Final Exam CSC-580 Winter 2025

Name: Ehtasham Nasir, James Kessler, Philip Haapala, and John Tempey

1. Components: (15 pts)

a. Define one (1) component: 1) the component's purpose is and how it accomplishes its goals) and 2) draw the component, including the classes that would be part of this component (be sure to identify the methods that would serve as the interface to this component also).

Component:

Intelligent Document Processor (IDP)

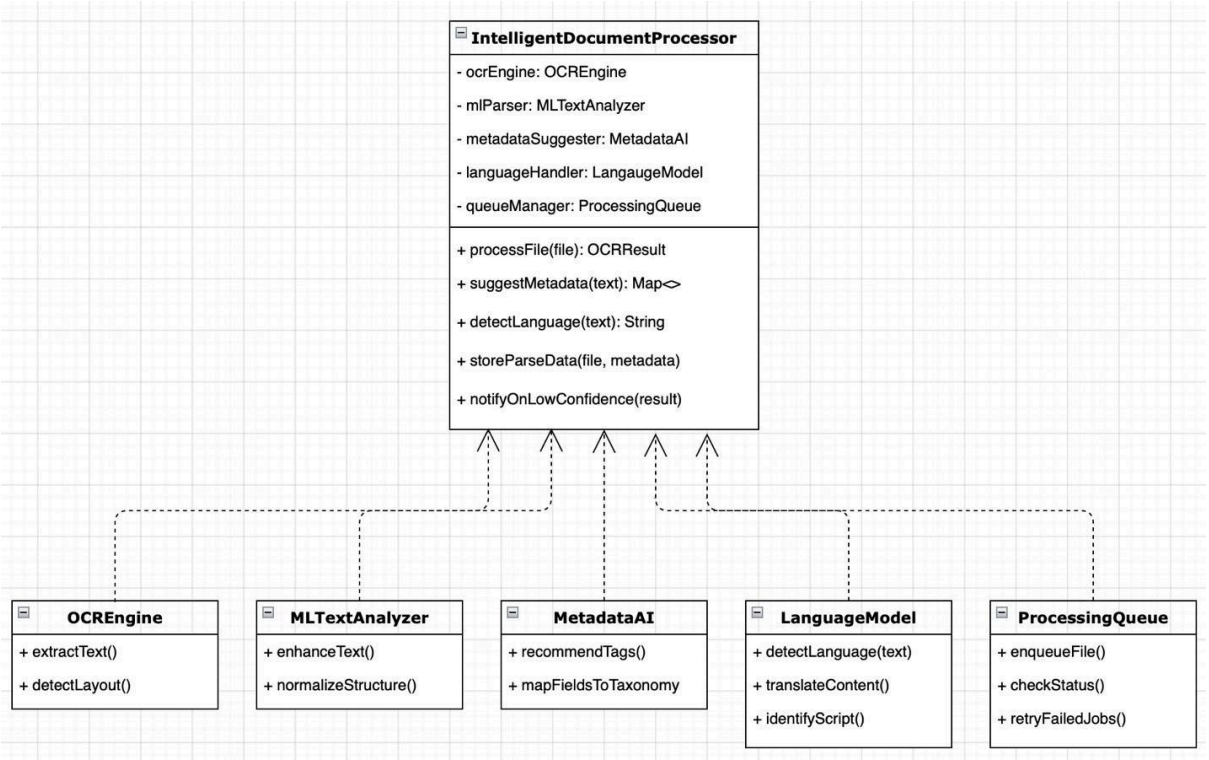
Purpose & Function:

The purpose of the Intelligent Document Processor (IDP) is to take scanned documents and uploaded media (e.g., PDFs, images, multilingual sources, etc.) and transform them into "smart" assets. In other words, these documents become not only fully searchable but also semantically tagged for efficient classification.

The IDP component automates the extraction and interpretation of document content using OCR and machine learning. Its responsibilities are threefold: First, it converts image-based documents to searchable text. Second, it enhances the extracted data ML-driven analysis. Third, it proposes relevant metadata to support accurate classification and retrieval within the EDMS.

This component is a key addition because it automates a traditionally manual process – text extraction, classification, and metadata tagging. It integrates directly with the Upload Module, the Centralized Repository, and the Access Control Layer to ensure that enriched content maintains document integrity and respects security policies.

Overall, the IDP aims to accomplish the following: (1) use OCR to extract text and key fields from uploads. (2) handle poor-quality scans or complex layouts via ML enhancements. (3) use document content and structure to suggest or auto-populate metadata tags. Fourth, once results are generated, send results into the version control and repository system. Fifth, if OCR results return "low confidence" flag for manual review.



IntelligentDocumentProcesser (Main Component)

Role

- Orchestrates document intelligence tasks.

Responsibilities

- Handles the OCR-to-metadata enrichment pipeline
- Utilizes five helper classes through internal logic
- Makes public methods available for external modules to start processing

Attributes (Private)

- ocrEngine: reference to helper module
- mlParser: reference to helper module
- metadataSuggester: reference to helper module
- queueManager: reference to helper module

Methods (Public)

- processFile(file) – kicks off the OCR and enrichment process
- suggestMetadata(text) – recommends tag suggestions from parsed content
- detectLanguage(text) – utilizing the language model, detects document language
- storeParsedData(file, metadata) – sends results to repository

- notifyOnLowConfidence(result) – sends out alerts on uncertain OCR outcomes

OCREngine

Role

- Pulls raw text and layout from content (e.g. image or PDF).

