Module Guide for Software Engineering

Team #1, Sanskrit Ciphers
Omar El Aref
Dylan Garner
Muhammad Umar Khan
Aswin Kuganesan
Yousef Shahin

November 13, 2025

1 Revision History

Date	Version	Notes
November 13	1.0	Added sections 2-5
November 13	1.1	Added sections 6-8
November 13	1.2	Added sections 9-12

2 Reference Material

This section records information for easy reference.

2.1 Abbreviations and Acronyms

symbol	description
AC	Anticipated Change
AI	Artificial Intelligence
API	Application Programming Interface
CNN	Convolutional Neural Network
CV	Computer Vision
DAG	Directed Acyclic Graph
JSON	JavaScript Object Notation
M	Module
MG	Module Guide
ML	Machine Learning
OCR	Optical Character Recognition
OS	Operating System
R	Requirement
SRS	Software Requirements Specification
UI	User Interface
UC	Unlikely Change

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) (Ciphers, 2025), 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: The specific hardware on which the software is running.

AC2: The format of the image input data (file formats, metadata structure).

AC3: The authentication and authorization mechanism for different user types.

AC4: The method for uploading and storing fragment images.

AC5: The algorithm used for searching and filtering fragments.

AC6: The machine learning model used for edge matching detection.

AC7: The algorithm for damage pattern analysis.

AC8: The method for script classification and similarity matching.

AC9: The optical character recognition engine used for text extraction.

AC10: The algorithm for text similarity comparison.

AC11: The user interface framework for the interactive canvas.

AC12: The database schema and storage mechanism for fragment data.

AC13: The structure of catalog metadata.

AC14: The orchestration logic for combining multiple ML model results.

AC15: The session management and state persistence mechanism.

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: Input/Output devices (Input: Image files via web upload, Output: Screen display and file export).

UC2: The system will be deployed as a web-based application accessible through standard browsers.

UC3: The system will process 2D images of manuscript fragments.

UC4: The system will support collaborative work through shared projects and annotations.

UC5: Machine learning will be a core component for fragment matching and analysis.

5 Module Hierarchy

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

M1: HW.ImageStorage Module

M2: UI.Canvas Module

M3: UI.Search Module

M4: UI.Auth Module

M5: UI.Upload Module

M6: Svc.API Module

M7: Svc.Search Module

M8: Svc.AuthZ Module

M9: Svc.Session Module

M10: Svc.ProcOrch Module

M11: Svc.MatchOrch Module

M12: Data.FragmentStore Module

M13: Data.Catalog Module

M14: Data.User Module

M15: Data.Project Module

M16: ML.EdgeMatch Module

M17: ML.Damage Module

M18: ML.ScriptClass Module

M19: ML.OCR Module

M20: ML.TextSim Module

M21: Error Handling Module

M22: Logging Module

M23: Configuration Management Module

M24: Test Stubs Module

Level 1	**Level 2**
Hardware-Hiding	Image Storage Interface
Behaviour-Hiding	User Interface: UI.Canvas, UI.Search, UI.Auth, UI.Upload Service Layer: Svc.API, Svc.Search, Svc.AuthZ, Svc.Session, Svc.ProcOrch, Svc.MatchOrch Data Management: Data.FragmentStore, Data.Catalog, Data.User, Data.Project Machine Learning Modules: ML.EdgeMatch, ML.Damage, ML.ScriptClass, ML.OCR, ML.TextSim
Software Decision	Utility and Support Modules (Error Handling, Logging, Configuration Management, Test Stubs)

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 1.

FR1.1 (User Authentication) is implemented through the M4 and M8 modules which manage credential verification and secure access control. Similarly, FR1.2 (Fragment Management) and FR1.3 (Display Fragments) are supported by the M2, M5, and M12 modules that enable fragment visualization and organization. The ML-oriented requirements FR3.1-FR3.3 are implemented by specialized modules such as M16, M17, and M18, coordinated by M11.

Non-functional requirements are distributed across multiple layers of the system. NFR1.1 (Usability) and NFR1.2 (Performance) are primarily addressed by front-end and orchestration modules to ensure responsiveness and ease of interaction. NFR2.1 (Scalability) and NFR2.2 (Reliability) are achieved through the use of modular orchestration, structured data storage, and robust error handling in modules such as M10, M13, and M21. Finally, NFR3.1-NFR3.3 (Accuracy, Efficiency, and Maintainability) govern the design of the AI/ML cluster, ensuring that the models (M16, M17, M20) can be retrained, tuned, and maintained without disrupting other components.

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 HW.ImageStorage (M1)

Secrets: Data structure and algorithms used to implement the image storage

Services: This module provides platform-specific file operations such as image storage and retrieval, and it serves as virtual hardware for the system.

