Fuel BEACON

Final Report

Pacific Air Forces

Mentor:

Professor Parteek Kumar

Washington State University

BEACON SOLUTIONS

Aidan Johnson

Andrew McGann

Caitlin Graves

Roy Zabetski

CptS 423 Software Design Project II

Spring 2025

TABLE OF CONTENTS

I. Introduction

Machine learning and natural language processing have opened up exciting possibilities for improving various processes, particularly in handling unstructured data like handwritten notes. As organizations seek more efficient ways to digitize and analyze such information, the integration of advanced AI-driven technologies has become increasingly essential. Our project focuses on creating a Proof-of-Concept application, utilizing visual language models (VLMs) for handwriting recognition and large language models (LLMs) for data processing. By leveraging these powerful tools, we aim to develop a system that can effectively read, process, and generate reports from handwritten fuel logs, transforming a traditionally time-consuming and error-prone task into a seamless, automated process.

The Fuel BEACON project is designed to enhance the United States Air Force's fuel accounting system, which currently relies on manual data entry. In its current state, personnel must painstakingly transcribe fuel transaction records by hand, increasing the likelihood of human errors and inefficiencies. By automating the extraction and processing of fuel transaction information, we hope to save time, improve data accuracy, and alleviate the burden on service members responsible for fuel management. Our user-friendly application runs locally on government-issued laptops, ensuring compliance with strict security protocols while providing personnel with an intuitive and efficient interface. Additionally, our approach prioritizes adaptability and scalability, meaning that this technology could potentially be expanded for use in other military and government applications.

Through this work, we aim to demonstrate how combining these advanced technologies can streamline operations and improve efficiency in military logistics. By showcasing the potential of AI-driven solutions in structured data processing, Fuel BEACON represents an important step toward modernizing mission-critical systems, ensuring that the Air Force can maintain operational readiness with greater accuracy and reliability.

Our primary client is the Pacific Air Forces 673rd Logistics Readiness Squadron, with Senior Airman Aaron Keys (aaron.keys@us.af.mil) acting as our primary contact. Senior Airman Keys is a Fuels Service Center accountant dedicated to integrating modern technology with the Air Force to improve fuel accounting processes. Our application is used as a proof of concept product by members of the Logistics Readiness Squadron to demonstrate the feasibility of extracting, processing, and verification of data from fuel reports.

Our mentor at Washington State University is Professor Parteek Kumar (parteek.kumar@wsu.edu), a distinguished faculty member at Washington State University, specializing in AI, and utilizing Machine Learning for social good. He provided invaluable academic guidance on the application of Large Language Models in this project.

II. Team Members & Bios

Caitlin Graves is a computer science major at Washington State University, set to graduate in May 2025. Her experience includes two internships in business analytics, as well as active involvement on campus as a teaching assistant, a cabinet member for Girls Who Code, and a mentor in the Team Mentoring Program. Caitlin's technical skills focus on databases and software design with experience in C/C++, C#, Python, and Web Development. For this project, her responsibilities as the team leader include managing database architecture, contributing to software development, and ensuring design alignment with project goals.

Andrew McGann is a computer science major/mathematics minor at Washington State University, set to graduate in May 2025. His experience is strongest in the category of web development with a strong foundation in algorithms and data manipulation. Andrew's skills include C/C++, C#, Java, Javascript, Python, utilizing tools such as React, Express, and .NET. For Beacon Solutions, his responsibilities include developing the report and query functionalities, creating the Sprint videos, and system testing.

Aidan Johnson is a computer science major at Washington State University, set to graduate in May 2025. His technical background is primarily in data mining and processing with Python. Aidan's skills include Python, Java, and C#, as well as Web Development with JavaScript and Flask. For this prototype, his responsibilities include developing the subsystem for processing scanned documents and extracting the handwritten text from the processed documents.

Roy is a computer science major at Washington State University, and his expected graduation date is May 2025. For this prototype, his responsibilities include designing and developing most of the front-end UI components. He enjoys working on full-stack applications and has a background in Python and various object-oriented programming languages. He has also worked with many popular tools and platforms on an enterprise level.

III. Project Requirements Specification

This section defines the project requirements of Fuel BEACON. This includes identifying relevant stakeholders in the project, expected use cases of the application, and the Functional and Non-Functional Requirements.

III.1 Project Requirements Specification

Our primary client is the Pacific Air Forces 673rd Logistics Readiness Squadron, with Senior Airman Aaron Keys acting as our primary contact. Our application is used by members of the Logistics Readiness Squadron as a proof of concept application to automate the extracting, processing, and querying of data from fuel delivery forms.

Resources available to the Logistics Readiness Squadron require that our application, including any utilized LLMs/VLMs, is capable of running on government issued laptops (16 gb RAM, no dedicated GPU).

In addition, the Pacific Air Forces heavily prefers the application to run locally, to avoid any potentially secure data from being stored on cloud networks.

The expected users of our application are logistics personnel in the Logistics Readiness Squadron, with no particular technical background. As such, our application is required to be simple and intuitive to use, with either a simple GUI wrapper or web application base.

The Fuel BEACON project is intended as a proof-of-concept application to show the Pacific Air Forces the potential in LLM/VLM automation of existing tasks, which means our code base must be clean and easy to interpret. If the application is successful, it is likely to be expanded or modified for different tasks in the future by other teams, and those teams must be able to understand our work in the future.

III.2 Use Cases

The use cases illustrate typical scenarios of user interactions with the system, demonstrating how different functional requirements are implemented in specific contexts. The proposed use cases are depicted in the use case diagram presented in Figure 1. Following the diagram each use case is listed out with their corresponding software flow.

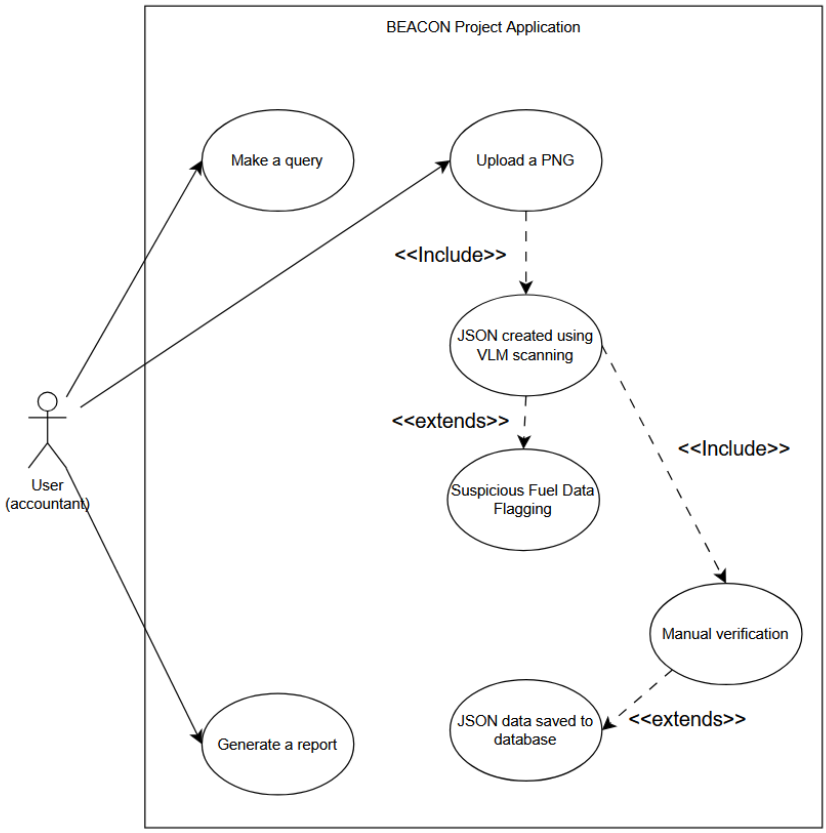


Figure 1: Use Case Diagram

III.2.1 **Document Upload** UC01

Actors	User, System
Pre-condition	The user has connected to the system and has a csv file scanned and in pdf format on their system.
Post-condition	The user has successfully uploaded their document into the local system for further processing.
Basic path	1. The user navigates to the Upload Document page. 2. The user clicks “Upload” and is prompted to select which document to upload to the local system. 3. The user selects the csv scanned file to upload.

	4. The document is uploaded to the local system. 5. The system prompts the user if they would like to upload another document, or scan the document they just uploaded. 6. If they choose to upload another document, they are taken back to step (ii) in the Main Flow. If they choose to scan the document, they are taken to the Scan Document page (UC02 Document Scan and Flagging).
Alternative path	● If something goes wrong with the file upload (incorrect file format, local memory low, etc.) then an error message is displayed with the proper error code and description. ● The user is prompted to either choose another file to upload, or re-try the same file upload.