Methods

- extractText() – principal OCR text extraction
- detectLayout() – recognizes structural elements (e.g. tables, headers, etc.)

MLTextAnalyzer

Role

- Uses machine learning to improve raw OCR output

Methods

- enhanceText() – filters out bad text (e.g. corrects errors)
- normalizeStructure() – either adds or fixes structure of document

MetadataAI

Role

- Organizes data appropriately and recommends semantic tags

Methods

- recommendTags() – proposes metadata classification (e.g., category, author, etc.)
- mapFieldsToTaxonomy() – organizes content based on user-defined tag structures

LanguageModel

Role

- Converts multilingual content and language inference

Methods

- detectLanguage(text) – recognizes document language
- translateContent() – upon request, converts content to a target language
- identifyScript() – determines script (e.g., Cyrillic, Latin)

ProcessingQueue

Role

- Asynchronously manages job execution

Methods

- enqueueFile() – pushes document to processing queue
- checkStatus() – tracks job state (e.g., queued, processing, etc.)
- retryFailedJobs() – reruns failed jobs due to issues

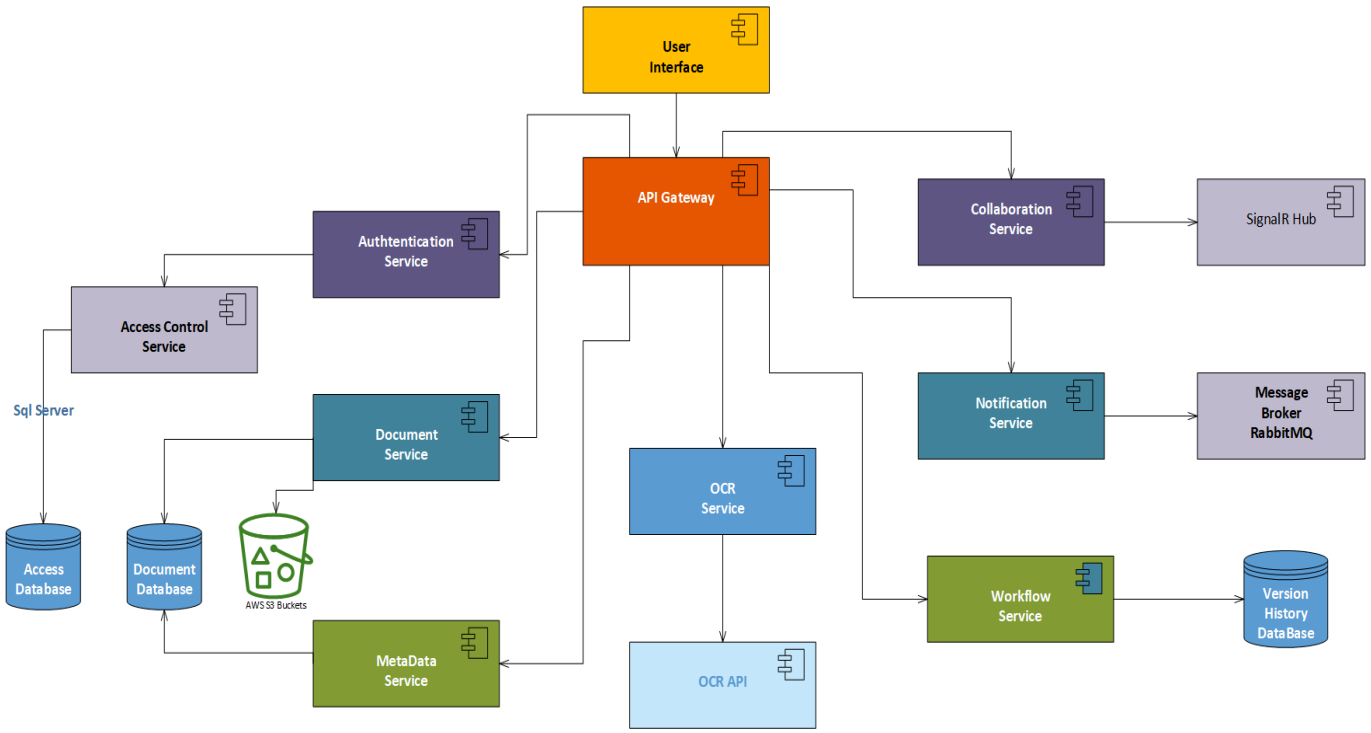
Question 2: Architecture (25 pts.)

a. Choose one (1) architectural style that you feel would be suitable for this application and draw it out.

Chosen Architectural Style: Microservices Architecture

Rationale for Selection: The EDMS requires scalability, modularity, and the ability to integrate with external cloud services (e.g., AWS, Amazon S3) and intelligent OCR technologies. Microservices architecture breaks the system into independent services (e.g., Document Service, OCR Service) that can be developed, deployed, and scaled individually, meeting the EDMS's need for handling diverse functionalities like document storage, collaboration, and access control while ensuring seamless integration with external systems.