Implemented By: Cloud or Local Storage Interface

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 UI.Canvas (M2)

Secrets: The implementation of the interactive canvas for fragment manipulation.

Services: Provides an interactive interface to arrange multiple fragments and displays match suggestions

Implemented By: Front-End Web Client (React/Next.js component)

Type of Module: Abstract Object

7.2.2 UI.Search (M3)

Secrets: The user interface design for search and filtering functionality.

Services: Provides a search interface for fragments, and supports search by script type, line count, and fragment size.

Implemented By: Front-End Web Client (React/Next.js component)

Type of Module: Abstract Object

7.2.3 UI.Auth (M4)

Secrets: The user interface for authentication and user management.

Services: Provides login/logout interface, user registration, role-based access control UI, and session management display.

Implemented By: Front-End Authentication Component

Type of Module: Abstract Object

7.2.4 UI.Upload (M5)

Secrets: The user interface for uploading fragment images and metadata.

Services: Provides a file upload interface to select and submit images to the backend for processing after validation.

Implemented By: Front-End Web Client (React/Next.js component)

Type of Module: Abstract Object

7.2.5 Svc.API (M6)

Secrets: The RESTful API endpoint definitions and routing logic.

Services: Sends incoming HTTP requests from the UI to services and handles request validation

Implemented By: Backend API Gateway (FastAPI, Flask, or Node.js)

Type of Module: Abstract Object

7.2.6 Svc.Search (M7)

Secrets: The search algorithm and query optimization logic.

Services: This module performs efficient fragment searches over metadata and fragments, and it can export the results as a PDF or CSV file

Implemented By: Backend Search Service

Type of Module: Abstract Object

7.2.7 Svc.AuthZ (M8)

Secrets: The authentication and authorization implementation.

Services: This module validates user credentials and tokens, and it enforces access control for protected operations

Implemented By: Authentication & Authorization Microservice

Type of Module: Abstract Object

7.2.8 Svc.Session (M9)

Secrets: The session state management and persistence mechanism.

Services: Manages user sessions and provides storage and retrieval of the saved user workspace

Implemented By: Backend Session Manager

Type of Module: Abstract Object

7.2.9 Svc.ProcOrch (M10)

Secrets: The orchestration logic for image processing pipelines.

Services: Coordinates preprocessing workflows for images uploaded by user, and manages job queues for ML tasks

Implemented By: Workflow Orchestrator

Type of Module: Abstract Object

7.2.10 Svc.MatchOrch (M11)

Secrets: The algorithm for combining results from multiple ML models.

Services: Aggregates and sorts results from all ML models and provides confidence scores

Implemented By: Backend Orchestrator Service

Type of Module: Abstract Object

7.2.11 Data.FragmentStore (M12)

Secrets: The database schema and queries for fragment storage.

Services: Stores and retrieves fragment entities such as fragment IDs, associated image IDs, script labels, line counts, and status flags

Implemented By: Data Access Layer

Type of Module: Abstract Data Type

7.2.12 Data.Catalog (M13)

Secrets: The catalog metadata structure and taxonomy.

Services: Stores mappings between internal fragments and external catalogue identifiers

Implemented By: Metadata Database

Type of Module: Abstract Data Type

7.2.13 Data.User (M14)

Secrets: The user data model and storage.

Services: Stores all user information such as accounts, credentials, roles and profile information for authentication and access control purposes

Implemented By: User Account Database Table

Type of Module: Record

7.2.14 Data.Project (M15)

Secrets: The project data structure for collaborative work.

Services: Manages project workspaces and collections and stores project-specific fragment groupings

Implemented By: Project Database Schema

Type of Module: Record

7.2.15 ML.EdgeMatch (M16)

Secrets: Model details including weights and hyperparameters, segmentation mask and edge matching algorithms.

Services: Provides edge match similarity probability and computes edge features for images.

Implemented By: AI/ML Cluster (Python-based ML Service)

Type of Module: Abstract Object

7.2.16 ML.Damage (M17)

Secrets: Physical damage information and hyperparameters for damage segmentation.

Services: Determines similarity score for damage and generates signatures for damage in fragments.

Implemented By: AI/ML Cluster (Python-based ML Service)

Type of Module: Abstract Object

7.2.17 ML.ScriptClass (M18)

Secrets: Model architecture and hyperparameters, and script label and metadata.

Services: Determines the type of script based on text in the script and provides the script label with a confidence score.

Implemented By: AI/ML Cluster (Text Classification Service)

Type of Module: Abstract Object

7.2.18 ML.OCR (M19)

Secrets: Hyperparameters, weights, thresholds and algorithms used for OCR model

Services: This module analyzes fragments to find transcribed text with associated confidence scores, and it provides approximate accuracy values.