III.2.2 Document Scanning and Flagging UC02

Actors	User, System
Pre-condition	The user has already successfully uploaded a document into the local system
Post-condition	The user has successfully scanned and flagged their document for suspicious information, and the data is ready to be verified.
Basic path	1. The user navigates to the Scan Document page. 2. The user clicks “Scan” and is prompted to select which uploaded document to scan into the local system’s database. 3. The user selects one of the previously uploaded documents. 4. The document is automatically scanned using the VLM Document Reader. Read values are stored in a structured JSON format. 5. Running concurrently with the VLM Document Reader, the Suspicious Data Flagger “flags” data that is deemed incorrect or inconsistent, or that falls below a certain confidence value. Again, this is done automatically. 6. Both the Reader and the Flagger finish at the same time.

	7. The user is brought to the Manual Data Verification page (UC03 Fuel Data Verification) to verify the data that was just scanned.
Alternative path	● If the user is brought from UC01 Document Upload, then skip to step (iv). ● If something goes wrong with the document scan and flagging, then an error message is displayed with the proper error code and description.

III.2.3 **Fuel Data Verification** UC03

Actors	User, System, Database
Pre-condition	The user has already successfully uploaded a document into the local system
Post-condition	The user has successfully queried for specific fuel data/reports based on the selected criteria.
Basic path	1. The user is brought from the Scan Document page (UC02 Document Scanning and Flagging) to the Manual Data Verification page. 2. The user is shown the uploaded document in png form on the left side of the screen, with the typed/handwritten original (actual) values displayed. On the right side of the screen are the system-read values that were just scanned from the document by the VLM Document Reader. 3. Those values on the right that were flagged during the scanning process are highlighted for extra attention with a small message indicating the reason (incorrect, inconsistent, not confident, etc.). 4. The user verifies all highlighted values by cross-referencing them with the actual values displayed on the left. Values can be edited as needed. Once all highlighted values are verified, the “All values are verified” button becomes available at the bottom of the interface. 5. The user can quickly verify all other non-highlighted values if they choose to do so. 6. The user clicks “All values are verified”. 7. The now verified values are stored in the local system’s database in JSON format.
Alternative path	● If no values were flagged for review, then the user skips steps (iii) and (iv).

	● If something goes wrong storing values in the database (low memory), then an error message is displayed with the reason, but re-verification is not necessary.

III.2.4 Fuel Data Query UC04

Actors	User, System, Database
Pre-condition	The user has already uploaded verified fuel data into the local system.
Post-condition	The user has successfully verified the scanned data with the actual data, and the verified fuel data has been stored in the local database in JSON format.
Basic path	1. The user navigates to the “Fuel Data Query” section/page. 2. The user selects a filter (month(s), data type/category, etc.) and/or enters a search term. 3. The system retrieves and displays the relevant fuel information based on the selected filters and/or search term. 4. The user scrolls through the results (if necessary). 5. If one of the results is a report, the user can click on the result and view the entire report.
Alternative path	● If no results are found for the selected filter and/or search term, the system displays a “No results found” message.

III.2.5 Report Generation UC05

Actors	User, System, Database
Pre-condition	The user has already uploaded verified fuel data into the local system.
Post-condition	The user has successfully generated a pdf report based on the inputted details
Basic path	1. The user navigates to the “Generate Report” page. A list of all previously generated reports is given, with timestamps. 2. The user clicks “Generate New Report”. 3. The user is prompted with entering the details for the report (type, time length for report, etc.). 4. Once the details are entered, the relevant data is compiled from the system’s database. 5. The system formats this data into the given report format.

	6. Once completed, the report is displayed on the screen with the option to download the file as a pdf, or print from a printer. 7. The user may choose to go back to the main report page with the "Generate New Report" button and the list of previous reports.
Alternative path	● If something goes wrong with the report generation (invalid details, invalid data, etc.) then an error message is displayed with the reason.

III.3 Functional Requirements

This section outlines the key functional requirements. The functional requirements include the ability to upload PNG images, process and extract relevant data using an open-source LLM, convert the extracted information into JSON format, and provide a user-friendly interface for interaction. Each functional requirement is listed below with a detailed description, source, and priority level.

III.3.1 User Interface

Document Uploader:

Description	An interface allows the user to upload documents to our application processes. This process verifies that a document with the correct format has been uploaded and that all the criteria match.
Source	Requirement originated from client
Priority	Priority Level 0: Essential and required functionality

Fuel Data Query Interface:

Description	A simple, intuitive interface allows users to query any available fuel data. We program this functionality to cater to what the user needs to query frequently. Different filters and third-party searching algorithms may be implemented in this functionality.
Source	Requirement originated from the client.
Priority	Priority Level 0: Essential and required functionality

III.3.2 VLM Document Reader and Review

Vision Language Model Document Reader:

Description	A vision language model (VLM) reads the user inputted documents, interprets their formatted structure and handwritten values, and stores them in a structured JSON format. We program this functionality to ensure the VLM correctly interprets each field with its corresponding value, reliably translates handwritten data, and stores the data in JSON format so that other functionalities can be used effectively.
Source	Requirement originated from client
Priority	Priority Level 0: Essential and required functionality

Fuel Data Suspicious/Missing Data Flagging:

Description	Our application reviews and flags data whenever it detects that it may be incorrect or missing. Since the input data is handwritten, we implement measures to flag values that may have been misread. We either do this by measuring VLM confidence on each value read or using multiple models to compare the results. The application parses the data for missing values and flags those as well.
Source	Requirement originated from Beacon Solutions team members with the goal of elevating the application's reliability.
Priority	Priority Level 1: Desirable functionality

III.3.3 Report Generation

PDF Report Generator:

Description	Our application generates a monthly PDF report summarizing all relevant fuel data. This process is crucial, as it will automate an 8-hour process and save a lot of time for our clients. We reference pre-existing reports to replicate them with our automated report.
Source	Requirement originated from the client.
Priority	Priority Level 0: Essential and required functionality

III.4 Non-Functional Requirements

The non-functional requirements describe the operational characteristics of the system, including performance, scalability, and security, ensuring it adheres to quality standards beyond its fundamental functionality. Further details on these non-functional requirements are provided below.

Non-Functional Requirement		Description
Locally Hosted		The application shall be locally hosted, with all data processing, including running the VLM/LLMs, being completed on the single machine. The application must be able to run on laptops with 16gb of RAM and no dedicated GPU.
Intuitive for Non-Technical Users		The application shall be designed in such a way that future users need minimal training to be able to process fuel reports and query data from previous reports. Users should not need to write SQL queries.
Accurate		The image recognition by the VLM shall be accurate enough that a human does not need to double check its work, or in the event that is not possible, potentially incorrect readings by the VLM must be flagged for human verification.
Fast		The runtime of the application shall be considerably faster than the time it would take for a human worker to achieve the same results.
Uses Domestic Technology		The VLM/LLMs used by the application should not have significant ties to foreign nations, particularly nations the U.S. are not friendly with (China, Russia, etc.). Due to the current state of Generative AI, this is a soft requirement, and can be ignored if it is not feasible.

IV. Software Design

This section explains the design and technical implementation of Fuel BEACON, focusing on how the system's architecture and subsystems address the project requirements. It includes an overview of the modular design, subsystems such as document upload, VLM-based data extraction, data flagging, querying, and report generation, as well as the technologies and methodologies used.

IV.1 Architecture Design

The system overview contains a general description of the functionality and design of the project. The overview will only briefly describe the overall design considerations and the comprehensive

explanations will be done in the sections to follow. The overview should serve as an introduction to these sections.

Fuel BEACON is separated into two major sections: Data Extraction and a Fuel Report database.

The Data Extraction section takes in handwritten documents, filled out by fuel truck drivers after delivery, and CSV files submitted by the dispatch centers. These documents are compared against each other, and the data within is analyzed for anomalies (i.e. a 6,000 gallon fuel truck delivering 8,000 gallons of fuel). This portion is managed by a VLM (QWEN-2), with human intervention required to handle any low confidence analysis by the VLM. [9]

Once the data has been analyzed and any discrepancies have been corrected, the completed Fuel Reports, along with submitted documents, are stored in a database for future use. The database is also queryable, with specific data from the submitted fuel reports able to be accessed as needed.

IV.1.1 Overview

The Fuel Beacon project has chosen to use the Model-View-Controller (MVC) architecture for its software design. This architectural pattern aligns well with the application's requirements, as it allows for a clear separation of concerns between the user interface, business logic, and data management. Given the need for a responsive user experience in a Python application developed with PyQt6, the MVC pattern is the most effective choice to facilitate independent development and maintenance of each component. Below is a component diagram that illustrates the overarching architecture of the Fuel Beacon application. This diagram serves a dual purpose: it acts as a guide for developers during implementation and provides a visual representation of the component architecture, which will be further detailed in the following sections. [7]

The components of the architecture include the View, which is responsible for presenting the user interface using PyQt6 and handling user interactions. The View communicates directly with the Controller, which manages the application flow and serves as the intermediary between the View and the Model. The Model comprises various subsystems, including the Fuel Transaction Query, which processes user queries to fetch relevant fuel transaction data, and the Handwritten Form Conversion, which handles the transformation of scanned documents into structured JSON data using a VLM. Additionally, the Document Upload subsystem allows users to upload files from their local storage for processing, while the Data Flagging subsystem reviews the extracted data to identify and highlight any discrepancies or missing information. Outside the model data from the Handwritten Form Conversion and Data Flagging subsystems are stored within the MongoDB database, while the Fuel Transaction Query subsystem retrieves information from this database as needed. Furthermore, both the Document Upload and Report Generation subsystems facilitate the saving of documents in designated locations within the system, ensuring that all relevant data is accessible and properly organized.

