Module Guide for Software Engineering

Team #, Team Name
Willie Pai
Hammad Pathan
Wajdan Faheen
Henushan Balachandran
Zahin Hossain

January 18, 2025

1 Revision History

Date	Version	Notes
January 13, 2025	1.0	Initial Revision
January 17, 2025	1.1	Final changes made according to rubric

2 Reference Material

This section records information for easy reference.

2.1 Abbreviations and Acronyms

symbol	description
AC	Anticipated Change
DAG	Directed Acyclic Graph
M	Module
MG	Module Guide
OS	Operating System
R	Requirement
SC	Scientific Computing
SRS	Software Requirements Specification
Software Engineering	Explanation of program name
UC	Unlikely Change
[etc. —SS]	[... —SS]

Contents

List of Tables

List of Figures

3 Introduction

Decomposing a system into modules is a commonly accepted approach to developing software. A module is a work assignment for a programmer or programming team (Parnas et al., 1984). We advocate a decomposition based on the principle of information hiding (Parnas, 1972). This principle supports design for change, because the "secrets" that each module hides represent likely future changes. Design for change is valuable in SC, where modifications are frequent, especially during initial development as the solution space is explored.

Our design follows the rules layed out by Parnas et al. (1984), as follows:

- System details that are likely to change independently should be the secrets of separate modules.
- Each data structure is implemented in only one module.
- Any other program that requires information stored in a module's data structures must obtain it by calling access programs belonging to that module.

After completing the first stage of the design, the Software Requirements Specification (SRS), the Module Guide (MG) is developed (Parnas et al., 1984). The MG specifies the modular structure of the system and is intended to allow both designers and maintainers to easily identify the parts of the software. The potential readers of this document are as follows:

- New project members: This document can be a guide for a new project member to easily understand the overall structure and quickly find the relevant modules they are searching for.
- Maintainers: The hierarchical structure of the module guide improves the maintainers' understanding when they need to make changes to the system. It is important for a maintainer to update the relevant sections of the document after changes have been made.
- Designers: Once the module guide has been written, it can be used to check for consistency, feasibility, and flexibility. Designers can verify the system in various ways, such as consistency among modules, feasibility of the decomposition, and flexibility of the design.

The rest of the document is organized as follows. Section 4 lists the anticipated and unlikely changes of the software requirements. Section 5 summarizes the module decomposition that was constructed according to the likely changes. Section 6 specifies the connections between the software requirements and the modules. Section 7 gives a detailed description of the modules. Section 8 includes two traceability matrices. One checks the completeness of the design against the requirements provided in the SRS. The other shows the relation between anticipated changes and the modules. Section 9 describes the use relation between modules.

4 Anticipated and Unlikely Changes

This section lists possible changes to the system. According to the likeliness of the change, the possible changes are classified into two categories. Anticipated changes are listed in Section 4.1, and unlikely changes are listed in Section 4.2.

4.1 Anticipated Changes

Anticipated changes are the source of the information that is to be hidden inside the modules. Ideally, changing one of the anticipated changes will only require changing the one module that hides the associated decision. The approach adapted here is called design for change.

AC1: Supporting additional hardware, such as new touchscreen displays or tablets, to ensure compatibility with future device upgrades and varying specifications.

AC2: Modifying input data formats to align with updated composite structures or metadata provided by Lifetouch.

AC3: Enhancing data processing workflows to adapt to different composite formats and improve OCR preprocessing accuracy.

AC4: Scaling the system to include graduation composites from additional faculties or departments across the university.

AC5: Updating the OCR model to accommodate higher-resolution composites or address accuracy improvements in text extraction.

AC6: Adding new user roles, such as external recruiters or event organizers, to expand the system's usability and access controls.

AC7: Redesigning the user interface to meet evolving accessibility standards and usability feedback for a better experience.

AC8: Extending the database schema to store new metadata, such as additional student achievements or career milestones.

AC9: Integrating the system with McMaster's existing alumni and event management platforms to streamline user engagement.

AC10: Adjusting backend processes to comply with updates to privacy policies, legal regulations, or internal university standards.