Implemented By: AI/ML Cluster (OCR Model Server)

Type of Module: Abstract Object

7.2.19 ML.TextSim (M20)

Secrets: Vectorization and normalization methods, and algorithm used for checking similarity

Services: Converts text to embedding vector and compares similarity between text segments using OCR or manual transcription

Implemented By: AI/ML Cluster (Text Similarity Service)

Type of Module: Abstract Object

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 Util.Error (M21)

Secrets: The error handling and exception management plan

Services: Defines standard error types and provides centralized error handling across all modules

Implemented By: Shared Backend Utility Package

Type of Module: Library

7.3.2 Util.Logging (M22)

Secrets: The logging implementation and structure of the log format

Services: This module records system events and user actions, and it provides structured logging for information, warnings and errors.

Implemented By: Shared Logging Framework

Type of Module: Library

7.3.3 Util.Config (M23)

Secrets: The configuration format and loading mechanism

Services: Loads configuration from files and provides read-only access to configuration values by providing the value or boolean based on key given.

Implemented By: Configuration Manager

Type of Module: Library

7.3.4 Util.TestStub (M24)

Secrets: The mock implementations and mock model outputs

Services: This module provides stub implementations of services and ML components, and it simulates ML model responses

Implemented By: Test Utilities

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**
FR1.1	M4, M8, M9, M14
FR1.2	M2, M5, M6, M10, M12, M13
FR1.3	M2, M3, M6
FR1.4	M2, M11, M16, M17, M18, M19, M20, M12
FR1.5	M2, M4, M9, M15
NFR1.1	M2, M3, M4, M5
NFR1.2	M1, M2, M5, M6
FR2.1	M6, M8, M9, M11
FR2.2	M8, M9, M14
FR2.3	M12, M13, M14, M15
NFR2.1	M6, M10, M11, M1
NFR2.2	M12, M13, M14, M15, M21, M22, M23
FR3.1	M16, M11
FR3.2	M18, M11
FR3.3	M17, M11
NFR3.1	M16, M17, M18, M19, M20
NFR3.2	M16, M17, M18, M19, M20, M10, M11
NFR3.3	M16, M17, M18, M19, M20, M22, M23, M24

Table 1: Trace Between Requirements and Modules

AC	**Modules**
AC1	M1, M23
AC2	M5, M10, M1
AC3	M4, M8, M14, M9, M6
AC4	M5, M10, M12
AC5	M3, M7, M13
AC6	M16, M11
AC7	M17, M11
AC8	M18, M11
AC9	M19, M10
AC10	M20, M11
AC11	M2, M3, M5
AC12	M12, M13, M15
AC13	M13, M7
AC14	M11, M16, M17, M18, M19, M20
AC15	M9, M14, M15, M8