The structured approach provided by MVC ensures that changes to the user interface or data handling can be made without affecting the entire system, promoting scalability and maintainability. In-depth descriptions of each component and its responsibilities will be provided in the following sections.

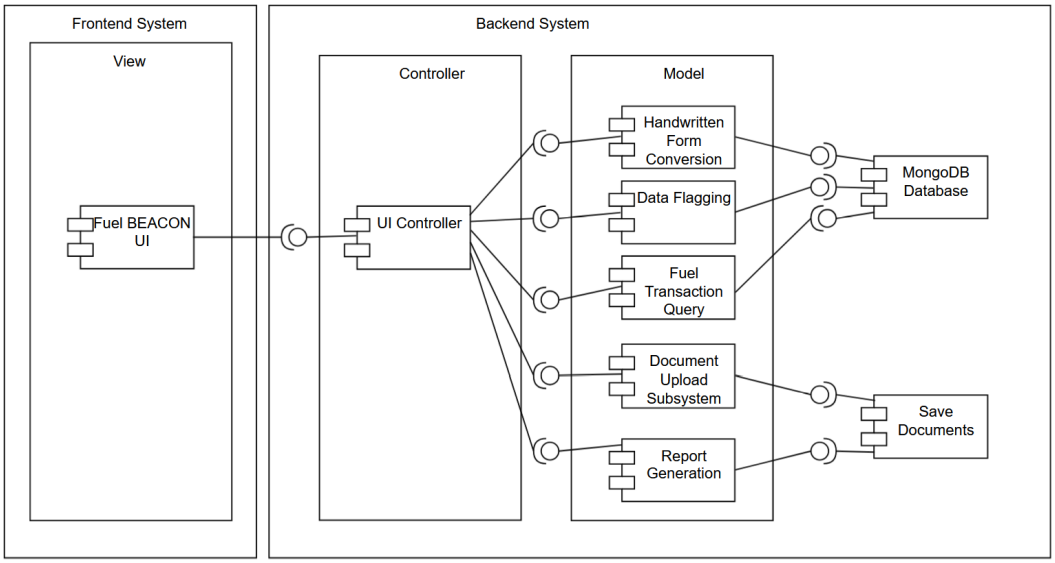


Figure 2: Architecture Design Diagram

IV.1.2 Subsystem Decomposition

The subsystem decomposition breaks down the application into smaller, functional components, each responsible for handling specific tasks within the system. Below are the key subsystems that drive the core functionality of the application, along with their descriptions, algorithms, and services provided and required.

IV.1.2.1 Fuel Transaction Query

- **Description:** The Fuel Transaction Query subsystem handles all query-related operations, including transaction look up via a search term and/or filters, viewing uploaded transaction reports, and data compilation over a set timeframe.
- **Concepts and Algorithms Generated**
    - Search Term Algorithm: Fetches all relevant data in accordance with a given search term.
    - Filter Application Algorithm: Fetches all relevant data in accordance with a given filter(s).
- **Interface Description**

Services Provided:

Service Name	Service Provided To	Description
Search	UI Controller, MongoDB Database	Accepts a search term and returns the relevant fuel data in a digestible manner
Filter	MongoDB Database	Accepts a given filter(s) and returns the relevant fuel data in a digestible manner

View Report	UI Controller, MongoDB Database	When a transaction report is returned, it can be viewed in full
View Timeframe	MongoDB Database	Returns a mini-report of the compiled data from a given timeframe

Services Required:

Service Name	Service Provided From
Document Upload	Document Upload subsystem
Data Storage	MongoDB Database

IV.1.2.2 Report Generation

- **Description:** The Report Generation subsystem compiles and provides reports from the integrated data of different transaction reports for Air Force fuel records.
- **Concepts and Algorithms Generated:**
    - Report Generation Algorithm: Aggregates data based on a selected timeframe and/or format, and generates a customized report.
- **Interface Description**

Services Provided:

Service Name	Service Provided To	Description
Generate Monthly Fuel Report	UI Controller, Save Documents	Creates reports based on predefined criteria.
Generate Customer Report	UI Controller, Save Documents	Creates reports based on a custom timeframe and/or format.

Services Required:

Service Name	Service Provided From
View Timeframe	Fuel Transaction Query Subsystem
Data Retrieval	MongoDB Database

IV.1.2.3 Handwritten Form Conversion

- **Description:** The Handwritten Form Conversion subsystem takes in PDFs of handwritten forms turned in by truck drivers after a delivery, and uses a VLM to convert those PDF forms into JSON files for analysis.
- **Concepts and Algorithms Generated**
    - The VLM Fuel BEACON is utilizing (QWEN-2) has a limited output of 128 - 256 tokens before querying time and accuracy become unsustainable. [9] To workaround this, each PDF is first converted into JPG files, which are then cropped to segments small enough for their output to be within the acceptable range.
    - The current method of cropping requires pixel coordinates for each region on the form to be cropped, which means each unique form template must be manually established in the cropping script before the subsystem can be utilized. [6]
    - Some cropped regions require information located in other regions of the PDF, for instance a table requires the column headers. This is problematic, as if the bottom rows of a table are being analyzed, and the headers need to be included, every row in between will also be included. This makes it difficult to avoid surpassing the output limit.
    - To avoid this, certain sections for the forms take in multiple images, for example the header row of the table and the selected row of data. This allows for each row to be processed with the headers, but leads to slower processing.
- **Interface Description**

Services Provided:

Service Name	Service Provided To	Description
JSON Data Extraction	UI Controller, MongoDB Database	Retrieves data using a VLM and saves within the database.

Services Required:

Service Name	Service Provided From
Data Cleaning and Flagging	Data Flagging Subsystem

IV.1.2.4 Document Upload Subsystem

- **Description**: The document uploader subsystem allows users to select documents from their machine's local filesystem and upload them to the application, where the application stores the files locally.
- **Concepts and Algorithms Generated:** The document uploading process likely does not

involve the use of algorithms, however concepts of good user interface and local file storage are implemented.

- **Interface Description**

Services Provided:

Service Name	Service Provided To	Description
Upload files	Save Documents	Uploads files of multiple types that can later be processed by our document scan subsystem.

Services Required:

Service Name	Service Provided From
Save Document to File System	Save Documents

IV.1.2.5 Data Flagging

- **Description**: This subsystem reviews scanned data and identifies potential entries that are either faulty or missing. These entries are then presented to users, who can either edit the data as needed or opt to ignore the flag.
- **Concepts and Algorithms Generated:**
    - For identifying missing data, as the system will expect an array of particular fields to be processed, and if one of these fields are missing or read with an empty value, the system will notify the user.
    - Identifying data that is potentially faulty involves the use of confidence level generated by the language model that processes the data. If the confidence level is below a certain threshold, the data will be flagged.
- **Interface Description**

Services Provided:

Service Name	Service Provided To	Description
Check for faulty data	MongoDB Database	Checks data to ensure there are no null values.
Generate Customer Report	UI Controller, Save Documents	Creates reports based on a custom timeframe and/or format.

Services Required:

Service Name	Service Provided From
Data Extraction from Form	Handwritten Form Conversion
Data Saving	MongoDB Database

IV.2 Data design

For this application, there are two primary data structures of concern, both of which interact with external subsystems. The first is the FuelRecord structure, which stores fuels accounting data extracted from PNG images and interacts with the language model for data extraction. The second structure is the FuelReport, which is generated based on multiple FuelRecord entries and used for creating detailed reports. Both data structures integrate with the MongoDB database for persistent storage and JSON generation, enabling seamless data manipulation. For more details on the internal subsystem interactions, refer to the System Architecture diagram and description above. [5]

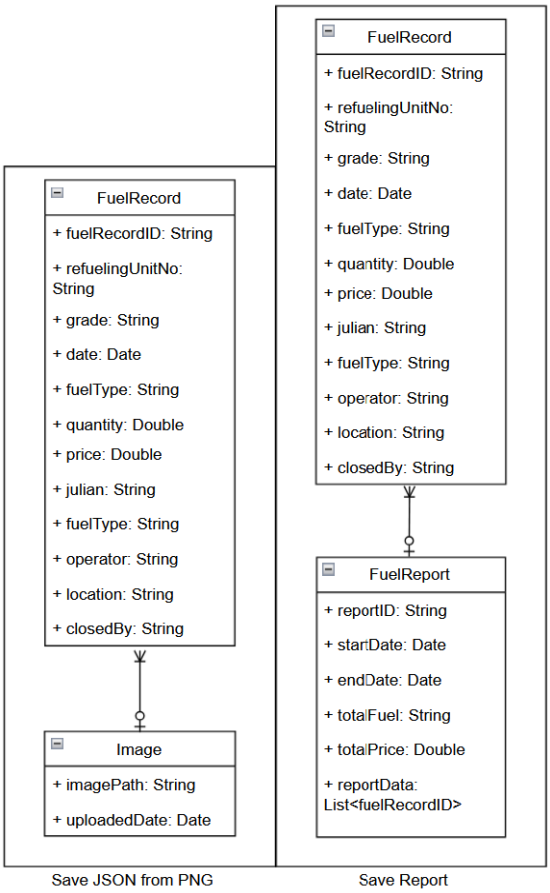


Figure 3: Data Design Diagram

Figure 3 details more of the relationships between the Image, FuelRecord, and FuelReport structures, illustrating how data flows through the system. More information on the diagram and each data structure is provided below.

