**1. Overview**

The Contract Monthly Claim System is a web-based application developed using ASP.NET Core MVC. It streamlines the submission, verification, and approval of monthly claims submitted by contract lecturers. The system is designed around three core principles:

* **Usability**: Simple, intuitive interfaces tailored to each user role.
* **Scalability**: Modular architecture and normalized database design for future expansion.
* **Transparency**: Clear status tracking and role-based workflows to ensure accountability.

**2. Design Choices**

**MVC Architecture in Context**

The system follows the Model-View-Controller (MVC) pattern, which separates the application into three layers:

* **Model**: Defines core entities such as Claim, Document, and Employee, and handles validation.
* **View**: Renders role-specific interfaces using Razor Pages (e.g., dashboards for lecturers, coordinators, and managers).
* **Controller**: Manages user input and orchestrates logic — e.g., LecturerController handles claim submission and document uploads.

This separation improves maintainability and testability. For example, GUI changes do not affect business logic, and workflows can be extended without disrupting the data layer.

**Workflow Logic**

The claim lifecycle follows a structured, role-based process:

1. **Lecturer Submission**  
   Lecturer submits a claim with hours, rate, notes, and supporting documents.  
   Claim is saved with status: Pending.
2. **Coordinator Verification**  
   Coordinator reviews pending claims.  
   Can either Verify (status → Verified) or Reject.
3. **Manager Approval**  
   Manager reviews verified claims.  
   Can Approve (status → Approved) or Reject.

This layered workflow mirrors real-world academic processes and ensures accountability at each stage. The use of two distinct status fields (VerifyStatus, ApproveStatus) prevents premature approvals and supports granular tracking.

**Role-Based Access and Views**

The system supports role-based authentication, allowing users to access only the views and actions relevant to their role:

* **Lecturer**: Can submit claims, upload documents, and view their own claim history.
* **Coordinator**: Can view and verify pending claims.
* **Manager**: Can view verified claims and approve or reject them.

MVC makes this easy to implement by routing each role to its own controller and view folder (Views/Lecturer, Views/Coordinator, Views/Manager). Authorization filters can be added to enforce access boundaries and prevent unauthorized actions.

**3. Database Structure**

The system uses a normalized relational schema centered around three entities:

* **Employee Table**  
  Stores lecturer details: EmployeeID, EmployeeName, Department, ContactInfo.
* **Claim Table**  
  Stores claim data: ClaimID, EmployeeID, HoursWorked, ClaimDate, VerifyStatus, ApproveStatus.
* **SupportingDocuments Table**  
  Stores uploaded files: DocumentID, ClaimID, FileName, FilePath, EncryptedData.

This structure avoids redundancy and supports scalability — for example, lecturers can submit multiple claims, each with multiple documents. Linking documents to claims via foreign keys ensures data integrity and flexibility.

**4. GUI Layout and Structure**

The graphical user interface (GUI) is designed for clarity, speed, and minimal user error. Each role is presented with a tailored dashboard:

**Lecturer Dashboard**

* Simple form with fields for hours, rate, notes, and file upload.
* File validation and encryption handled on submission.
* Visual feedback via success messages and claim history table.

**Coordinator Dashboard**

* Table of pending claims with Verify and Reject buttons.
* Color-coded status badges (yellow = pending, blue = verified).
* Document links open in new tabs for quick review.

**Manager Dashboard**

* Table of verified claims with Approve and Reject buttons.
* Green = approved, red = rejected.
* Status updates are immediate and intuitive.

**Visual Enhancements**

* Progress bars show claim status evolution.
* Badges provide instant status recognition.
* Buttons are used instead of dropdowns to reduce input errors.

The GUI is responsive and designed to minimize cognitive load, ensuring that users can complete tasks quickly and confidently.

**5. Assumptions and Constraints**

**Assumptions**

1. **User Competency**  
   Users (lecturers, coordinators, managers) have basic digital literacy and can navigate forms and buttons.
2. **Infrastructure Availability**  
   The system will be deployed in environments with reliable computers and internet access.
3. **Role-Based Access**  
   Users will log in under their correct roles, ensuring that only authorized users can verify or approve claims.
4. **Prototype Scope**  
   The current prototype is front-end only; claim status updates and database operations are simulated. Future phases will integrate a full backend and authentication system.
5. **Consistency of Processes**  
   Organizational claim policies are assumed to be consistent across departments, validating the two-stage review process.

**Constraints**

1. **Time Constraint**  
   Development must be completed within the academic POE PART 1 timeline (six weeks), limiting full backend implementation.
2. **Prototype Limitation**  
   The system currently provides a visual representation of functionality, with no requirement for data persistence or advanced logic.
3. **Technology Stack Constraint**  
   The project is restricted to using .NET Core (MVC or WPF) for the GUI, as per POE requirements.
4. **Version Control Requirement**  
   A minimum of five commits to GitHub with descriptive messages is mandatory, enforcing structured version control.
5. **Platform Constraint**  
   The application is designed for Windows-based systems. While cross-platform compatibility is not addressed in the prototype phase, future iterations may include Linux and macOS support.

**6. Rationale**

Every design decision balances user-friendliness, system reliability, and long-term scalability. By adopting MVC, a normalized database design, and an intuitive GUI, the system ensures it can evolve into a fully functional production solution while remaining easy to use for non-technical staff. The layered workflow and role-based views mirror real-world academic structures, making the system both practical and extensible.