Component Diagram for Microservices Architecture



The Component Diagram shows the EDMS system as a collection of microservices, each interacting through an API Gateway. The User Interface (web/mobile app) connects to the API Gateway, which routes requests to microservices like Document Service, OCR Service, and Collaboration Service.Each microservice interacts with its own database or external service (e.g., Document Service with Amazon S3, Collaboration Service with SignalR for real-time features).

b. Fill out the following table for your architecture to determine if the one you selected would be a good choice.

Sort	Attributes	Importance Factor (1-3 or 1-5) 1 being MOST important	Why is this attribute important?	Met by this Architecture? (Y/N)	How does the architecture meet it?
1	Security	1	Custom access control and encryption are critical for document confidentiality and compliance.	Y	The Access Control Service enforces role-based policies, and each microservice can implement its own security measures (e.g., encryption).
2	Reliability	1	The system must remain operational, ensuring users can access documents even if some components fail.	Y	Microservices are loosely coupled; failure in one service (e.g., OCR) doesn't affect others (e.g., Document Service), ensuring system reliability.
3	Availability	1	The EDMS must be highly available to support users across locations, including remote workers.	Y	Microservices can be deployed across multiple servers or cloud regions, ensuring high availability through redundancy.
4	Performance	2	Real-time co-authoring and fast document retrieval are essential for user experience.	Y	Independent services can be optimized (e.g., caching in Document Service), and load balancing ensures efficient performance.
5	Usability	2	The EDMS must be easy for users to interact with for tasks like uploading and sharing documents.	Y	Microservices enable a modular UI (e.g., separate Document Service, Collaboration Service), allowing intuitive, role-specific interfaces.
6	Supportability	2	The architecture should be easy to monitor and troubleshoot for IT administrators.	Y	Microservices can be monitored independently (e.g., using tools like Datadog), and logs are isolated per service, simplifying support.
7	Maintainability	2	The system should be easy to update as new features (e.g., OCR improvements) are added.	Y	Each microservice has its own codebase, allowing updates (e.g., adding OCR features) without affecting the entire system.

8	Safety	3	The system must prevent unsafe operations (e.g., data corruption during concurrent edits).	Y	Version control and conflict resolution (e.g., in the Workflow Service) ensure safe concurrent edits and data integrity.

c. Create a narrative response to answer in depth: 1) how the architecture was selected, 2) why it was selected (what purpose it should fulfill), 3) how it strengthens the enterprise environment (functionality it adds), and 4) which alternatives were evaluated.

Narrative Response:

How the architecture was selected:
The Microservices architecture was selected after analyzing the ED Quantum Management System (EDMS) requirements from the initial requirements and the expanded features, such as cloud integration, OCR, and real-time collaboration. The initial requirements emphasized modularity (e.g., separate concerns like document storage and access control) and scalability (e.g., supporting employees across locations). The new requirements added complexity with external integrations (e.g., Amazon S3, OCR APIs) and real-time features, which demand a flexible and scalable architecture. Microservices were chosen by evaluating architectural styles like Client-Server, Layered, and Event-Bus, where Microservices best met the needs for modularity, scalability, and integration.

Why it was selected (what purpose it should fulfill):
Microservices architecture was selected to fulfill the purpose of building a scalable, modular, and resilient EDMS that can handle diverse functionalities while integrating with external systems. It ensures that each component (e.g., OCR, collaboration, access control) operates independently, allowing targeted scaling and updates. For example, the Document Service can scale to handle increased uploads without affecting the OCR Service. Microservices also support fault tolerance—if the OCR Service fails, users can still upload or share documents. Additionally, this architecture facilitates seamless integration with cloud storage and OCR APIs through standardized communication protocols like REST.

How it strengthens the enterprise environment (functionality it adds):
Microservices strengthen the enterprise environment by enabling scalability, flexibility, and enhanced collaboration. It allows the EDMS to scale specific components (e.g., Collaboration Service for real-time co-authoring) to meet demand, ensuring consistent performance. The modularity simplifies maintenance and feature updates, such as adding new OCR capabilities without redeploying the entire system. It also enhances security by isolating sensitive operations (e.g., Access Control Service enforces custom permissions). Furthermore, the architecture supports collaboration goals by enabling real-time features through dedicated services, improving workflow efficiency for employees, managers, and external collaborators.

Which alternatives were evaluated:
Alternatives evaluated include Client-Server, Layered, and Event-Bus architectures. Client-Server was considered but rejected because it lacks the access detail and scalability needed for independent components like OCR and collaboration. The Layered architecture was evaluated but deemed unsuitable due to its monolithic nature, which hinders independent scaling and integration with external services. Event-Bus was a strong

