set up the Milvus database to store generated embeddings from images in the collection.
- Integrate the front-end with the backend for seamless communication.

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

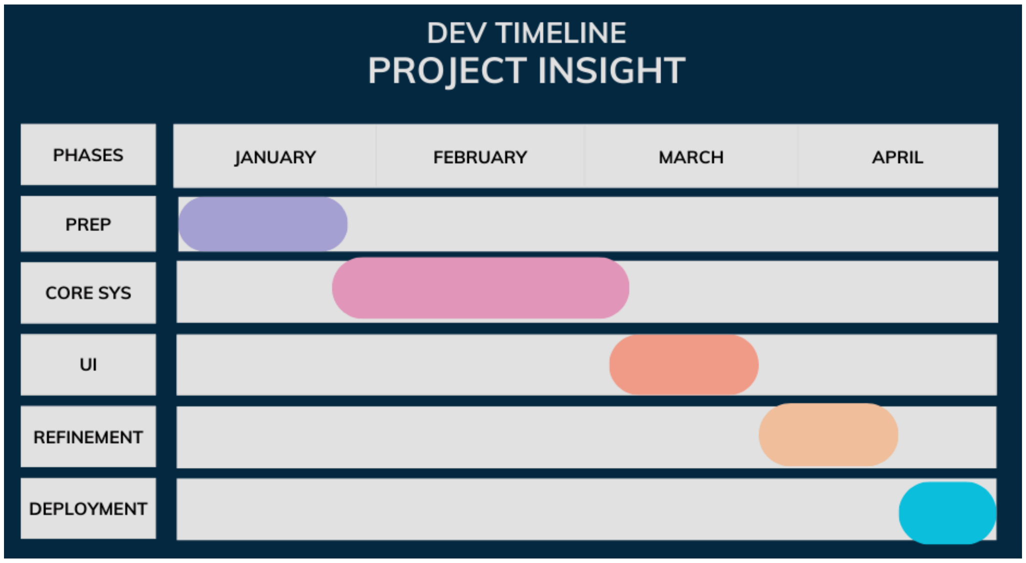
- Implement and further refine React-based UI for searching and displaying results.
- Improve error handling and user notifications.
- Conduct internal testing for usability and responsiveness.

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.
- Deliver the system to the client.



7. Future Work

The INSIGHT image similarity searching tool meets all the core requirements that our client desired. But there are several opportunities for enhancements and expansion in the future. These future improvements can further increase the system's utility, scalability, and ease of use.

- **Search Filtering by Collection or Metadata Fields**
  Currently, searches span all images in the database. Including filtering options such as limiting tsearches to specific collections, locations, styles, and subjects would provide more refined and targeted results for researchers.
- **Support for Face Detection and Recognition**
  Implementing face detection or recognition capabilities would allow users to identify individuals in historical photographs. This feature could be particularly useful for genealogical research or archival photo identification.
- **Improved Mobile Responsiveness**
  Although users can view this application on mobile devices, support can be improved, and continued refinement could better optimize the user experience across a wider range of mobile devices and browsers.
- **Image Upload History and Favorites**
  Allowing users to view their past uploaded images or save favorite results for later reference would enhance usability, especially for long-term research sessions.
- **Administrative Dashboard for the SCA**
  Developing an admin interface would empower SCA staff to monitor usage, upload new metadata, track insert progress, and trigger database maintenance tasks without needing to access the backend or EC2 instance directly.
- **Support for More Image Formats and Larger Files**

Expanding image format compatibility (e.g., .tiff, .png, .bmp) and improving handling of large image files would make the system more robust and accessible to a broader user base.

- **AI Model Fine-Tuning**
  While the current embedding model performs well, future iterations could explore fine-tuning the ViT model using SCA-specific imagery to improve relevance and accuracy in similarity results.
- **Accessibility Enhancements**
  Future updates could focus on improving accessibility compliance (e.g., WCAG standards), making the site easier to navigate for users with visual or motor impairments.