For the “Save JSON from PNG” data structure, the main relationship involves two entities: Image and FuelRecord. In the Image-to-FuelRecord (“Save JSON from PNG”) relationship, an Image serves as an input source for fuel data extraction. Each Image object represents a PNG file that is processed to generate one or more FuelRecord entries. The Image contains details such as imagePath and uploadDate, which are necessary for processing. After processing, the fuel data extracted from an Image results in multiple FuelRecord objects, each of which encapsulates data like the fuelType, quantity, price, and the date of the transaction. This one-to-many relationship signifies that a single Image can lead to multiple FuelRecords, depending on the amount of fuel information extracted.

On the other hand, the FuelRecord-to-FuelReport relationship forms a many-to-one association. Multiple FuelRecord objects are compiled into a single FuelReport. Each FuelReport consolidates FuelRecord objects for a specified period or set of criteria, storing information like the reportId, startDate, endDate, totalFuel, and totalPrice for that report. This structure allows a comprehensive view of all fuel transactions over time, as each FuelReport is essentially an aggregation of multiple FuelRecords. The FuelReport provides a summarized overview of fuel consumption and costs, making it an essential component for users tracking fuel usage across different timeframes.

In this design, the FuelRecord acts as the fundamental data object, connecting the Image source to the FuelReport output. The many-to-one relationship between FuelRecord and Image, and the many-to-one relationship between FuelRecord and FuelReport, ensures a flexible and scalable architecture for handling fuel data in the Fuel Beacon project.

IV.3 User Interface Design

We designed our GUI with the intention of creating a modern yet navigable and intuitive design for our client. We wanted each component to have a distinct and apparent purpose to minimize ambiguity that may come up. Upon opening the application, the home screen which can be found in the Appendix (Image 1) is shown in a way that allows the user to immediately understand the application’s purpose and capabilities, with clearly labeled buttons for document management, data processing, and report generation. The appearance is minimalist and focused, as shown in the Appendix (Image 1).

Upon clicking on any of the 4 buttons, the UI displays in a way that is clear to the user for how to approach the process. For example, if “ViewData” is selected as shown in the Appendix (Image 2), the user can first select the query type (English / By Metric), shown at the top, then the user inputs the prompt and the output is displayed and labelled clearly below. The flow of the approach is designed from top to bottom as shown in the Appendix (Image 2).

We plan further to modernize the UI for future iterations of our application, making it visually appealing and more navigable for our client. We plan to improve specific areas of the user interface with this modernization. For example, instead of a "Back to Home" button, the user can navigate across the four areas via a tab system, similar to how we would navigate different tabs

in a web browser. We have implemented these changes while keeping our application's simple and minimalistic nature.

V. Test Case Specifications and Results

This section outlines the testing strategy and methods employed to validate Fuel BEACON's functionality, reliability, and performance. It describes the testing environment, unit and integration tests, and the approach to system testing, ensuring that all components meet the specified requirements. The test plan also addresses the use of continuous integration and feedback loops to maintain high quality throughout development.

V.1 Testing Overview

The objective of testing in the Fuel BEACON project is to build confidence that the application reliably and consistently executes all intended functionalities—from processing handwritten fuel forms to generating accurate reports. Our testing strategy combines automated unit and integration tests with manual, end-to-end system tests. Automated tests currently run in our continuous integration (CI) pipeline, ensuring that any future code updates do not break core functionality, while manual tests validate user interactions and system behavior under realistic conditions. [2][8]

For our automated testing, we are utilizing the "pytest" library in Python. This library is designed for all sorts of testing from Unit Testing to Functional Testing. As Fuel BEACON's codebase is entirely within Python, we don't require any additional tools except for visualizing the performance. Viztracer and snakeviz are used to visualize performance in terms of the time and system resources spent in running the VLM. Our integration testing is performed primarily through the pytest library, as each subsystem can be expected to produce specific types of outputs. [8]

However, due to the inherent unpredictability of LLM outputs, we are not able to fully automate the testing process with predicted outputs. System Testing is needed to be almost entirely manual, and Unit Tests involving LLM output are confirmed with manually entered test data. We are also testing User Acceptance, which by definition requires our user, Aaron Keys, to test the product.

Our testing plan is detailed in the following:

- A set of Unit Tests for each subsystem of Fuel BEACON, with tests for each non-trivial function with the subsystem. These tests cover any foreseeable inputs, including edge cases. Some unit tests include testing the database for correct mongodb connection and testing correct data cleaning.

- A set of Integration Tests between each subsystem in the pipeline, ensuring that outputs are standardized as expected. This includes testing the ImageProcessor to correctly interact with the VLM and file cropping.

- A set of Performance Tests between running the VLM and scanning different file types. Data such as runtime and memory is collected and stored in files within the application.

This section outlines the structured test plans designed to ensure Fuel Beacon's functionality, reliability, and user readiness. Through a combination of unit, integration, and system testing, the development team aims to validate each component independently, assess interactions across modules, and evaluate the complete system against its requirements and performance standards.

The testing process is divided into two key workflows:

V.1.1 Developer Testing Process

- **Code Implementation:**
  Developers integrate new features directly into the Fuel BEACON codebase.
- **Test Case Development:**
  For each core functionality, at least two test cases are defined: one for normal operation and one for handling exceptional or invalid inputs.
- **Automated Testing:**
  Developers run all unit and integration tests (using Pytest) to ensure that recent changes meet requirements before pushing code.
- **Continuous Integration:**
  Code is pushed to the remote repository where a CI pipeline automatically runs all tests. A branch is eligible for merge only if all tests pass.
- **Peer Review:**
  Code reviews are performed as part of the pull request process, ensuring quality and adherence to project standards.
- **Manual End-to-End Testing:**
  In addition to automated tests, manual tests are performed to verify the overall system workflow from document upload to report generation.

V.1.2 User Acceptance Testing Process

- **Build Preparation:**
  A stable build of Fuel BEACON is created for end-user testing.
- **Deployment to Testers:**
  The application is deployed to the client, Aaron Keys. It is also demonstrated to the user through Zoom meetings.
- **Feedback Collection:**
  Testers use structured feedback to document their experience, noting usability, performance, and any issues encountered.
- **Data Analysis and Iteration:**
  Feedback is analyzed and prioritized for subsequent iterations, ensuring that the application continues to evolve in line with user needs and expectations.

V.1.3 Unit Testing

Our team follows traditional unit testing procedures. We implement Python's "pytest" unit testing framework to carry this out. The Pytest testing framework was originally inspired by the unittest module in Python and other xUnit-style frameworks like JUnit. It supports test automation, sharing setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. We plan to utilize these features to test all isolated functionalities of the application. We have developed standard tests to validate functionalities when regular input is given. In addition, we have tests for edge cases and

exceptions to test valid uncommon inputs and if the function can handle possible invalid inputs. Our unit tests are conducted on every component in our application except for the UI.

V.1.4 Integration Testing

We also follow traditional integration testing procedures. We test isolated code clusters to validate the functionality of components working together. Since we are developing a full-stack application, we want to validate how different stack levels are integrated. This can include front-end components integrating with back-end components, integration with external language models and libraries, database integration, and other aspects.

V.1.5 System Testing

This section provides an overview of the testing approaches applied to Fuel Beacon, including functional testing to ensure each requirement is met, performance testing to validate system efficiency under expected and stress conditions, and user acceptance testing to confirm that the system aligns with stakeholder needs and is ready for operational deployment.

V.1.5.1 Functional Testing

The Fuel Beacon team's functional testing plan at first relies primarily on manual testing conducted by developers, focusing on the requirements outlined in the Requirements and Specifications document. Manual testing is vital to identify problem areas in this project as the VLM may have different outputs depending on what is inputted as text or for the image in each query. Each functional requirement is paired with a specific functional test to verify that all necessary capabilities of the system operate as expected. Furthermore, unit testing for the functional requirements is performed for every functional requirement such as successful upload of a document and correct JSON procured from a test pdf. Continuous integration is kept in mind for every functional test as with each merge request a GitLab Actions file tests that a document may be successfully uploaded and translated to JSON. These requirements span the ability to upload PNG images, convert them to JSON format using the fine-tuned Qwen-2 model, and store the results securely in local storage. Given the nature of Fuel Beacon as a desktop application and the necessity for offline functionality, developers validate all core functionalities of the system in production. In cases where a functional test fails, the developer performing the test records detailed conditions and creates an issue for further analysis and debugging.