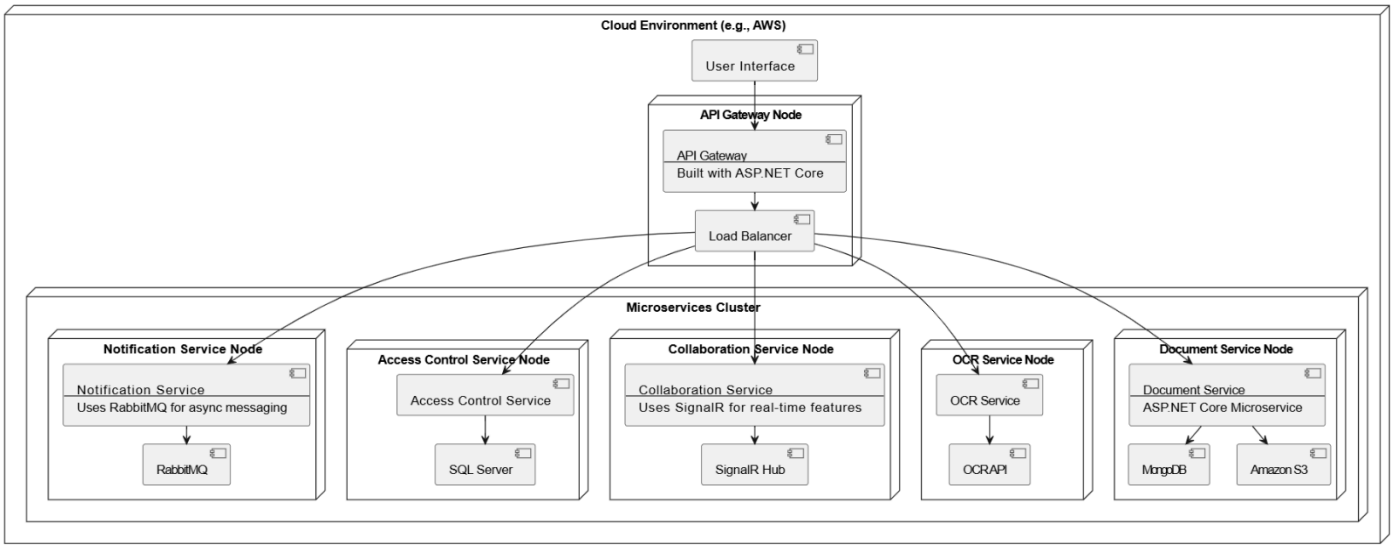
contender due to its support for asynchronous communication (useful for notifications), but it was less ideal for the EDMS's need for modularity and direct API-based integrations with cloud services. Microservices were chosen for their balance of modularity, scalability, and integration capabilities.

Question 3: Framework (25 pts.)

a. Choose one (1) Enterprise Framework style that you feel would be suitable for this application and draw it out.

Chosen Enterprise Framework: ASP.NET Core

Rationale for Selection: ASP.NET Core is a high-performance, cross-platform framework ideal for building microservices in the .NET ecosystem. It supports rapid development with features like dependency injection, minimal APIs, and integration with cloud services (e.g., AWS SDK for .NET for Amazon S3). ASP.NET Core also provides robust security (ASP.NET Core Identity), real-time communication (SignalR), and database access (Entity Framework Core), aligning with the EDMS's requirements for scalability, security, and collaboration.



The Deployment Diagram shows the EDMS deployed in a cloud environment, with nodes representing the API Gateway and microservices. Each microservice (e.g., Document Service, Collaboration Service) runs on its own node, implemented using ASP.NET Core. The API Gateway routes requests to microservices via a Load Balancer. External integrations (e.g., Amazon S3, OCR API) and internal components (e.g., SignalR, RabbitMQ) are shown with their connections. Notes clarify the use of ASP.NET Core and specific .NET features like SignalR.

c. Fill out the following table for your framework selection to determine if the one you selected would be a good choice.

Sort	Attribute	Importance Factor (1-3 or 1-5) 1 being MOST important	Why is this attribute important?	Met by this Architecture? (Y/N)	How does the architecture meet it?
1	Security	1	Custom access control and encryption are critical for document confidentiality and compliance.	Y	ASP.NET Core Identity provides role-based access control and encryption for secure document handling.

2	Reliability	1	The system must remain operational, ensuring users can access documents consistently.	Y	ASP.NET Core MVC is enterprise-grade, with robust error handling and middleware ensuring reliability.
3	Availability	1	The EDMS must be highly available to support users across locations, including remote workers.	Y	ASP.NET Core MVC can be deployed to the cloud (e.g., AWS) with load balancing for high availability.
4	Performance	2	Real-time co-authoring and fast document retrieval are essential for user experience.	Y	ASP.NET Core MVC leverages .NET Core's high performance, and SignalR enables real-time features.
5	Usability	2	The EDMS must be easy for users to interact with for tasks like uploading and sharing documents.	Y	ASP.NET Core MVC provides a clear structure (Model-View-Controller), enabling intuitive UI design for users.
6	Supportability	2	The architecture should be easy to monitor and troubleshoot for IT administrators.	Y	ASP.NET Core MVC supports monitoring tools (e.g., Application Insights) for easy troubleshooting.
7	Maintainability	2	The system should be easy to update as new features (e.g., OCR improvements) are added.	Y	MVC's separation of concerns allows updates to specific components (e.g., Model for OCR) without affecting others.
8	Safety	3	The system must prevent unsafe operations (e.g., data corruption during concurrent edits).	Y	ASP.NET Core MVC uses version control (via Entity Framework Core) and concurrency controls to ensure safety.

c. Create a narrative response to answer in depth: 1) how the framework was selected, 2) why it was selected (what purpose it should fulfill), 3) how it strengthens the enterprise environment (functionality it adds), and 4) which alternatives were evaluated.