8. Conclusion

The INSIGHT Capstone project was accomplished with a clear and important issue at hand. Streamlining the process of identifying and locating similar images from the vast collection in possession of the SCA. The process previously in place was inefficient and inaccurate at times due to the nature of the manual tagging process. This process is what prompted our client, Mike Taylor, to allow us to implement a smarter, faster way to traverse through a photo collection of this scale, utilizing modern image processing and search technologies. To address this, our team designed and delivered a robust image similarity search tool, utilizing a scalable AWS EC2 environment and machine learning technologies from PyTorch and HuggingFace. These modern technologies enabled us to not only deliver a new streamlined process for searching with images, but also add additional features such as explore mode and nonsimilar searching. Further enhancing the way users can interact and explore the SCA collection. Reflecting on this experience, our team has gained invaluable skills in full-stack development, system deployment, and machine learning integration. Collaborating closely with a real-world client enhanced our communication and management abilities. Most importantly, we saw how applied software development can meaningfully solve actual issues. We are proud of the solution and system we delivered and are excited about its potential to grow beyond our work.

9. Glossary

Term	Definition
SCA (Special Collections and Archives)	A department within NAU's Cline Library that manages a large repository of digitized historical images and documents.
ContentDM	A digital collection management system used by libraries and archives to organize and present digital collections online.
Vector Database	A specialized database that stores and indexes high-dimensional vectors (embeddings) and enables fast similarity searches.
Milvus	An open-source vector database optimized for storing and querying large-scale vector data such as image embeddings.
ViT (Vision Transformer)	A machine learning model architecture designed for image analysis that uses self-attention mechanisms, provided by HuggingFace, in this project.
Embedding	A numeric representation of an image (or other data) used in machine learning to capture semantic meaning or visual features for comparison.
Base64	A binary-to-text encoding scheme used to represent image files as text strings, simplifying transmission over HTTP.
HuggingFace	A platform that hosts pre-trained AI/ML models, including vision and language models, widely used in modern ML applications.
EC2 (Elastic Compute Cloud)	Amazon Web Services' virtual server platform is used to deploy and host applications.
PyTorch	An open-source machine learning framework used to build and train deep learning models.
Pytest	A Python testing framework used to test backend functionality and services.

Jest	A JavaScript testing framework used for unit and integration testing in front-end applications.
React.js	A JavaScript library for building interactive user interfaces, used here for the search tool's front end.
Express.js	A Node.js web framework used for building backend APIs.
Explore Mode	A feature allowing users to select a result from a search and use it as a new query to further explore related images.
SCRUM	An Agile development framework involves iterative development cycles (Sprints), daily stand-ups, and continuous feedback.

10. Appendix

A.1 Hardware

This section outlines the hardware infrastructure utilized to host, develop, and deploy the reverse image search tool for the Special Collections and Archives (SCA).

- **AWS EC2 Instance**
    - Type: t2.large
    - Operating System: Ubuntu Server 22.04 LTS
    - CPU: 2 vCPUs
    - RAM: 8 GB
    - Storage: 30 GB EBS storage

A.2 Tools

This section documents the software tools, platforms, and technologies that enabled the project's successful implementation.

Development Tools & Environments

- **Visual Studio Code**: Primary IDE used for frontend (JavaScript/React) and backend (Node.js, Python Flask) development.
- **Docker**: Containerization tool utilized for deploying the Milvus vector database and maintaining environment consistency across development and deployment.

Project Management & Collaboration

- **GitHub**: Version control and code management platform for maintaining the repository, managing branches, code review, and continuous integration.
- **Jira**: Project management software utilized for Sprint planning, backlog management, task assignment, and progress tracking.
- **Discord**: Primary communication tool for daily interactions, quick feedback, and informal meetings.

Machine Learning & AI Frameworks

- **PyTorch**: Open-source machine learning framework used to search embeddings within Milvus.
- **HuggingFace Transformers**: Provided pre-trained models, specifically the ViT model utilized for image embedding generation.

Web Technologies

- **React.js**: Front-end JavaScript library used to build a responsive, intuitive, and user-friendly web interface.
- **Node.js & Express**: Backend runtime and web framework facilitating communication between the front-end and vector database service.

Database and Storage

- **Milvus Vector Database**: Open-source vector database optimized for managing high-dimensional embedding data and rapid similarity searching.
- **ContentDM**: External content management platform integrated via API to retrieve image metadata and thumbnails.

Testing Tools

- **Jest**: JavaScript testing framework used for unit and integration tests on frontend components.
- **Pytest**: Python testing framework employed for validating backend and vector database service functionalities.