V.1.5.2 Performance Testing

Performance testing for Fuel Beacon focuses on validating non-functional requirements, ensuring that the application operates smoothly and efficiently under various operational scenarios. This includes checking the processing speed of the VLM, memory usage, and CPU load on the specified hardware configuration (Intel i7 processor with 16GB RAM and no dedicated GPU). The team uses profiling tools to gather performance data and observe how the system behaves under typical and heavy usage.

Additionally, stress testing is conducted to evaluate system robustness. The application is run with large files and extended usage times to test its stability and identify performance degradation points. If any performance issues arise during testing, they are documented, and a plan for optimization are put in place to ensure that Fuel Beacon can handle user demands without crashing or experiencing severe slowdowns.

V.1.5.3 User Acceptance Testing

The Fuel Beacon team conducts user acceptance testing in close collaboration with the client, Aaron Keys. This phase confirms that Fuel Beacon meets all requirements specified at the project's inception, ensuring that it is ready for operational deployment.

The following steps will outline our approach to user acceptance testing:

- Resources
    - A stable build of the application with documentation for installation and usage
    - Feedback forms tailored for stakeholder use cases
    - A detailed guide for developers to assist end-users as needed during testing

- Instructions
    - Stakeholders receive a full build of Fuel Beacon for evaluation.
    - The testing period spans a predefined time frame, allowing stakeholders to thoroughly explore the system's functionality.
    - After testing, stakeholders complete feedback forms detailing their experience, any issues encountered, and suggestions for improvement.

- Revision
    - The development team reviews all feedback and identifies key areas for improvement.
    - Based on stakeholder input, the team prioritizes modifications or bug fixes and documents these changes.
    - If a major feature is added or an issue resolved, relevant testing and documentation updates occur.

*This outline is intended to be reiterated for testing purposes.

User acceptance testing is considered successful when stakeholders verify that all core functionality operates as expected, the system performs effectively under normal usage, and stakeholders confirm satisfaction with the system’s readiness for use.

V.2 Environment Requirements

For testing, the application as a whole runs on our client's Windows 11 operating system computers with 16 GB RAM, so our testing environment is conducted on such a system to ensure our application executes in a time range that fits our client's requirements. We implement Python's "pytest" testing framework for testing smaller components since Python is our tech stack's primary tool. The Pytest testing framework was built to account for Python’s built in Pytest framework shortcomings. It supports test automation, sharing setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework. We use this framework to conduct unit, integration, and performance tests. [8]

V.3 Test Results

The following tables detail a series of manual test cases executed to validate core functionalities of Fuel BEACON. Each test case specifies the test ID, the expected inputs and outputs, the actual outputs observed, and the pass/fail result.

Data Testing: This is a series of unit tests created using Pytest to ensure all edge cases are covered.

Test ID	Aspect being tested	Expected Result	Observed Result	Test Result	Test Case Requirements
01	Data saving	Correct data is inserted to the database	Unit test pass	Pass	None
02	MongoDB server closure	Server connection is closed	Unit test pass	Pass	Database connection
03	Variety of record types passed into data cleaning	Keys are formatted to camelCase, dates to datetime, and data has only needed characters	Unit test pass	Pass	Database connection

VLM Accuracy Testing: This is a series of manual and automatic test cases for ensuring correct VLM output. Data discrepancies were found using manual tests and analyzed manually.

Test ID	Aspect being tested	Expected Result	Observed Result	Test Result	Test Case Requirements
01	VLM r11 image scanning	All typed and handwritten data in the form is converted to JSON accurately	Received data for all data areas but data is not accurate for all handwriting	Pass	VLM running
02	VLM r12 image scanning	All typed and handwritten data in the form is converted to JSON accurately	Received data for all data areas but data is not accurate for all handwriting	Pass	VLM running

VLM Performance Testing: This is a series of automatic test cases for testing and visualizing the VLM performance.

Test ID	Aspect being tested	Expected Result	Observed Result	Test Result	Test Case Requirements
01	Test CPU Usace for VLM	Output CPU metrics and ensure they	Received CPU metrics and store	Pass	VLM running

		are below 30 MB (below max usage)	graphical visualization in reports		
02	Test Mass File Runs	Gather data for file run times to compare each one	Ran data for a variety of file sizes and types to compare file run times. Files varied but were similar enough to not find any bugs.	Pass	VLM running
03	Viztracer Profiling	Run viztracer profiling tools to find call stack bugs.	Vistracer ran and no bugs found. The VLM functionality had a large call stack and run time as expected.	Pass	VLM running

Image cropping tests: Images passed into the image cropper have clear visibility and are adjusted correctly.

Test ID	Aspect being tested	Expected Result	Observed Result	Test Result	Test Case Requirements
01	Images passed through the image scanner are cropped with the correct adjustments	Image is correctly cropped to correct sections.	Image is cropped with needed section.	Pass	None

NLP Query Testing: This is a series of manual test cases for ensuring correct queries.

Test ID	Aspect being tested	Expected Result	Observed Result	Test Result	Test Case Requirements
01	Test for record within date	Record received	Record received with query	Pass	Database Connection With Records
02	Test for unavailable date	No records returned	No records	Pass	Database Connection With Records

03	Test for getting specific string grade	Record of specific grade received	Record received with query	Pass	Database Connection With Records
04	Test for getting numeric fuel amount	Record of specific fuel received	Record received with query	Pass	Database Connection With Records

User Testing: This is a series of manual user tests to ensure user satisfaction.

Test ID	Aspect being tested	Expected Result	Observed Result	Test Result	Test Case Requirements
01	Verify UI is satisfactory	Interface is clear, intuitive, and no issues navigating	User was able to navigate and interact with no issues	Pass	GUI Running
02	Verify querying is satisfactory	User enters a query and receives accurate results or a clear “no results” message	Correct records returned or “no records” message shown	Pass	GUI Running, Database Connection With Records
03	Verify reports generation is satisfactory	User can generate report for specific filters with expected output	Report generated correctly based on selected filters	Pass	GUI Running, Database Connection With Records

VI. Projects and Tools Used

Tool/library/framework	Purpose
pytest	Used to make unit, integration, and performance tests [8]
MongoDB	NoSQL database for storing fuel record information. [5]
PyQt6	The GUI framework for building the desktop application. [7]
pymongo	Allows interaction with MongoDB databases using Python. [5]
opencv-python-headless	Image processor for cropping and processing each image. [6]
transformers	Hugging Face’s NLP library to run Qwen-2 [1][7]

spacy	NLP framework used for querying the MongoDB database [4]
qwen-vl-utils	Utility package for working with Qwen-VL, the Vision Language Model used for processing each image to pull out JSON [1][7]
python-dateutil	Used to convert and parse through dates to save as datetimes in the database.
accelerate	Utility library for optimizing large-scale deep learning models
GitHub Actions	YAML files to run CI/CD test pipelines in GitHub [2]

Languages Used in Project
Python

VII. Description of Final Prototype

Fuel Beacon is a document processing and fuel report generation system designed to digitize and streamline fuel transaction record-keeping for logistics and fuel management teams. The system extracts data from handwritten forms and CSV files, flags inconsistencies, and generates structured JSON records for querying and reporting. By automating this process, Fuel Beacon significantly reduces manual data entry time, improves accuracy, and enhances fraud detection capabilities.The system operates through five key stages, each corresponding to a primary function of the software:

- **Home Screen:** Displays four main options—Upload Documents, Scan Documents, View Data, and Generate Report.

Figure 5: Home Page

- **Document Upload and Storage**
  - Users upload handwritten fuel logs (PNG/PDF) and CSV records.
  - The system ensures proper file format validation and secure local storage.
  - Documents can be reviewed before processing.

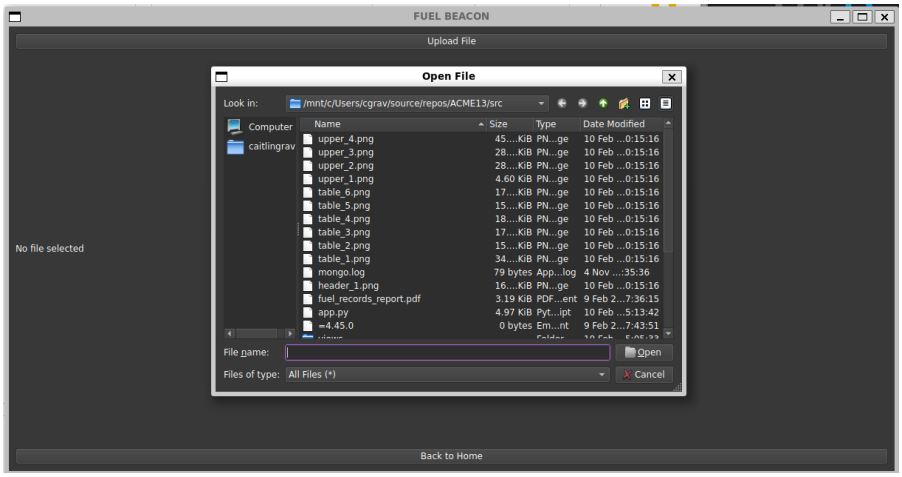


Figure 6: Document Upload

- **Scan Documents:** The system utilizes visual language models (VLMs) to extract handwritten data from uploaded forms and presents the data in a structured format.