Narrative Response:

How the framework was selected:
ASP.NET Core was selected after evaluating the EDMS's technical requirements and the Microservices architecture chosen for the system. The initial requirements highlighted the need for features like dependency injection, middleware pipelines, and minimal APIs, which ASP.NET Core supports natively. The new requirements, such as cloud integration (e.g., Amazon S3) and real-time collaboration, necessitated a framework with strong integration and real-time capabilities. ASP.NET Core was chosen for its alignment with microservices, enterprise features (e.g., security, cloud integration), and familiarity to a .NET developer.

Why it was selected (what purpose it should fulfill):
ASP.NET Core was selected to fulfill the purpose of simplifying microservice development while providing robust enterprise features for the EDMS. It streamlines the creation of independent microservices (e.g., Document Service, OCR Service) with minimal APIs, built-in dependency injection, and cross-platform support. ASP.NET Core Identity ensures secure authentication and restricted access control, critical for document security and compliance. The framework also supports real-time communication via SignalR, enabling features like co-authoring and notifications, and integrates with cloud services like Amazon S3 using the AWS SDK for .NET.

How it strengthens the enterprise environment (functionality it adds):
ASP.NET Core strengthens the enterprise environment by accelerating development, enhancing security, and enabling scalability. It allows rapid development of microservices using minimal APIs and Visual Studio tools, ensuring the EDMS can be built efficiently. ASP.NET Core Identity provides robust authentication and authorization, protecting sensitive documents. Integration with cloud storage (e.g., Amazon S3) enhances accessibility and scalability, while SignalR supports real-time collaboration, improving workflow efficiency for employees, managers, and external collaborators.

Which alternatives were evaluated:
Alternatives evaluated include Laravel (PHP), Node.js frameworks, and .NET Framework. Laravel was considered but rejected because, despite its robust features, it lacks the high-performance capabilities and seamless enterprise-grade cloud integration of ASP.NET Core, which are critical for the EDMS's scalability and deployment needs. Node.js frameworks were assessed for their real-time capabilities (e.g., Socket.IO) but were dismissed because they lack the comprehensive enterprise features of ASP.NET Core, such as robust security and seamless cloud integration. The .NET Framework was also evaluated but ruled out as it is not cross-platform and misses modern features like minimal APIs and SignalR. ASP.NET Core was selected for its cross-platform support, enterprise-grade features (e.g., advanced security, cloud integration), modern capabilities (e.g., minimal APIs, SignalR), and familiarity to .NET developers.

Question 4: Design Patterns (15 pts.)

a. Choose at least four (4) patterns that would be suitable to your system and describe:
    i. How they were selected
    ii. How they will be useful
    iii. How they will be implemented
    iv. How it will (or will not) be reused throughout the enterprise

Chosen Design Patterns:

Chosen Design Pattern 1: Facade Pattern

Rationale for selection: The Facade design pattern fits well with our chosen Microservices Architecture and allows for the use of the API Gateway, as seen in the ASP.NET Core framework diagram, which plays a critical role in our system as a whole. The API Gateway provides a single, unified entry point for a defined group of microservices (i.e. Document Service, Notification Service, etc.). Instead of front-end clients (e.g. the user interface) calling

dozens of individual microservice APIs directly, they make calls to the API Gateway. The API Gateway then routes these calls to the appropriate downstream microservice(s).

How it will be useful: This design pattern simplifies client interaction/responsibility, allows for easy scalability through the decoupled structure of the system's various services, and centralizes cross-cutting concerns, such as authentication checks. The EDMS has multiple services (Document, OCR, Access Control, Collaboration, etc.). The UI shouldn't need to know the specific address and API details of every single one. The Gateway provides a stable interface. Furthermore, this design pattern hides the internal microservice structure and allows services to be refactored or replaced without impacting the client directly. Finally, it simplifies the system by handling authentication checks once at the edge, rather than duplicating logic in each service.

How to implement: First, we would create a dedicated ASP.NET Core application project specifically for the Gateway. Then, we could use ASP.NET Core's built-in routing capabilities to define endpoints that the client will call. Inside these endpoints, use HttpClientFactory, for example, to make HTTP calls to the internal microservice endpoints. Finally, we could implement authentication/authorization using ASP.NET Core middleware.

Potential for reuse: The Facade design pattern is a fundamental building block for almost any Microservices Architecture within an enterprise. While the API Gateway's specific routes and logic are application-specific, the pattern of having a managed entry point is universally applicable across different business domains implementing microservices. Therefore, this design pattern has a *high potential for reuse* throughout the enterprise.

Chosen Design Pattern 2: Repository Pattern

Rationale for selection: The EDMS has various data stores (SQL Server for Access Control, Document Database like MongoDB for Document Service, AWS S3 for file storage), which lends itself nicely to the use of the Repository design pattern. The Repository design pattern mediates between the domain (business logic) and data mapping layers (data access code). It provides an abstraction layer over the data store, making it look like an in-memory collection of domain objects. It encapsulates the logic required to access data sources (databases, cloud services).

How it will be useful: Similar to the API Gateway pattern, the Repository pattern also exhibits a decoupled structure as well as centralizing certain logical flows, such as data access. Additionally, the Repository pattern allows each service's business logic to interact with data consistently, regardless of the underlying storage mechanism. Finally this design pattern enables better testability because business logic can be unit tested by providing mock implementations of the repository interface, without needing a real database connection.