Table 2: Trace Between Anticipated Changes and Modules

9 Use Hierarchy Between Modules

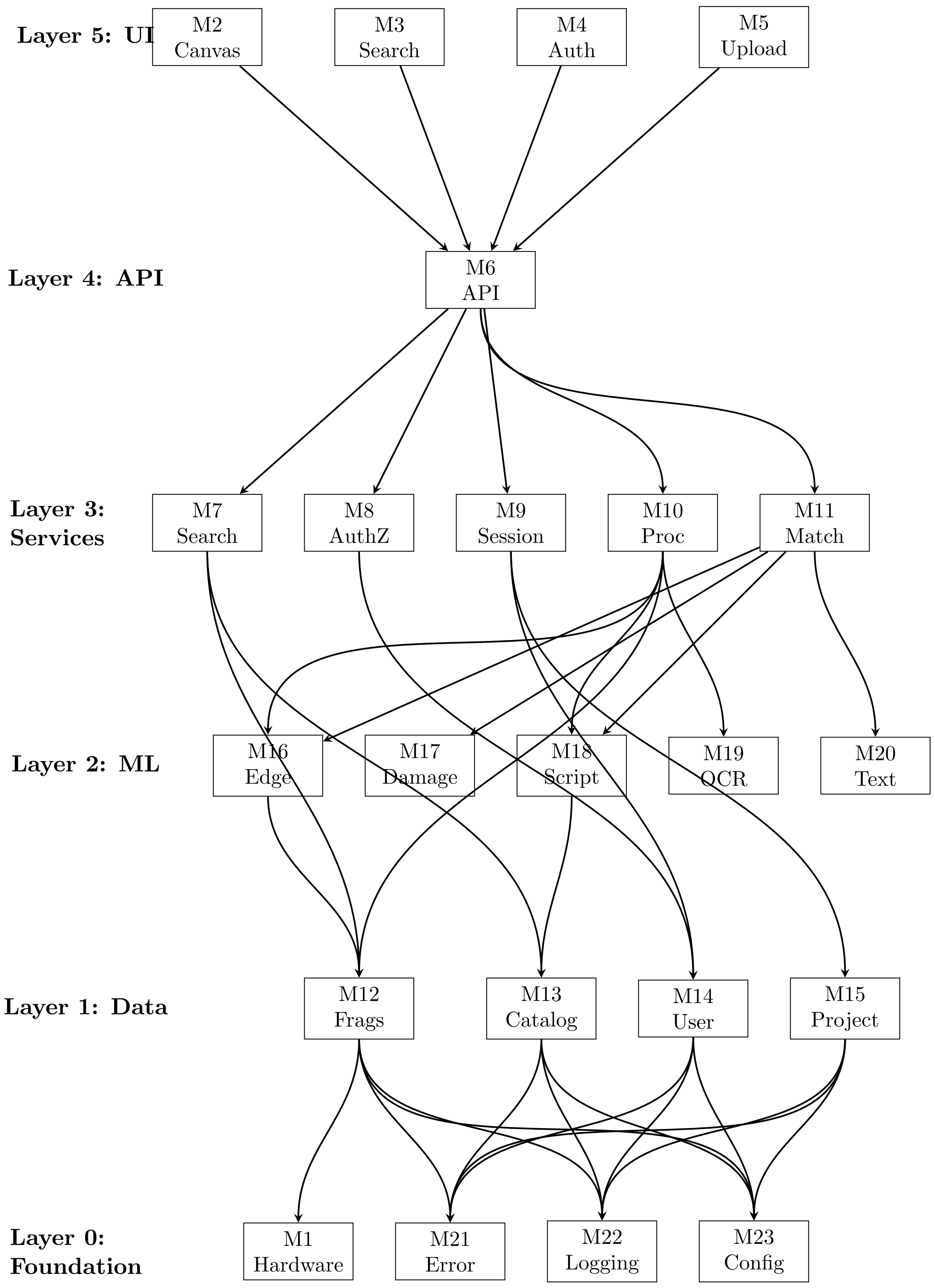

Figure 1: Use hierarchy among modules (simplified view showing key dependencies)

9.1 Uses Hierarchy Description

The uses hierarchy follows a strict layered architecture where modules at each level depend only on services provided by lower levels. The diagram shows direct dependencies between modules, with arrows indicating the uses relationship.

9.1.1 Layer 5: User Interface

All UI modules communicate exclusively through the API layer:

- **Canvas (M2), Search (M3), Auth (M4), Upload (M5)** all directly use **API (M6)**
- UI modules have no direct access to service, data, or ML layers, ensuring clean separation between presentation and business logic

9.1.2 Layer 4: API Gateway

API (M6) serves as the central gateway and coordinates all service layer modules:

- Directly uses all five service modules: **SearchSvc (M7)**, **AuthZ (M8)**, **Session (M9)**, **ProcOrch (M10)**, and **MatchOrch (M11)**
- Provides unified RESTful interface to all backend services for the UI layer
- Handles request routing, validation, response formatting, and rate limiting
- Acts as the single entry point for all business logic operations

9.1.3 Layer 3: Service Orchestration

Service modules implement business logic and coordinate lower-level modules:

Data Access Services:

- **SearchSvc (M7)** uses **FragmentStore (M12)** and **Catalog (M13)** to perform search and filtering operations across fragments
- **AuthZ (M8)** uses **User (M14)** for credential verification and role-based authorization
- **Session (M9)** uses **User (M14)** and **Project (M15)** to manage user sessions and collaborative workspace state

Orchestration Services:

- **ProcOrch (M10)** orchestrates image processing pipelines by using:
  - **FragmentStore (M12)** for accessing fragment images
  - **EdgeMatch (M16)** for edge detection during preprocessing
  - **ScriptClass (M18)** for initial script classification
  - **OCR (M19)** for text extraction from fragment images
- **MatchOrch (M11)** coordinates fragment matching by using:
  - **EdgeMatch (M16)** for edge profile comparison
  - **Damage (M17)** for damage pattern analysis
  - **ScriptClass (M18)** for script similarity comparison
  - **TextSim (M20)** for comparing extracted text between fragments

Note that ProcOrch focuses on preprocessing and feature extraction, while MatchOrch focuses on comparing fragments and ranking matches.