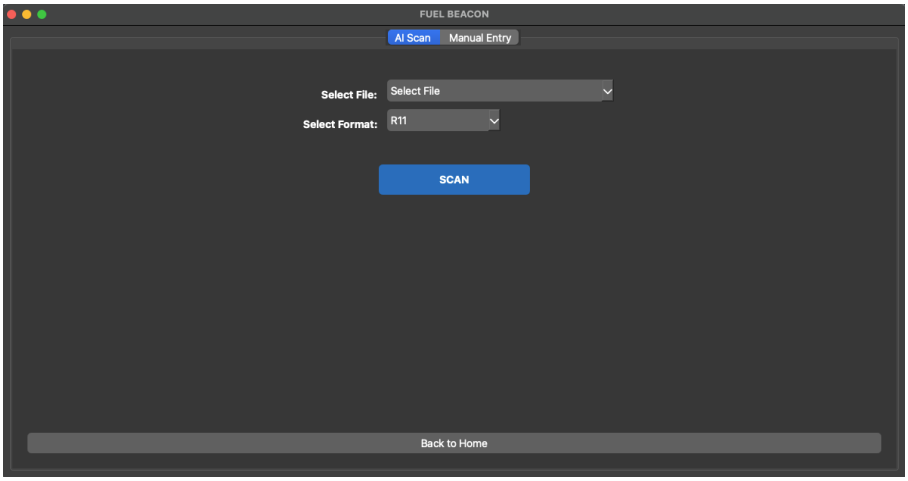


Figure 7: Document Scanning with VLM

- **Fuel Data Query & Analysis**
  - Users can search for fuel transactions using filters (date, vehicle ID, fuel type, location, etc.).
  - The system retrieves relevant records and displays fuel data reports.
  - Historical data can be analyzed for trends and anomalies.

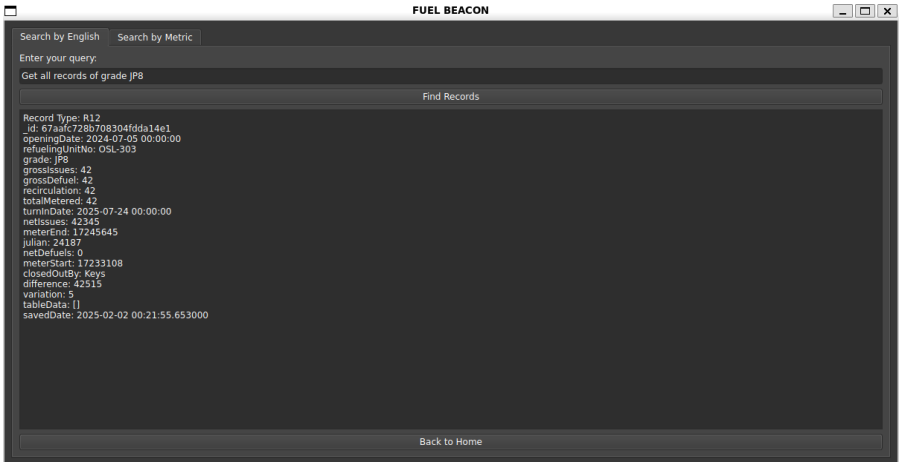


Figure 8: Querying Records

- **Report Generation & Export**
  - Users generate customized PDF reports summarizing fuel usage.
  - Reports are timestamped and can be downloaded, printed, or exported.
  - The system replicates industry-standard fuel reports, automating a previously manual 8-hour process.

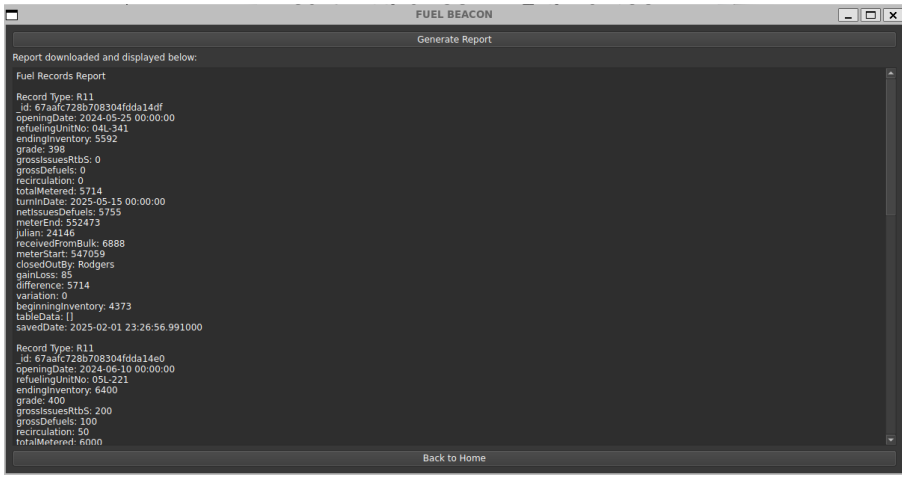


Figure 9: Report Generation

VIII. Product Delivery Status

The final Fuel BEACON project is delivered as a Docker downloadable image and installable on government-issued laptops used by the Pacific Air Forces 673rd Logistics Readiness Squadron. Prior to user testing, the team met with our primary client, Senior Airman Aaron Keys, and relevant IT personnel to facilitate installation and provide any necessary training. Senior Airman Keys has reviewed the Alpha Prototype and subsequent updates provided at the end of each sprint.

Project Location Information:

- The latest release of Fuel BEACON is available under the Releases section in our team's GitHub repository: https://github.com/caitlingraves/ACME13
- The main branch of the GitHub repository contains the stable version of the application.
- A pre-built docker image of Fuel BEACON is available in our team's Docker repository: https://hub.docker.com/r/caitlingraves/acme_app_image

Set Up Instructions:

Given the nature of the Fuel BEACON project, there are two installation guides: one for developers and one for end users. The user-oriented guide includes only the image installation steps.

Prerequisites

- **Developer:**
  - Python 3.12 (or later): Required for running and developing the application.
  - MongoDB: For local testing of the fuel transaction database.
  - Virtual Environment Tools: Such as Venv or Virtualenv to manage dependencies.
  - IDE or Text Editor: (VS Code) for code development.
  - Install all other required packages with the command: pip install -r requirements.txt
- **User:**
  - Docker: For pulling the working image of the application.
  - X Server or other graphical displays: For displaying the application

Installation Steps

- **Developer:**
  - To clone the repository, open a terminal and run: git clone https://github.com/ACME13/FuelBeacon.git
  - Navigate to the Project Directory: cd ACME13
  - Create and Activate a Virtual Environment:
    - On Windows: python -m venv venv venv\Scripts\activate
    - On macOS/Linux: python3 -m venv venv source venv/bin/activate
  - Install Dependencies: pip install -r requirements.txt
  - Execute the main application with: python3.12 src/app.py
- **User:**
  - To pull the Fuel BEACON application using Docker, open a terminal and run: docker pull caitlingraves/acme_app_image
  - Start X Server or your chosen graphical display
  - Start the application with the following command: docker run -e DISPLAY=host.docker.internal:0.0 -p 8000:8000 caitlingraves/acme_app_image

IX. Conclusions and Future Work

This section details the current end-state of Fuel BEACON, with a focus on areas that can be improved or goals that were not reached. An overview of avenues for further improvement are also described, with discussion of possible strategies to overcome hang ups during Fuel BEACON's initial development.

IX.1 Limitations and Recommendations

Handwriting Recognition Accuracy
Currently, the Vision Language Model (VLM) Document Reader is capable of extracting structured data from handwritten fuel logs. However, due to the inherent variability in handwriting styles and quality, the VLM struggles with 100% accuracy in recognizing certain handwritten entries. Some challenges include:

- Poor handwriting legibility leading to misinterpretations.
- Faded or incomplete entries affecting accuracy.
- Numeric misclassification (e.g., misreading a "7" as a "1" or a "0" as an "8").
- Inconsistent confidence levels, requiring frequent manual verification.

To improve accuracy, the team suggests fine-tuning the VLM on a dataset specifically containing handwritten fuel logs with common industry formats. Multiple iterations of each image can also be processed through the VLM, with different preprocessing applied (saturation, threshold, rotation), with the outputs being averaged by consensus. Furthermore, the team suggests that if new versions of open source low memory VLM's are released the current VLM, QWEN-2 should be replaced. [10]

Unofficial Report Format
Currently, our client does not have an official required report to print out. Our query page allows our client to print reports on as needed basis. However, if an official report style is required in the future the implementation needs to be updated. This could include requiring saved report queries to print repeated reports.