How to implement: To start, we would define an interface in our service's core/domain project. Next, we might create concrete implementation classes in an infrastructure project. This class would contain the actual code using the MongoDB Driver to interact with the database. Then, we could use ASP.NET Core's built-in Dependency Injection (DI) container to register the implementation. Finally, this would allow us to inject the interface into the constructors of services or controllers where data access is needed. The DI container provides the concrete instance at runtime.

Potential for reuse: The Repository design pattern is a standard pattern for data access abstraction in object-oriented design. Any application or service within the enterprise that interacts with a data store can benefit from this pattern. Therefore, the design pattern has a *high potential for reuse* throughout the enterprise.

Chosen Design Pattern 3: Strategy Pattern

Rationale for selection: The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm (back-end) vary independently from the clients (front-end) that use it. This pattern will allow us to define an interface representing an operation, and create multiple concrete classes implementing that interface, each representing a different "strategy".

How it will be useful: It will help address requirement 1 (Cloud Storage Flexibility) because the Strategy pattern would allow the Document Service to switch between S3, Google Drive, or OneDrive without changing its core workflow. The Strategy pattern also addresses requirements 5 & 6 because if, after first implementing a basic OCR technology, we want to later add a more advanced ML-based one, or need different OCR engines for different languages/document types, this pattern will allow the OCR Service to select the appropriate engine dynamically. Finally, this design pattern helps to future-proof our system because it gives us the freedom to be able to add new strategies (new cloud providers, new OCR engines) later with minimal changes to the core service logic.

How to implement: The first step would be to define an interface representing the strategy we would like to implement. After this, we would create concrete implementation classes for each strategy. Next, we would register all available strategies, and then implement a specific strategy based on configuration. The service using the strategy holds a reference to the interface and calls its methods, unaware of the concrete implementation being used.

Potential for reuse: This pattern is applicable anywhere in the enterprise where multiple ways exist to perform the same fundamental task. Therefore, it has a *high potential for reuse* throughout the enterprise.

Chosen Design Pattern 4: Abstract Factory pattern

Rationale for selection: The EDMS requires integration with multiple distinct cloud storage providers (Requirement #1: Amazon S3, Google Drive, Microsoft OneDrive) which represent different "platforms" for storage operations. Directly implementing logic for each provider within the core Document Service would lead to complex conditional statements and tight coupling. The Abstract Factory pattern is selected because it provides an interface for creating families of related storage objects without specifying their concrete classes, effectively encapsulating the platform dependencies.

How it will be useful: The Abstract Factory design pattern will allow the EDMS application to interchangeably switch between cloud storage providers with minimal code changes, primarily by changing the configuration that determines which concrete factory is instantiated. Additionally, it will ensure consistency because all the components used for interacting with a specific cloud provider will be from the same compatible family. Finally, similar to the Repository and API Gateway design patterns, the Abstract Factory design

pattern will decouple implementations–the Document Service code will depend only on the abstract factory and the abstract product interfaces, not on the concrete implementations for S3, Google Drive, etc.–and centralize the potentially complex logic of instantiating and configuring provider-specific objects within each concrete factory.

How to implement: First, we can define an abstract factory interface with methods to create each product, and abstract product interfaces. Then, we could create concrete factory classes for each provider. These implement the factory methods to return provider-specific concrete product instances. Next, we could create concrete product classes that implement the product interfaces using the provider's specific SDK. Within the Document Service, for instance, we could then use ASP.NET Core's Dependency Injection (DI). We would configure the DI container to register a specific concrete factory for the interface, based on application configuration. Finally, we would simply inject the interface into the Document Service. The Document Service would then call the factory methods to get instances of the required storage components and interact with them via their interfaces.

Potential for reuse: The Abstract Factory pattern is highly reusable across the enterprise. It is ideal for any application or system that needs to create families of related objects whose concrete types depend on configuration or the environment. Examples include supporting multiple database vendors, different UI toolkit themes, various reporting engines, or integrating with different external systems that require sets of related interaction components. Therefore, it has a *high potential for reuse* throughout the enterprise.

5. Testing: (20 pts.)

a. Describe in detail how the application will be tested – this will include:

i. Which testing scenarios will be utilized (regression and/or standard scenarios)

Beginning during development, developers will unit test their Feature or Component code at their own benches. Any interfaces or dependencies on other Features should be stubbed in the automated unit test framework if required.. Each time a Feature is added or changed, the unit test cases should be created or updated at the same time. All unit tests must pass by the time code is merged into the Main branch for compilation into an application. The goal of unit testing is automated regression testing of the individual Features without regard for interactions between Features.

Based on the Project timings, Features will be bundled into the complete Subsystems for verification testing by a dedicated test team. Each bulleted item in the final exam requirements document will be considered a Subsystem unless the requirement is non-functional.

The dedicated test team has the responsibility of doing gray or black box testing to try to exercise the Subsystem after Feature integration in both happy and sad path scenarios. Happy path testing will assume that the user or another Feature or

Subsystem completes all of the interactions with the system as expected by the requirements and design successfully. Sad path assumes one or more of the interactions with the user or inputs from other areas of the system do not happen as expected - testing to ensure that the system responds appropriately. Special attention should be paid to complex interactions between Features and Subsystems.