9.1.4 Layer 2: Machine Learning

ML modules provide specialized analysis capabilities:

- **EdgeMatch (M16)** uses **FragmentStore (M12)** to access fragment images for edge detection and comparison
- **ScriptClass (M18)** uses both **Catalog (M13)** for script taxonomy metadata and **FragmentStore (M12)** for fragment images
- Other ML modules (**Damage (M17), OCR (M19), TextSim (M20)**) also use **FragmentStore (M12)** for data access (connections shown representatively through EdgeMatch to reduce visual complexity)
- ML modules are always coordinated by service orchestrators (ProcOrch or MatchOrch), never invoked directly by the API

9.1.5 Layer 1: Data Storage

Data modules encapsulate all persistence operations and provide CRUD interfaces:

- **FragmentStore (M12)** uses:
  - **Hardware Hiding (M1)** for file system and storage operations
  - **Error Handling (M21)** for exception management

  - **Logging (M22)** for audit trails of fragment operations
  - **Config (M23)** for storage configuration parameters

- **Catalog (M13)** uses **Error Handling (M21)**, **Logging (M22)**, and **Config (M23)** for managing controlled vocabularies and metadata schemas
- **User (M14)** uses **Error Handling (M21)**, **Logging (M22)**, and **Config (M23)** for user profile management and authentication data
- **Project (M15)** uses **Error Handling (M21)**, **Logging (M22)**, and **Config (M23)** for collaborative workspace data

All data modules depend heavily on foundation modules to ensure reliable, configurable, and auditable data operations.

9.1.6 Layer 0: Foundation

Foundation modules provide cross-cutting system services with no dependencies on other system modules:

- **Hardware Hiding (M1)** abstracts operating system and file system operations, used by FragmentStore for persistent storage
- **Error Handling (M21)** provides exception management and error reporting, used by all data modules and propagated throughout upper layers
- **Logging (M22)** records system events, user actions, and audit trails, used extensively by data and service layers
- **Config (M23)** manages application configuration from files and environment variables, used by all data modules and some services

9.1.7 Key Architectural Properties

The uses hierarchy exhibits several important design properties:

1. **Strict Layering:** All dependencies flow downward through the layer hierarchy. Modules only use modules in lower layers, never peers or higher layers, ensuring clear separation of concerns.
2. **API Gateway Pattern:** The API module (M6) serves as the single entry point for all UI modules, providing a unified interface to backend services and preventing direct UI-to-service coupling.
3. **Orchestration Pattern:** Complex workflows involving multiple ML modules are coordinated by dedicated orchestrator services (ProcOrch and MatchOrch), which hide the complexity of multi-model coordination from the API layer.

4. **Data Encapsulation:** All persistent data access flows through dedicated data modules (M12-M15), preventing direct database access from service or ML layers and enabling consistent data validation and access control.

5. **Foundation Ubiquity:** Utility modules (Error, Logging, Config) are used throughout the system, particularly by the data layer. The diagram shows representative uses of these modules by all data modules, though they are actually used more broadly throughout the system.

6. **DAG Structure:** The dependency graph forms a directed acyclic graph with no circular dependencies, which enables:

    - Independent development and testing of each layer
    - Incremental system integration from bottom to top
    - Module substitution without affecting higher layers
    - Clear understanding of system dependencies and impact analysis

7. **Testable Subsets:** Each layer represents a testable subset of the system. Layer 0 can be tested independently, Layer 1 requires only Layer 0, and so on, supporting systematic integration testing.

8. **Separation of Concerns:** The architecture clearly separates presentation (Layer 5), API interface (Layer 4), business logic (Layer 3), specialized algorithms (Layer 2), persistence (Layer 1), and system utilities (Layer 0).

This hierarchical organization supports maintainability, testability, and evolution of the system over time.

10 User Interfaces

The Sanskrit Manuscript Reconstruction Platform provides a web-based interface designed for Buddhist Studies scholars to interact with manuscript fragments. The interface is built using React with TypeScript and styled using CSS to ensure consistency and responsiveness.

10.1 Primary Interface Components

10.1.1 Authentication Interface

- **Login Screen:** Secure credential entry with username and password fields
- **Session Management:** Token-based authentication with automatic session persistence
- **Error Handling:** Clear error messages for invalid credentials ("Invalid username or password")

10.1.2 Fragment Upload and Management

- **Upload Interface:** Drag-and-drop or file browser support for image uploads (JPG, PNG, TIFF formats, max 50 MB)
- **Preprocessing Visualization:** Real-time display of orientation correction with confidence scores
- **Fragment Browser:** Thumbnail grid view of uploaded fragments with metadata overlay

10.1.3 Interactive Canvas Workspace

- **Drag-and-Drop Canvas:** Visual workspace for arranging and comparing fragments
- **Fragment Manipulation:** Rotation, zoom, and positioning controls with snap-to-edge functionality
- **Session Persistence:** Save and restore workspace arrangements
- **Response Time Target:** UI drag/drop feedback latency under 200 ms

