Technical Summary Document

# 1. Problem Statement

This solution implements a simplified Insurance Claims Management System designed to:  
- Allow hospitals or policyholders to submit insurance claims.  
- Enable admin users to view and update the status of claims.  
- Maintain core insurance entities such as users, policyholders, policies, and claims.  
- Track claim details and statuses effectively.  
The system addresses the need for a basic, functional backend API for claim submission and approval workflows within a constrained 2-hour development timeframe.

# 2. Design Considerations

- Architecture:  
 Adopted a Clean Architecture approach with clear separation of concerns into layers: API (Controllers), Services (Business Logic), and Data Access (Entity Framework Core). This ensures maintainability and testability.  
- Patterns:  
 - Repository pattern to abstract data operations.  
 - Dependency Injection for loose coupling and testability.  
 - Async/Await for non-blocking IO operations.  
- Key Decisions & Trade-offs:  
 - JWT-based authentication was omitted to fit the 2-hour limit; user roles are handled via a simple string property.  
 - Policy type table was removed for simplicity.  
 - Used Serilog for structured logging instead of a more complex monitoring stack.  
 - Limited the domain to essential entities only (Users, Policyholders, Policies, Claims).  
 - Pagination implemented on list endpoints to optimize performance.

# 3. Technical Features Implemented

- Dependency Injection:  
 Utilized the built-in ASP.NET Core DI container to inject repositories and services into controllers.  
  
 - Integrated Serilog for logging to console and file with structured logs.  
- External Libraries/Packages Used:  
 - Entity Framework Core (local DB) for ORM and persistent data storage.  
 - Serilog for logging.  
- Async/Await:  
 All data access and service methods are asynchronous to improve scalability and responsiveness.  
- Pagination:  
 Implemented pagination for list endpoints (e.g., claims listing) to improve performance and usability when handling large datasets.

# 4. Known Limitations and Future Scope

- Authentication & Authorization:  
 Currently, no JWT or token-based authentication; security can be improved by integrating ASP.NET Core Identity or JWT tokens.  
- Role Management:  
 Roles are handled as a string property; implementing proper role entities and claims would enhance flexibility.  
- API Features:  
 Limited CRUD operations implemented; future work could include advanced filtering, sorting, and comprehensive validation.  
- UI Layer:  
 No frontend/UI included; adding a web or mobile client would complete the user experience.  
- Notifications & Auditing:  
 Could add email notifications and audit trails for claims.  
- Performance & Scalability:  
 Optimization and caching layers can be added for production readiness.