Once the dedicated test team has decided that the risk of defects has been reduced to a level that is acceptable to the management team, all outstanding show-stopper and high priority defects have been resolved and re-tested with passing results, and the program is near the launch phase of the current release, then User Acceptance Testing (UAT) will commence.

UAT attempts to determine the suitability of the application for use by end-users. It involves a complete or mostly completed application presented to a representative team of users or potential users of the application. The level of experience should encompass the complete spectrum of users - with special focus paid to the inexperienced and elderly population who did not grow up with smartphones. Instruction from the development and testing teams should be minimal - users should be free to explore the application and how to operate it. Minimal guidance in the form of common tasks or bundles of tasks are to be provided to the users by the test team. However, a line of reasoning or testing investigation may warrant additional testing based on previous test results - bundles are not set in stone. This is at the discretion of the test team.

User feedback generated during the UAT sessions should be defects in which the software violates a requirement, where a requirement is incomplete or incorrect to meet the user needs, or a non-functional requirement such as responsiveness or security has been violated. Defects should be reproduced, triaged, and handed to the development team lead for correction by the developers.

Since this application includes sensitive documents uploaded by users, security is an important consideration. During development, secure software development best practices should be followed so that as few defects as possible make into the app as is reasonably possible. Additionally, a qualified penetration test team should attempt to compromise customer data that is stored in the app, with any critical or high priority defects being addressed prior to application shipment or as soon as possible if the app is in customer hands.

Finally, performance testing should be performed as part of testing the application's server-side non-functional requirements. A performance test should be at the Application level, testing the proposed release. A load, simulating users performing various tasks that are supported by the requirements, will be applied to the application. Under load, there are specific time expectations from the frontend user perspective to be able to perform certain operations as well as backend application

expectations such as data consistency and API endpoint availability and response time.

A similar set of scenarios should also be developed for the mobile application and performed on a PC-based smartphone emulator and on popular smartphone hardware. Special attention should be paid to include popular smartphones that are nearing end of life and are therefore less capable.

Defects that make it past the development stage should be non-trivial in nature; that is: involve a sequence of steps that is long or complicated, require a certain very specific set of inputs, etc. Trivial defects making it past the development stage is cause for the team to reevaluate testing practices.

The decision to release the application shall be taken together with the development team and the test team, with final approval and responsibility resting with the management team.

ii. Testing Roles

- Developer
    - Writes software components, resolves defects in software based on direction from developer lead.
- Developer Lead
    - Sets developer team priorities and mentors team in developing defect free software
- QA Test Team Member
    - Tests software beyond the software unit or component level. Mentors development team members in unit testing.
- QA Lead
    - Works with the management team to lead the test team in mitigating risk posed by undiscovered defects. Leads test team to reduce defects by detection and resolution of defects with test and development teams.
- Management Team (QA Manager/Development Manager)
    - Ultimately responsible for the quality level of delivered software. Works with development and test teams to accomplish quality deliverables.

iii. Quality Assurance Vs. User Acceptance Scope

See i.

iv. Example Test Plan

9-4: EDMS (Electronic Document Management System)

Integration Testing Plan

Assumptions

1. **Stable Environment**: The test environment will mirror production, including microservices configurations (e.g., Document Service with S3 integration), data structures, and user roles, ensuring test results reflect live operations. To achieve this, the UAT and verification tests will be executed in a staging environment that replicates the production cloud infrastructure, including the API Gateway, document storage integrations, and user authentication protocols.
2. **Finalized Requirements**: All business requirements (e.g., document approval workflows, real-time collaboration) are signed off before testing begins, with any changes managed via a formal change control process to avoid delays.
3. **Test Data Preparedness**: Sanitized test data, including synthetic documents and user profiles (e.g., employees, managers), will be available, covering happy (e.g., successful uploads), sad (e.g., permission denials), and edge cases (e.g., large files). To ensure consistency with evolving system features and workflows, test data will be refreshed prior to each major testing cycle. This refresh includes: (1) uploading new sample documents and (2) resetting access permissions to simulate real-world scenarios.

.

Constraints

1. Limited Testing Window: integration testing must be completed within a 5-day window (05/31/2025 to 06/04/2025) due to project deadlines, with extensions only via change control approval.

2. Resource Availability: The QA Test Team will perform testing with feedback from the Development team, with potential conflicts in direction and strategy set by the QA Lead.

3. Scope Restriction: Functional requirements (happy and sad path, edge cases) are in scope. Non functional requirements (security, performance) are also in scope.

Team

See ii.