[Anticipated changes relate to changes that would be made in requirements, design or implementation choices. They are not related to changes that are made at run-time, like the values of parameters. —SS]

4.2 Unlikely Changes

The module design should be as general as possible. However, a general system is more complex. Sometimes this complexity is not necessary. Fixing some design decisions at the system architecture stage can simplify the software design. If these decision should later need to be changed, then many parts of the design will potentially need to be modified. Hence, it is not intended that these decisions will be changed.

UC1: The source of the input data will always be from predefined repositories, such as LifeTouch or McMaster databases.

UC2: The system will always display graduation composites on touchscreen interfaces and other devices as defined during development.

UC3: The primary goal of the system will remain focused on facilitating the browsing and searching of graduation composites for students, alumni, and staff.

UC4: The search functionality will always rely on a combination of Optical Character Recognition (OCR) data and metadata stored within the system.

UC5: The backend database architecture will remain relational (or NoSQL, as selected initially) to meet system scalability and storage requirements.

UC6: The system will only support English as the primary language for search and display functionalities.

UC7: The project will be exclusively deployed on McMaster's internal infrastructure, adhering to university IT policies and avoiding external hosting solutions.

UC8: The privacy and security protocols implemented will always follow GDPR and university policies, ensuring compliance with data handling standards.

5 Module Hierarchy

This section provides an overview of the module design. Modules are summarized in a hierarchy decomposed by secrets in Table 1. The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

M1: Cloud

M2: Input Format

M3: Data Upload

M4: OCR Processing

M5: Output Storage

M6: UI Parsing

M7: Graphical User Interface

Level 1	**Level 2**
Hardware-Hiding Module	Cloud
Behaviour-Hiding Module	Input Format Data Upload OCR Processing Output Storage
Software Decision Module	UI Parsing Graphical User Interface

Table 1: Module Hierarchy

6 Connection Between Requirements and Design

The design of the system is intended to satisfy the requirements developed in the SRS. In this stage, the system is decomposed into modules. The connection between requirements and modules is listed in Table 2.

[The intention of this section is to document decisions that are made "between" the requirements and the design. To satisfy some requirements, design decisions need to be made. Rather than make these decisions implicit, they are explicitly recorded here. For instance, if a program has security requirements, a specific design decision may be made to satisfy those requirements with a password. —SS]

7 Module Decomposition

Modules are decomposed according to the principle of "information hiding" proposed by Parnas et al. (1984). The *Secrets* field in a module decomposition is a brief statement of the design decision hidden by the module. The *Services* field specifies *what* the module will do without documenting *how* to do it. For each module, a suggestion for the implementing software is given under the *Implemented By* title. If the entry is *OS*, this means that the module is provided by the operating system or by standard programming language libraries. *Software Engineering* means the module will be implemented by the Software Engineering software.

Only the leaf modules in the hierarchy have to be implemented. If a dash (–) is shown, this means that the module is not a leaf and will not have to be implemented.

7.1 Hardware Hiding Modules

Secrets: The data structure and algorithm used to implement the virtual hardware.

Services: Serves as a virtual hardware used by the rest of the system. This module provides the interface between the hardware and the software. So, the system can use it to display outputs or to accept inputs.

Implemented By: OS

7.1.1 Cloud Module (M1)

Secrets: The SDKs, APIs, and configurations used to interact with AWS (S3, Lambda, and DynamoDB), including details of network protocols, security settings, and hardware infrastructure.

Services: Provides a virtual hardware layer by abstracting cloud storage (Amazon S3), serverless compute (AWS Lambda), and database operations (Amazon DynamoDB). Allows the system to upload and retrieve data, execute image processing logic, and store processed metadata seamlessly without exposing hardware details.

Implemented By: AWS Services and SDKs

Type of Module: Abstract Object

7.2 Behaviour-Hiding Module

Secrets: The contents of the required behaviours.

Services: Includes programs that provide externally visible behaviour of the system as specified in the software requirements specification (SRS) documents. This module serves as a communication layer between the hardware-hiding module and the software decision module. The programs in this module will need to change if there are changes in the SRS.

Implemented By: –

7.2.1 Input Module (M2)

Secrets: The structure of the image and metadata provided by the client (admin). Includes details such as accepted file formats, metadata keys, and expected payload structure.