Our current report printing page allows for our user to explore the different options for printing records, but in the future a more official one may be needed. As a POC project, this is expected to be built upon if an official product is created in the future.

IX.2 Future Work

The majority of the development for Fuel BEACON is intended to be completed by the end of the second Capstone semester. The team intends to update all code on GitHub to production level and to post a Docker image. However, extensions for the project would allow future improvements. Here are the major improvements expected to be improved on if further time is allowed or a future team continues development on Fuel BEACON:

Implement functionality to Query, Flag, and Interact with Fuel Data
This is the major remaining functionality that we were not able to develop on due to lack of matching data with CSV's. Our current prototype can scan and store the fuel data in the database, however, a key functional requirement for the application is to be able to display the data and have the user interact with it. There should also be a functionality implemented to flag the data based on the document scanning confidence results.

Report Generation - Formatting Improvements
Our last major task for improving the prototype is to effectively generate a report based on the application's fuel data and format it satisfactorily for our client. We currently have a basic implementation of this functionality, displaying unformatted fuel data on the application interface. If given time for future improvements we will further expand on this to format the data and generate a PDF for this formatted data.

Updating VLM Version
Additionally, as our current VLM option was just released at the beginning of this project better versions are expected to be released. In the future if more efficient VLM's are available, it would be best to consider switching to one of these to improve system reliability. However, as technology stands now the Qwen-2 VLM is tested as the best current option for this project. [9]

X. Acknowledgements

For this section, the team would like to acknowledge our client, Senior Airman Aaron Keys, from the Pacific Air Forces 673rd Logistics Readiness Squadron, for his continuous feedback and support in ensuring Fuel BEACON meets operational needs. We also extend our thanks to our project mentor, Professor Parteek Kumar from Washington State University, for his guidance and expertise throughout development. Finally, we appreciate the contributions of our teammates and testers who helped refine the system through their feedback and testing efforts.

XI. Glossary

Continuous Development (CD) - A software development approach that extends continuous integration by automating the deployment of code changes to production environments. [2]

Continuous Integration (CI) - A software development practice where code changes are automatically tested and integrated into a shared repository multiple times a day, helping to catch bugs early and improve overall software quality through immediate feedback and streamlined workflows. [2]

GPU (Graphics Processing Unit) - A specialized processor designed to accelerate computations by performing parallel processing tasks, making it highly effective for training and inference in machine learning models. In machine learning, GPUs significantly reduce training time for complex algorithms, such as deep learning networks, by efficiently handling large datasets and performing numerous calculations simultaneously.

GUI (Graphical User Interface) - A visual interface that allows users to interact with software through graphics, such as windows, buttons, and icons, making applications easier to use than text-based command-line interfaces.

Integration Test - A testing phase in software development where individual units or components of a program are combined and tested as a group to verify that they work together correctly and to detect interface issues between integrated parts. [2]

JSON (JavaScript Object Notation) - A lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate.

LLMs (Large Language Models) - A type of artificial intelligence algorithm that utilizes deep learning techniques and large data sets to understand, manipulate, generate, and predict new content.

MongoDB - A NoSQL, document-oriented database that stores data in flexible, JSON-like documents. It allows for scalable and high-performance data management, supporting dynamic schema and powerful querying capabilities. [5]

POC (Proof of Concept) - A demonstration that verifies the feasibility of a product by showing the concept can be turned into a reality and fulfills customer requirements.

Python - A high-level, versatile programming language known for its readability and ease of use. Python supports multiple programming paradigms, including procedural, object-oriented, and functional programming, making it a popular choice for utilizing with LLM's.

Pull Request (PR) - A request submitted by a developer to merge changes from one branch of a repository into another, typically accompanied by a discussion for code review and collaboration, allowing team members to review, comment, and approve the proposed changes before integration.

PyQt6 - A set of Python bindings for the Qt application framework, enabling the development of cross-platform desktop applications with a rich graphical user interface. [7]

RAM (Random Access Memory) - A type of volatile memory used in computers and devices to store data that is actively being used or processed. It allows for fast read and write access, enabling efficient performance of applications and multitasking.

SQL (Structured Query Language) - A standard programming language used for managing and manipulating relational databases. It enables users to query, insert, update, and delete data in a structured, table-based format, enforcing a predefined schema with relationships between tables.

Unit Test - A software testing method where individual components or functions of a program are tested in isolation to ensure they perform as intended, helping to identify and fix bugs early in the development process. [8]

VLMs (Visual Language Models) - A type of artificial intelligence that combines visual and textual data, enabling it to understand and generate responses based on both images and natural language inputs. [3]

XII. References

[1] “API Inference Documentation.” Hugging Face. Accessed: Oct. 13, 2024. [Online.] Available: https://huggingface.co/docs/api-inference/en/index

[2] “GitHub Actions.” GitHub. Accessed: Nov. 3, 2024. [Online.] Available: https://github.com/features/actions

[3] “Google Gemini LLM explanation and applications.” Google Cloud. Accessed: Oct. 13, 2024. [Online.] Available: https://cloud.google.com/ai/llms

[4] “Library Architecture · Spacy API Documentation.” spacy. Accessed February 14, 2025. https://spacy.io/api

[5] “MongoDB: The Developer Data Platform.” MongoDB. Accessed: Oct. 18, 2024. [Online.] Available: https://www.mongodb.com/

[6] “OpenCV Documentation.” OpenCV. Accessed November 18, 2024. https://opencv.org/

[7] “PyQt6 Tutorial: The Complete Guide.” Python GUIs. Accessed: Oct. 13, 2024. [Online.] Available: https://www.pythonguis.com/pyqt6-tutorial/

[8] “pytest — Unit testing framework.” pytest. Accessed: Nov. 3, 2024. [Online.] Available: https://docs.pytest.org/en/stable/

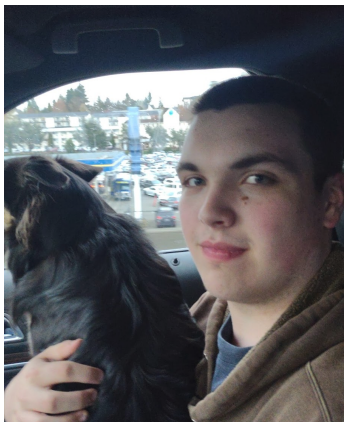
[9] “Vision language model repository.” GitHub. Accessed: Oct. 13, 2024. [Online.] Available: https://github.com/QwenLM/Qwen2-VL

[10] Huang, L., Yao, C., Zhang, L. *et al.* Enhancing computer image recognition with improved image algorithms. *Sci Rep* **14**, 13709 (2024). https://doi.org/10.1038/s41598-024-64193-3

XIII. Appendix A - Team Information

Caitlin Graves

Aidan Johnson



Andrew McGann

Roy Zabetski

XIV. Appendix B - Example Testing Strategy Reporting

For the testing section of this project, the team decided to take a structured approach using conventional testing methodologies, including System, Functional, and Acceptance testing, alongside targeted user testing with our client, Senior Airman Aaron Keys, and his team.

Requirements Being Tested

- **Automated Data Extraction:** Ensuring the system correctly extracts handwritten fuel transaction data.
- **Data Accuracy & Integrity:** Verifying extracted data matches original fuel logs.
- **User Interface & Usability:** Testing ease of use for application UI.
- **Performance & Load Handling:** Testing system responsiveness under varying document loads.

Automated Testing

- **Methodology:** Due to the document-heavy nature of the Fuel BEACON system, the team focused on testing backend components that handle data extraction, processing, and storage. Automated CI/CD GitHub Actions tests were created primarily for the handwritten form conversion, database, and report generation subsystems to validate functionality and error handling. Given the critical role of the Visual Language Model (VLM), extensive tests were conducted to assess the accuracy and consistency of extracted text under different scan qualities.
- **Results:** The following images, figure 10 and 11, depict test cases evaluating expected behavior and edge cases, including situations where the VLM could output poor data. The GitHub Actions pipeline runs successful checks for the VLM and general unit tests. These passing test cases provide confidence that Fuel BEACON processes handwritten documents reliably and maintains data integrity.