Example Test Scenarios Outline for Subset of Requirements

- Cloud integration for storing documents
    - Upload
        - Document to Cloud

                - By unauthenticated user
                - By authenticated user
                    - By authorized user
                    - By unauthorized user
        - Download
            - Document from Cloud
                - By unauthenticated user
                - By authenticated user
        - Synchronization
            - Metadata
                - Fill out blank metadata field
                - Update existing metadata field
                    - To same value
                    - To different value
                - Delete value from existing metadata field
                - Delete metadata field
- Granular access control
            - Give access
                - To single document
                    - By user
                    - By group for user role
                - To group of documents
                    - Each document specified individually
                    - Based on sufficient security clearance level
                    - Based on filter matching all allow metadata criteria
                    - Based on filter matching no deny metadata criteria
            - Deny access
                - To single document
                    - By user
                    - By group for user role
                - To group of documents
                    - Based on multiple matching deny metadata criteria
                    - Based on single matching deny metadata criteria
                    - Based on insufficient security clearance level

Example Test Cases

The following test cases will be used for UAT to ensure the EDMS meets business requirements and functions as expected for end-users:

Test Case ID	Test Case	Test Steps	Expected Result	Actual Result	Status (Pass/ Fail)	Comments
TC001	Document Upload and Storage	1) Employee logs in, 2) Uploads a PDF, 3) Confirms upload completion.	PDF is uploaded to S3, success message displayed within 5 sec.	Successful PDF upload and displayed in UISuccess message within 2.3 sec	Pass	No issues encountered
TC002	Document Sharing with Collaborator	1) Employee selects document, 2) Shares with external collaborator, 3) Sends invite.	Collaborator receives invite, accesses document.	Collaborator accessed shared document link in under 10 sec	Pass	Collaborator UI loaded slowly on mobile
TC003	Real-Time Collaboration	1) Two employees open a document, 2) Both edit simultaneously, 3) Save changes.	Changes sync in real-time (<1 sec), both see updates.	[TBD]	[TBD]	[TBD]
TC004	Document Approval Workflow	1) Employee submits document for approval, 2) Manager approves, 3) Check status.	Status updates to “Approved,” notification sent in <10 sec.	[TBD]	[TBD]	[TBD]
TC005	Access Control (AC) for Restricted Document	1) Employee tries to access restricted document, 2) System checks permissions.	Access denied, error message displayed.	Document opened without restriction	Fail	Critical bug - AC failed to enforce permissions; issue reported to dev team and logged in Jira
TC006	Authenticated but unauthorized user attempts upload of document	1) User logs in 2) User attempts to upload a document	1) Log in successful 2) User does not have option to upload document	[TBD]	[TBD]	[TBD]
TC007	User accesses list of all documents that their security clearance	1) User logs in 2) User goes to list of all documents visible to them	1) Log in successful 2) Documents user has sufficient security	[TBD]	[TBD]	[TBD]

	allows them to see		clearance to see are visible in list, documents for which they have insufficient security clearance are not visible			
TC008	User accesses a document they have access to	1) User logs in 2) User goes to list of all documents visible to them 3) User selects a document	1) Log in successful 2) Documents user has sufficient security clearance to see are visible 3) Document opens in viewer window	[TBD]	[TBD]	[TBD]
TC009	User attempts to access a document they do not have sufficient security clearance	1) User logs in 2) User attempts to access a document they do not have sufficient security clearance level to see using document ID on the command line (e.g. curl)	1) Log in successful 2) HTTP error 403: Unauthorized is returned instead of document contents	[TBD]	[TBD]	[TBD]

Test Results Summary

As of the delivery of this plan, our team has not yet performed UAT for the EDMS for PY 2025. When the UAT environment is ready (05/31/2025), the team will execute the test cases above, involving business users (employees, managers, external collaborators) to validate workflows and usability. Results will be documented in the table, with a target of 90% task completion rate and a user satisfaction score of ≥4/5. Defects will be logged in Jira, triaged daily, and resolved per the SLA (critical: 24 hours, high: 48 hours).

Version History

Version	Date	Author	Changes Made	Remarks
1.0	2025-04-28	Alice Brown	Initial creation of UAT test plan.	Initial version for review.
1.1	2025-04-29	Jane Doe	Added test cases for collaboration, approvals.	Updated based on SME feedback.
1.2	2025-04-30	Alice Brown	Included edge case for restricted access.	Finalized for internal review.

v. What will constitute successful tests and overall testing success

The decision to release a set of integrated software components into the production environment rests with management. They must balance the required deliverable content and quality against the current state of the software product. A number of low impact failing test cases may be acceptable, but the QA Manager would make this determination and would be responsible for any fallout from these decisions.

The key metrics for managers to make the decision to release include: test coverage percentage at the component and system levels, open defects and their potential impact, and confidence of test and development teams of product suitability for customer use.

They will use these metrics to manage risk and impact of defects occurring in the wild. Risk of undesired system performance. Risk of user data compromise. Risk of performance degradation. Risk of undesirable user experience.

Criteria such as these are rolled up into a decision by the QA Manager to determine success of the development and testing processes. The decision to release depends on the rollup of this information and the final decision by management.

Group participation

Our group worked collaboratively and equitably throughout the completion of this final exam. Philip Haapala drafted the initial response to **Question 1 (Components)**, while **Ehtasham Nasir** took the lead on **Questions 2 and 3 (Architecture and Framework)**. **James Kessler** developed the response for **Question 4 (Design Patterns)**, and **John Tempey** authored **Question 5 (Testing)**.

We held multiple group meetings to discuss each question, review each other's work, and ensure that our responses were cohesive, technically sound, and aligned with the course requirements. Every member contributed actively to feedback, revisions, and final proofing before submission. This exam represents a strong

collaborative effort and a shared understanding of the concepts and practices learned in CSC-580.