Services: Converts client-provided data (composite image) into a format suitable for uploading to S3 and processing by Lambda. Ensures compliance with predefined input standards.

Implemented By: Back-End Entity

Type of Module: Library

7.2.2 Data Upload Module (M3)

Secrets: The configuration of API endpoints, security mechanisms (e.g., HTTPS, authentication), and retry logic for uploads.

Services: Handles the transfer of input data (image) to Amazon S3. Encapsulates the logic for SDK interactions, upload retries, and error handling.

Implemented By: Back-End Entity and AWS S3 Entity

Type of Module: Abstract Object

7.2.3 OCR Processing Module (M4)

Secrets: The OCR algorithm and processing logic for extracting names and center points of individuals from composite images.

Services: Processes the uploaded images to extract required metadata (e.g., individual names and center points). Transforms the processed data into a format ready for storage in DynamoDB.

Implemented By: AWS Lambda

Type of Module: Library

7.2.4 Output Storage Module (M5)

Secrets: The schema for DynamoDB, including primary keys, data types, and indexing strategies.

Services: Stores extracted data and metadata in DynamoDB for later retrieval. Abstracts database operations such as creating, reading, updating, and deleting records.

Implemented By: DynamoDB Entity

Type of Module: Abstract Data Type

7.3 Software Decision Module

Secrets: The design decision based on mathematical theorems, physical facts, or programming considerations. The secrets of this module are *not* described in the SRS.

Services: Includes data structure and algorithms used in the system that do not provide direct interaction with the user.

Implemented By: –

7.3.1 UI Parsing Module (M6)

Secrets: The implementation logic for uploading images and viewing composite images through user interface and interactions.

Services: Implements rules for managing workflows, such as fallback mechanisms for failed uploads or processing. Encodes data transformation rules for formatting outputs.

Implemented By: Back-End Logic and AWS Lambda

Type of Module: Library

7.3.2 Graphical User Interface Module (M7)

Secrets: User interaction event handling, user input controls, states, The methods to implement data formats (such as text-boxes, buttons, file dialogs), visual layout and styling (eg. colours or sizing) for user interaction.

Services: Provides methods for building a visual interface for the user to see and interact with.

Implemented By: React

Type of Module: Library

8 Traceability Matrix

This section shows two traceability matrices: between the modules and the requirements and between the modules and the anticipated changes.

Req.	Modules
R1	M2, M3, M6, M7
R2	M1, M2, M3, M4, M5, M6
R3	M2, M7
R4	M5, M7
R5	M1, M5, M7
R6	M1, M2

Table 2: Trace Between Requirements and Modules

AC	Modules
AC1	M7
AC2	M2
AC3	M4
AC4	M1, M5
AC5	M4
AC6	M2, M7
AC7	M7
AC8	M5
AC9	M1
AC10	M1, M2

Table 3: Trace Between Anticipated Changes and Modules

9 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. Parnas (1978) said of two programs A and B that A *uses* B if correct execution of B may be necessary for A to complete the task described in its specification. That is, A *uses* B if there exist situations in which the correct functioning of A depends upon the availability of a correct implementation of B. Figure 6 illustrates the use relation between the modules. It can be seen that the graph is a directed acyclic graph (DAG). Each level of the hierarchy offers a testable and usable subset of the system, and modules in the higher level of the hierarchy are essentially simpler because they use modules from the lower levels.

[The uses relation is not a data flow diagram. In the code there will often be an import statement in module A when it directly uses module B. Module B provides the services that module A needs. The code for module A needs to be able to see these services (hence the import statement). Since the uses relation is transitive, there is a use relation without an import, but the arrows in the diagram typically correspond to the presence of import statement. —SS]

[If module A uses module B, the arrow is directed from A to B. —SS]