10.1.4 Search and Filter Interface

- **Search Bar:** Fragment ID search with autocomplete
- **Metadata Filters:** Multi-criteria filtering (script type, line count, fragment size)
- **Results Display:** Grid or list view with sortable columns
- **Export Options:** Download filtered results as CSV or PDF

10.1.5 Matching Suggestions Display

- **Ranked Match List:** Top-ranked fragment matches with similarity scores and confidence levels
- **Visual Comparison:** Side-by-side or overlay view for comparing fragments
- **Trust Signals:** Confidence scores, edge pattern overlays, and match explanations

10.1.6 Transcription Assistance

- **OCR Output Display:** Extracted text with character-level confidence indicators
- **Script Classification:** Display of identified script type (e.g., Gupta, Tibetan, Chinese) with confidence $\geq 70\%$
- **Manual Correction:** Editable text fields for scholarly review and correction

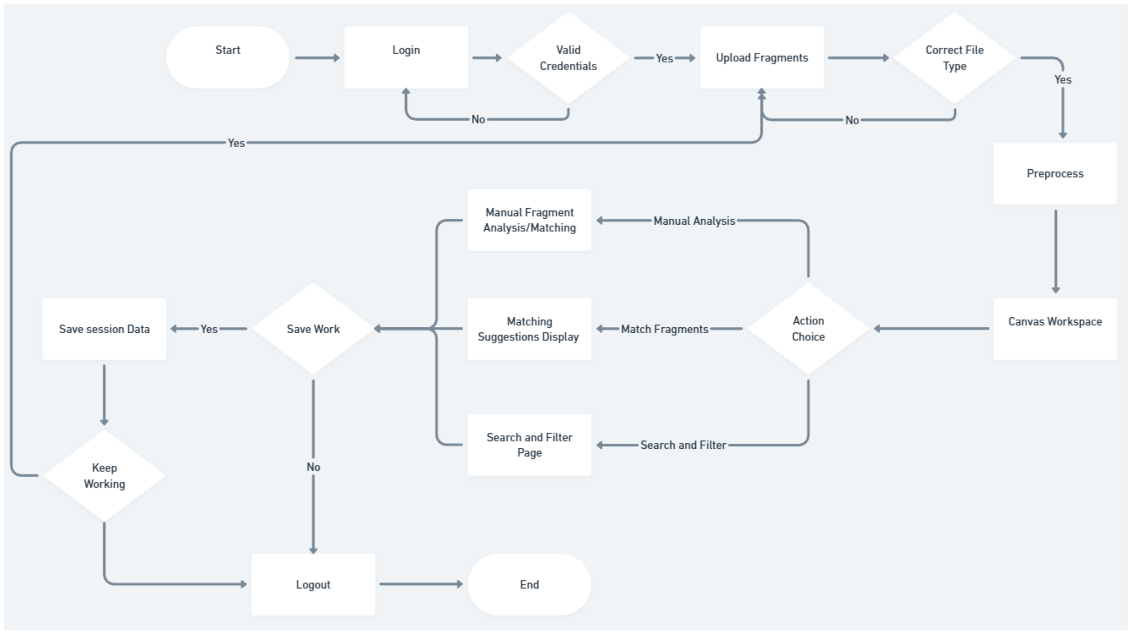

Figure 2: User Interface Flow Diagram showing the complete interaction workflow from login to fragment analysis

11 Design of Communication Protocols

The Sanskrit Manuscript Reconstruction Platform uses a client-server architecture with RESTful API communication between the frontend and backend services. This section describes the communication protocols and data exchange formats.

11.1 API Architecture

11.1.1 Backend API Layer

The backend API is implemented using Flask, FastAPI, or Django (final selection to be determined during implementation) and provides versioned RESTful endpoints for all system operations.

API Design Principles:

- **RESTful Convention:** Resource-based URLs with standard HTTP methods (GET, POST, PUT, DELETE)
- **Versioning:** All endpoints include version prefix (e.g., `/api/v1/`)
- **Status Codes:** Proper HTTP status codes (200 OK, 201 Created, 400 Bad Request, 401 Unauthorized, 403 Forbidden, 404 Not Found, 500 Internal Server Error)

- **JSON Format:** All request and response bodies use JSON
- **Schema Validation:** Request/response schemas defined using Pydantic models (backend) and TypeScript interfaces (frontend)

11.2 Core API Endpoints

11.2.1 Authentication Endpoints

- `POST /api/v1/auth/login` - User authentication, returns session token
- `POST /api/v1/auth/logout` - Session termination
- `GET /api/v1/auth/verify` - Token validation