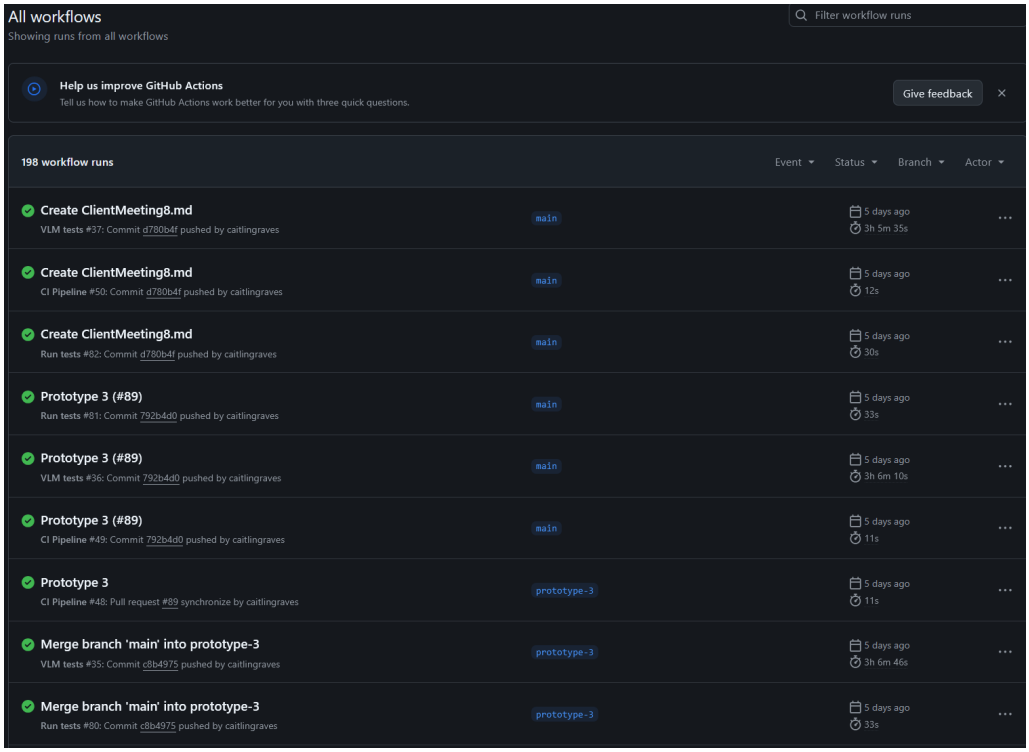


Figure 10: GitHub Actions CI/CD Pipeline

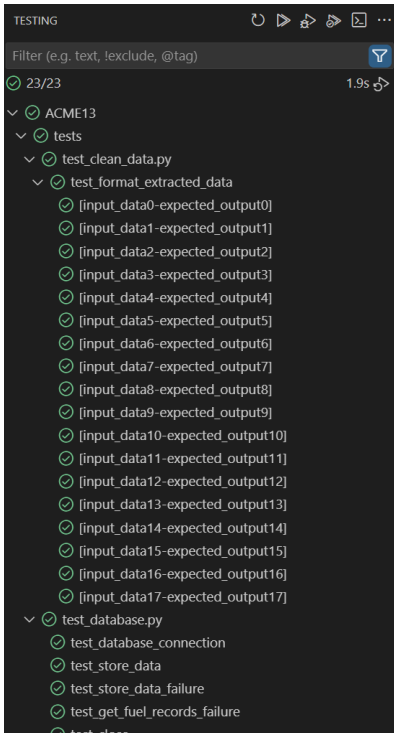


Figure 11: Automated Testing Checks in Visual Studio

User Testing

- **Methodology:** Given the operational nature of Fuel BEACON within the Pacific Air Forces, user testing was conducted with Aaron Keys to ensure the system meets real-world requirements. The team deployed a stable version of the software and observed user interactions while collecting feedback through demos done through Zoom. Questionnaires were also conducted to receive further information. The primary focus was to assess system usability, verify the accuracy of automated data extraction, and identify potential workflow improvements. Users were also encouraged to report any usability concerns or system errors.
- **Results:** We received positive feedback as well as a few critiques from our client in each demo. This information helps refine the report generation process. We also created a way to manually insert data as records if there is low confidence for the VLM's output. Additionally, feedback highlighted minor UI adjustments that could improve user experience.

XV. Appendix C - Project Management

The team's schedule includes biweekly meetings usually done on Tuesdays and Thursdays during the capstone class time. The team discusses what needs to be completed for the week and plans for the upcoming sprint. Each month the team meets with the client, Aaron Keys, to provide an update on code progress, discuss changes, and get feedback. Furthermore, during the month the team meets with mentor Parteek Kumar to provide a progress update.

Team Meetings

The team meets biweekly to discuss progress, assign tasks for the sprint, and address any development challenges. During these meetings, we track issues using a Kanban board to monitor tasks in progress, pending review, and completed. These meetings are done on Zoom, Discord, or in person on the Washington State University campus. The team uses issues to track what needs to be done assigned to team members (Figure 11). A Kanban board is used to track what needs to be completed during the sprint (Figure 12).

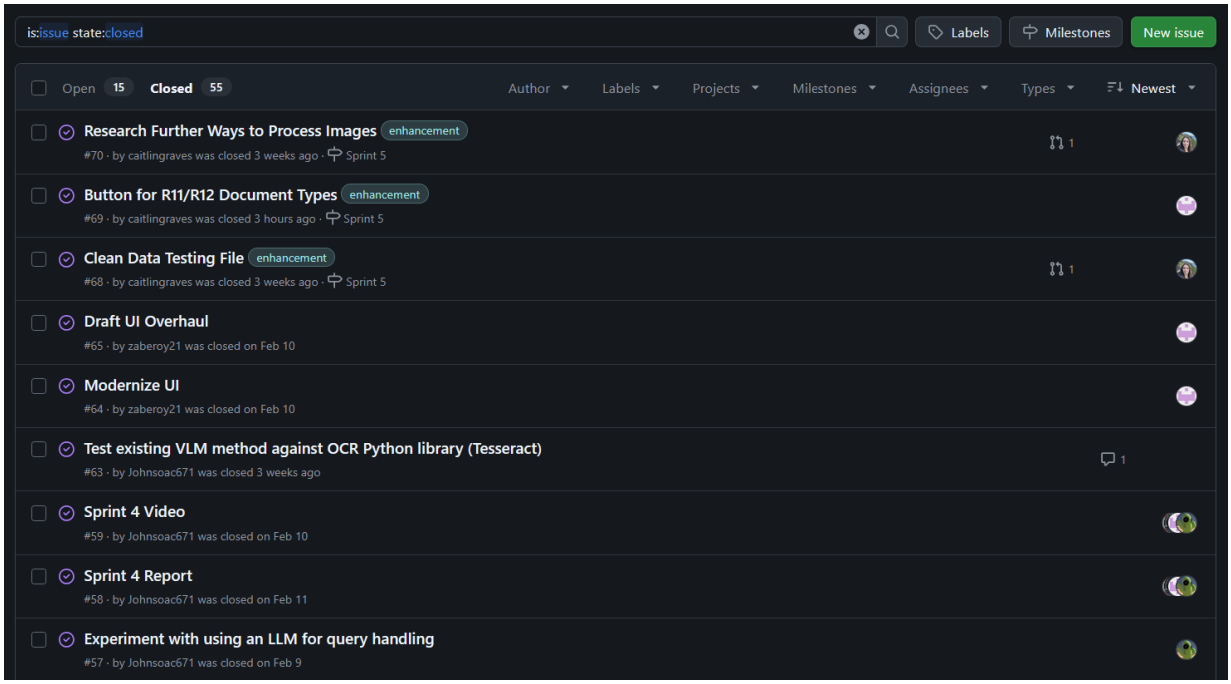


Figure 11: Finished Issues

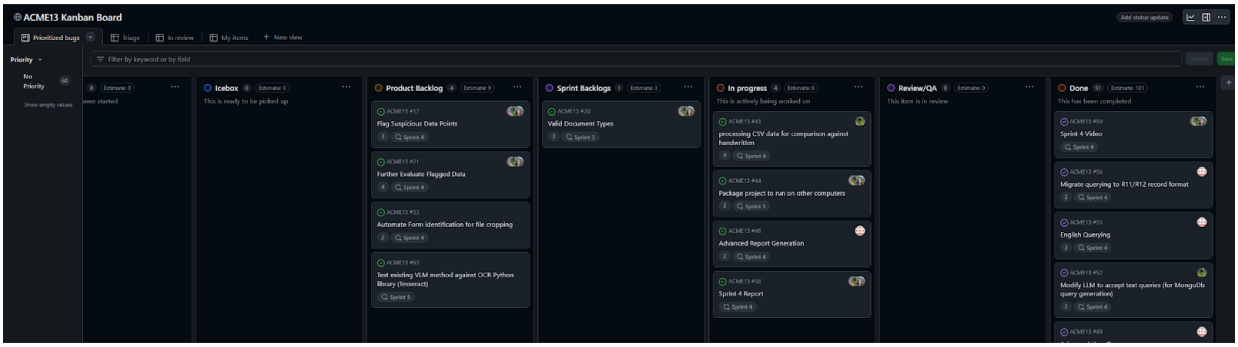


Figure 12: Kanban Board

Client Meetings

Every month, the team meets with Aaron Keys to present progress updates and gather feedback. Meetings focus on refining the user experience, validating extracted data accuracy, and ensuring the system aligns with operational needs. As user testing progresses, these meetings included discussions on observed challenges and recommended improvements. These meetings are conducted over Zoom and coordinated via email with an Outlook invite sent to all members.

Mentor Meetings

Once a month, the team meets with Professor Parteek Kumar to review technical progress and documentation. These discussions help refine the testing strategy, troubleshoot any development roadblocks, and ensure Fuel BEACON adheres to best software engineering practices. Meetings are conducted in person at Professor Parteek Kumar's office, with a set schedule posted on Canvas.