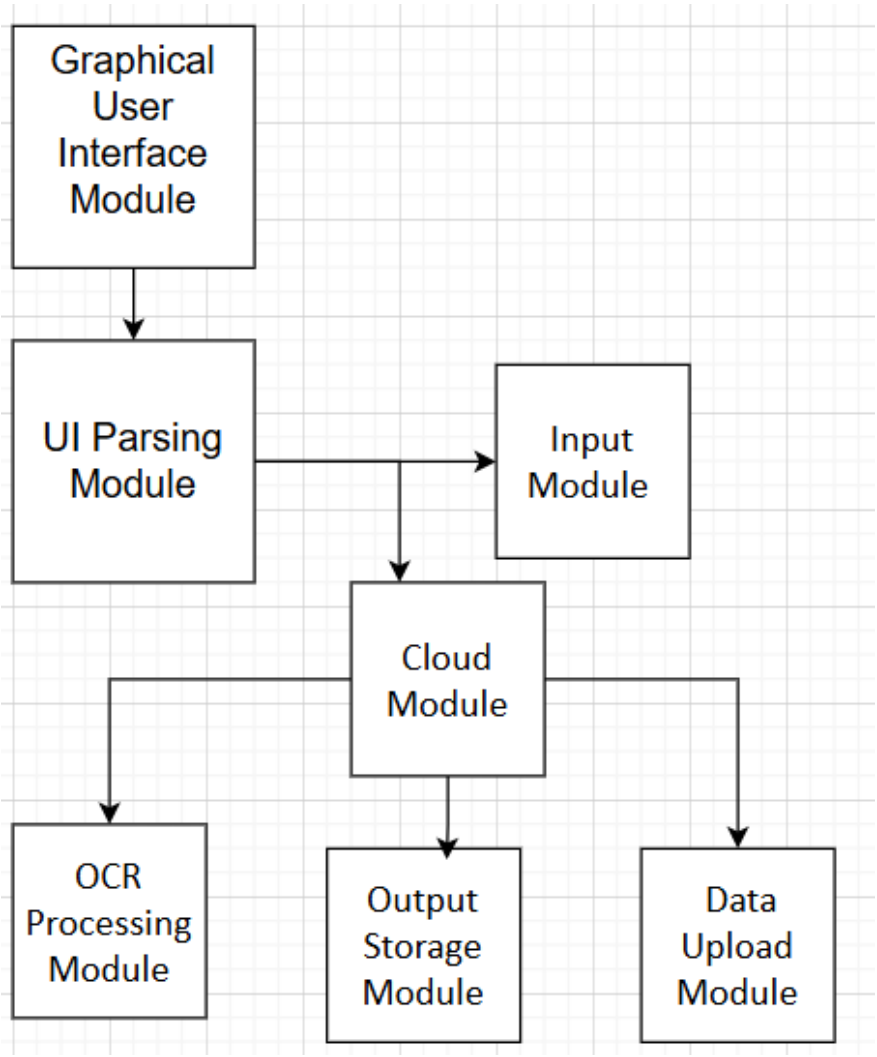

Figure 1: Use hierarchy among modules

10 User Interfaces

[Design of user interface for software and hardware. Attach an appendix if needed. Drawings, Sketches, Figma —SS]