11.2.2 Fragment Management Endpoints

- `POST /api/v1/fragments/upload` - Upload fragment image (multipart/form-data)
- `GET /api/v1/fragments/{id}` - Retrieve fragment metadata and image URL
- `GET /api/v1/fragments` - List/search fragments with query parameters
- `DELETE /api/v1/fragments/{id}` - Delete fragment (authorized users only)

11.2.3 Preprocessing Endpoints

- `POST /api/v1/preprocess/orientation` - Trigger orientation correction
- `GET /api/v1/preprocess/status/{job_id}` - Check preprocessing job status

11.2.4 Matching and Similarity Endpoints

- `POST /api/v1/matching/suggest` - Generate match suggestions for fragment
- `GET /api/v1/matching/results/{fragment_id}` - Retrieve ranked match list

11.2.5 OCR and Script Identification Endpoints

- `POST /api/v1/ocr/transcribe` - Extract text from fragment image
- `POST /api/v1/classification/script` - Identify script type
- `GET /api/v1/ocr/results/{fragment_id}` - Retrieve OCR output

11.2.6 Session and Workspace Endpoints

- `POST /api/v1/sessions` - Create new canvas workspace session
- `PUT /api/v1/sessions/{id}` - Update/save workspace state
- `GET /api/v1/sessions/{id}` - Restore workspace session
- `DELETE /api/v1/sessions/{id}` - Delete session

11.3 Data Exchange Formats

11.3.1 Request Schema Example (Fragment Upload)

```
POST /api/v1/fragments/upload
Content-Type: multipart/form-data

{
    "file": <binary image data>,
    "metadata": {
        "collection": "British Library",
        "shelfmark": "BL-12345",
        "provenance": "Gandhara",
        "notes": "Optional user notes"
    }
}
```

11.3.2 Response Schema Example (Match Suggestions)

```
GET /api/v1/matching/results/{fragment_id}
Response: 200 OK

{
    "fragment_id": "BL-12345",
    "matches": [
        {
            "match_id": "BL-67890",
            "similarity_score": 0.87,
            "confidence": 0.92,
            "match_features": ["edge_pattern", "damage_signature"],
            "thumbnail_url": "/api/v1/fragments/BL-67890/thumbnail"
        },
        ...
    ],
    "timestamp": "2025-11-11T14:30:00Z"
}
```

11.4 Inter-Service Communication

11.4.1 Backend to ML/AI Cluster

The backend orchestration service communicates with AI/ML components using internal API calls or message queues:

- **OCR Service:** Asynchronous job queue for text extraction
- **Matching Service:** Real-time or batch processing for similarity computation
- **Script Classification Service:** Synchronous inference requests

11.4.2 Database Communication

- **ORM Layer:** SQLAlchemy for database abstraction (PostgreSQL or SQLite)
- **Query Optimization:** Indexed searches for metadata filtering
- **Connection Pooling:** Maintained for concurrent user support (target: 15+ concurrent users)

11.5 Security and Data Integrity

- **Authentication:** Token-based authentication (JWT or similar)
- **Authorization:** Role-based access control for sensitive operations
- **Input Validation:** Server-side validation of all API inputs
- **SQL Injection Prevention:** Parameterized queries via ORM
- **API Rate Limiting:** Throttling to prevent abuse
- **HTTPS:** Encrypted communication in production deployment

12 Timeline

The development of the Sanskrit Manuscript Reconstruction Platform follows a phased approach, with module implementation organized by architectural layer. The timeline ensures that foundational components are established before dependent modules are developed, following the uses hierarchy described in Section 9.

12.1 Phase 1: Foundation and Infrastructure (Weeks 1-3)

12.1.1 Week 1: Project Setup and Foundation Modules

- **Setup Development Environment**
  - Configure MongoDB, Node.js/Express backend, and React frontend repositories
  - Set up version control, CI/CD pipelines, and development/staging environments
  - *Responsible: Full Team*
- **Implement Foundation Modules (Layer 0)**
  - M21: Error Handling Module - Define error types, implement centralized exception handling
  - M22: Logging Module - Configure structured logging framework (Winston/Bunyan)
  - M23: Configuration Management - Implement environment-based configuration loading
  - *Responsible: Yousef Shahin, Aswin Kuganesan*

12.1.2 Week 2-3: Data Layer Implementation

- **Hardware-Hiding Module**
  - M1: Implement file storage abstraction (local/cloud storage interface)
  - *Responsible: Omar El Aref*
- **Data Modules (Layer 1)**
  - M14: User database schema, authentication data models
  - M12: Fragment storage schema, image metadata, CRUD operations
  - M13: Catalog metadata structure, controlled vocabularies
  - M15: Project workspace schema, collaborative annotations
  - *Responsible: Umar Khan, Dylan Garner*