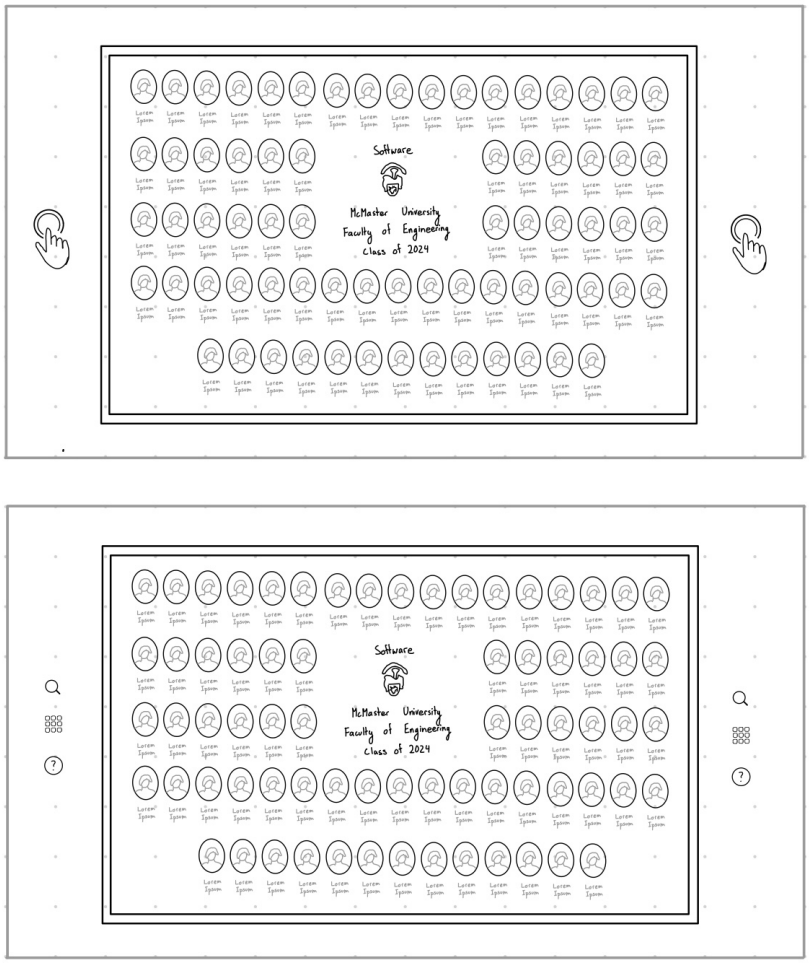

Figure 2: User Interface Design

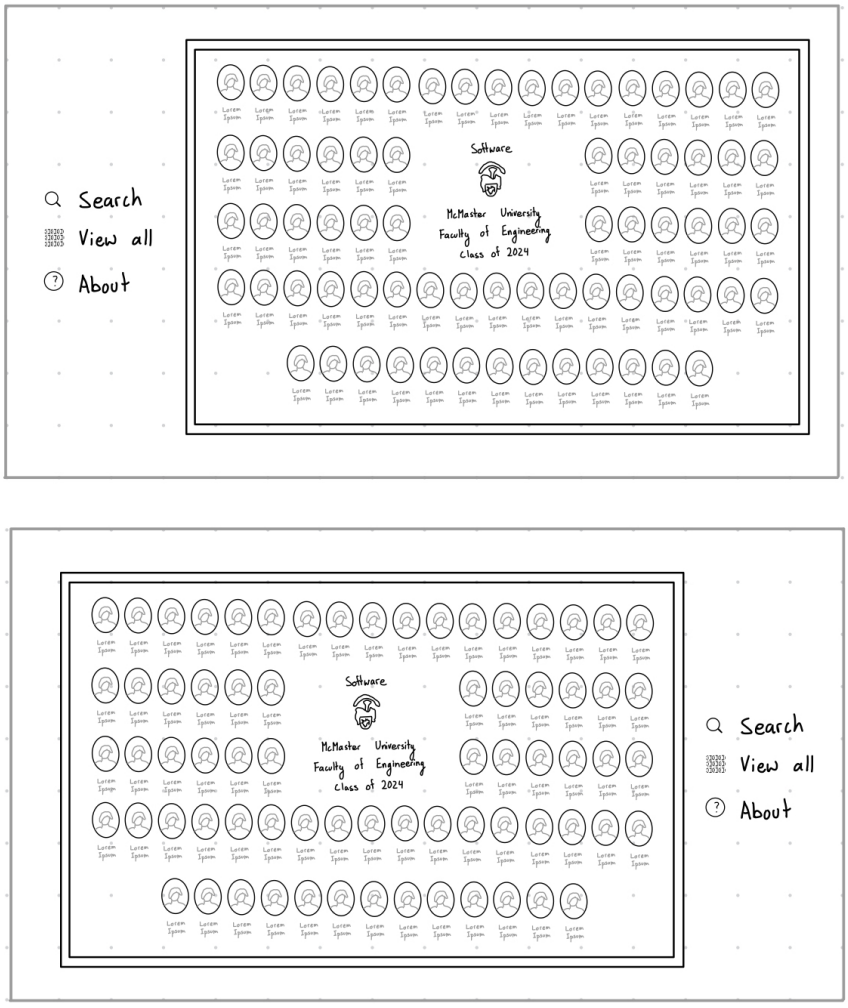

Figure 3: User Interface Design

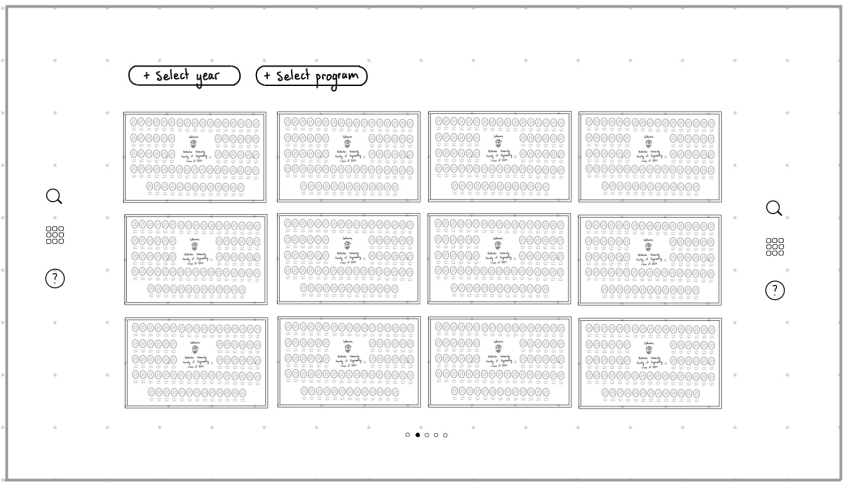

Figure 4: User Interface Design

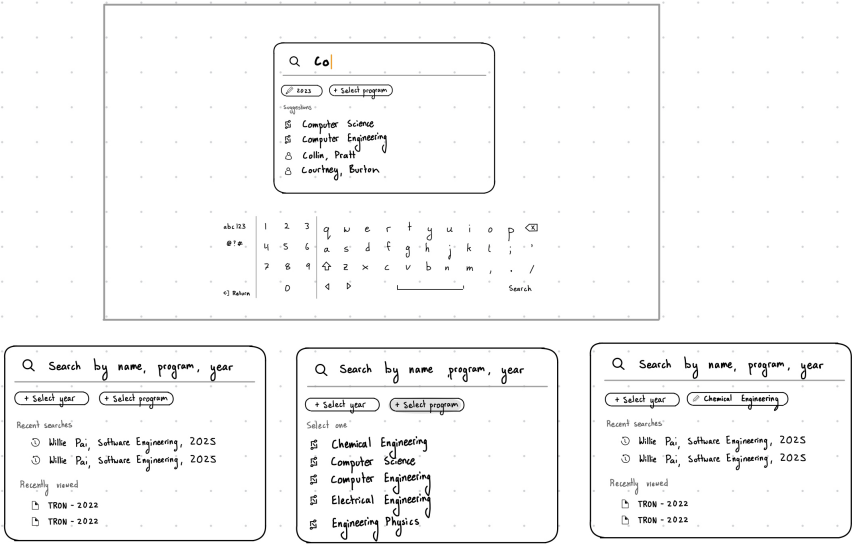

Figure 5: User Interface Design

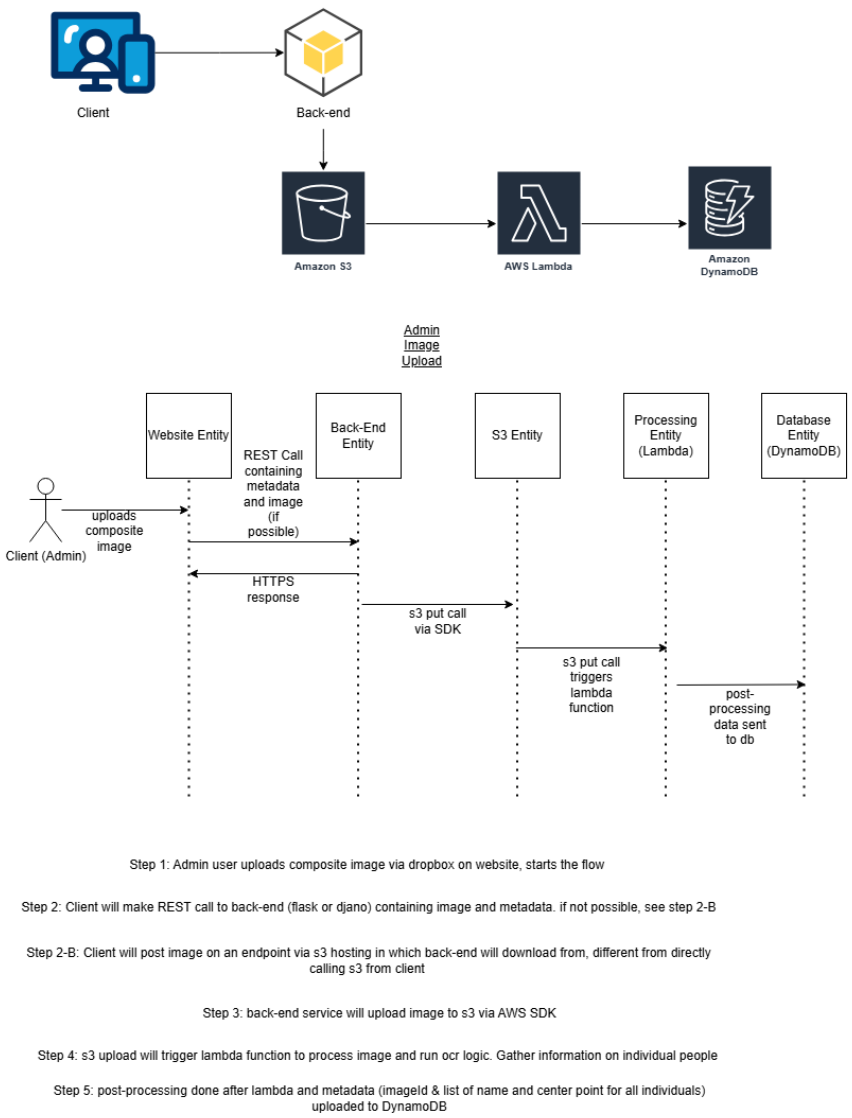

Figure 6: Interface Software Architecture Design

11 Design of Communication Protocols

[NA —SS]

12 Timeline

[Schedule of tasks and who is responsible —SS] [You can point to GitHub if this information is included there —SS]

12.1 Phase: Requirements Gathering and System Design

Dates: Sept 23 – Oct 9
Tasks:

- Define functional and nonfunctional requirements
- Create high-level system architecture

Responsible:

- All team members collaboratively
- Project Lead oversees progress