12.2 Phase 2: Service Layer and API (Weeks 4-6)

12.2.1 Week 4: Authentication and Session Management

- **Authentication Services (Layer 3)**
  - M8: JWT-based authentication, role-based access control
  - M9: Session state management, workspace persistence
  - *Responsible: Yousef Shahin, Aswin Kuganesan*

- **API Gateway (Layer 4)**
  - M6: Express.js RESTful API setup, authentication endpoints
  - *Responsible: Omar El Aref*

12.2.2 Week 5-6: Core Service Modules

- **Service Orchestration (Layer 3)**
  - M7: Fragment search and filtering logic, query optimization
  - M10: Image preprocessing pipeline orchestration, job queue management
  - M11: ML model result aggregation, confidence scoring (initial implementation without ML models)
  - *Responsible: Umar Khan, Dylan Garner*
- **API Endpoint Completion**
  - Complete all RESTful endpoints for fragment management, search, sessions, and preprocessing
  - *Responsible: Omar El Aref, Yousef Shahin*

12.3 Phase 3: Machine Learning Integration (Weeks 7-10)

12.3.1 Week 7-8: ML Infrastructure and Basic Models

- **ML Service Setup**
  - Python ML service infrastructure, REST API for ML models
  - Model serving framework (Flask/FastAPI for Python services)
  - *Responsible: Aswin Kuganesan, Umar Khan*
- **Initial ML Modules (Layer 2)**
  - M16: Edge detection and matching algorithm implementation
  - M18: Script classification model training and deployment
  - *Responsible: Dylan Garner, Yousef Shahin*

12.3.2 Week 9-10: Advanced ML Models

- **ML Module Completion (Layer 2)**
  - M17: Damage pattern analysis and similarity scoring

  - M19: OCR model integration (using existing libraries like Tesseract/EasyOCR, fine-tuned for ancient scripts)
  - M20: Text similarity comparison using embeddings
  - *Responsible: Omar El Aref, Aswin Kuganesan*

- **Orchestrator Integration**
  - Connect ML modules to M10 and M11
  - Implement weighted scoring and confidence calculation
  - *Responsible: Umar Khan, Dylan Garner*

12.4 Phase 4: User Interface Development (Weeks 8-11)

Note: UI development runs partially in parallel with ML integration

12.4.1 Week 8-9: Core UI Components

- **Authentication UI (Layer 5)**
  - M4: Login/logout interface, user registration, session display
  - *Responsible: Yousef Shahin*
- **Fragment Management UI**
  - M5: File upload interface with drag-and-drop, metadata forms
  - M3: Search bar, filtering interface, results display
  - *Responsible: Omar El Aref, Aswin Kuganesan*

12.4.2 Week 10-11: Interactive Canvas

- **Canvas Workspace (Layer 5)**
  - M2: Drag-and-drop canvas for fragment arrangement
  - Real-time match visualization, collaborative workspace features
  - Integration with M11 for displaying suggestions
  - *Responsible: Umar Khan, Dylan Garner*

12.5 Phase 5: Testing and Refinement (Weeks 12-14)

12.5.1 Week 12: Integration Testing

- **Test Infrastructure**
  - Mock implementations for all services and ML components
  - *Responsible: Full Team*
- **Layer-by-Layer Integration Testing**
  - Test data layer with foundation modules
  - Test service layer with data layer
  - Test ML integration with orchestrators
  - Test UI with complete backend
  - *Responsible: Full Team*

12.5.2 Week 13: Performance Optimization and ML Tuning

- Database query optimization, API response time tuning
- ML model fine-tuning based on test fragment dataset
- Concurrent user testing (target: 15+ users)
- *Responsible: Yousef Shahin, Aswin Kuganesan, Omar El Aref*

12.5.3 Week 14: User Acceptance Testing and Documentation

- User testing with domain experts (Buddhist Studies scholars)
- Bug fixes and usability improvements
- Complete API documentation
- User manual and deployment documentation
- *Responsible: Full Team*

12.6 Phase 6: Deployment and Monitoring (Week 15)

- Production environment setup and deployment
- Monitoring and logging infrastructure configuration
- Performance monitoring and error tracking setup

- Handoff and training for stakeholders
- *Responsible: Full Team*

Note: For detailed task assignments, sprint planning, and issue tracking, refer to the project's GitHub repository at: `https://github.com/DylanG5/sanskrit-cipher`

References

Team Sanskrit Ciphers. Software requirements specification (srs). https://github.com/DylanG5/sanskrit-ciphers-requirements/blob/9938018b682709551d0a6248a0a9f6e5794112ee/index.pdf, 2025. Specifies functional and non-functional requirements for the system.

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

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.