12.2 Phase: Front-End and Back-End Development

Dates: Oct 9 – Nov 22
Tasks:

- Develop front-end interface using React
- Implement back-end logic for data processing and storage
- Integrate APIs for OCR and composite data retrieval

Responsible:

- Front-End: Willie, Henushan
- Back-End: Zahin, Wajdan, Henushan, Hammad

12.3 Phase: Proof of Concept and Testing

Dates: Nov 22 – Feb 3
Tasks:

- Deploy proof-of-concept prototype
- Perform unit, integration, and usability testing
- Address security and privacy compliance

Responsible:

- Testing Lead: Hammad
- All members participate in debugging and verification

12.4 Phase: Final Deployment and Demonstrations

Dates: Feb 3 – Mar 30
Tasks:

- Complete final system deployment on McMaster servers
- Conduct usability testing and refine interface
- Prepare for final demonstration and Expo presentation

Responsible:

- All members
- Final Presentation Lead: Willie

12.5 Phase: Post-Deployment Support and Documentation

Dates: Mar 30 – Apr 2
Tasks:

- Ensure system maintenance procedures are documented
- Provide user and administrator guides

Responsible:

- Zahin, Hammad

12.6 Task Tracking

Description: Real-time task management and updates
GitHub Repository: https://github.com/PaisWillie/Digital-Composite

References

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(2):1053–1058, December 1972.

David L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press. ISBN none.

D.L. Parnas, P.C. Clement, and D. M. Weiss. The modular structure of complex systems. In *International Conference on Software Engineering*, pages 408–419, 1984.