Table of Contents

# Insurance System Technical Design Document

## Document Metadata

* **Document ID**: TDD-INS-2024-001
* **Version**: 1.0
* **Status**: Draft
* **Created**: 2024-03-15
* **Last Updated**: 2024-03-15
* **Author**: Technical Design Team
* **Reviewers**:
  + John Doe (Lead Architect)
  + Jane Smith (Security Expert)
  + Mike Johnson (DevOps Lead)
  + Sarah Williams (QA Manager)
* **Approvers**:
  + CTO
  + Product Owner
  + Security Officer

## Table of Contents

1. [Executive Summary](#executive-summary)
2. [Project Overview](#X1c555f901308e0ca1fe83531c356e1a10b9c010)
3. [High-Level Functionality](#X51bff9d2ae314aee317c48691019190212af7a4)
4. [Detailed Functionality Implementation](#Xbc7448207c4069c786df8d2433dbf17ecbce68d)
5. [Service Implementation Details](#X6a89d1eae10683489b1473ba391490116c50b77)
6. [Integration Points](#X5202e3cfc11be3dab2d8f5f285e15b82848439a)
7. [Error Handling and Monitoring](#X038bfe51298267caa93e79c67826043deb97e47)
8. [System Architecture](#X1fee9a95e3db63f825d3e0659da26059b86ade2)
9. [Data Flow](#Xd54ad638ce933fe741bbbf0f6234aefab169f47)
10. [Security Considerations](#Xceb666e44c796cd82f1d67eae8ae76338044d64)
11. [Deployment Architecture](#X6332cc5cf84dfa4849fa5f85b02a678e218890a)
12. [Performance Considerations](#X575b956ebbbab97de830121db541a3a12acacaf)
13. [Testing Strategy](#X4a055e139979e8c8247f6f3193f5d7453c63391)
14. [API Documentation](#Xdc077aee43b6d538c6a13d96f7680fadb216965)
15. [Scalability Considerations](#X9c927935349bcbbcfa660283b885e6878377988)
16. [Future Enhancements](#Xb7f4b924ebc9e256d3cf7e47b47e68d719dccf5)
17. [Non-Functional Requirements](#X51036e9b60ca4cad00cb8aae8f96b91d8c8efcd)
18. [Compliance Requirements](#Xea8f7a060a955e8bdb5918ccfed65e6d169ffab)
19. [Data Migration Strategy](#X1fc287633c40156a1e3aa752fe0db9314f6d0bb)
20. [Disaster Recovery Plan](#Xa00a13ad55412ac82fac206494129b9484615b0)
21. [Cost Estimation](#X4745057f5c301a6b20526c98a96d5286799430e)
22. [Team Structure and Responsibilities](#X83c0e11a4d3389687ea8fd86d46422f20c635fd)
23. [Third-Party Dependencies](#Xa75f877ac8c4762f380ac13560c220475ba266a)
24. [API Versioning Strategy](#X0e7bfc41464ac8e6a53b1de8e3d2fa4ca796b92)
25. [Documentation Standards](#Xe6c573a8f72c10a3a3988d4fc600666ae844a0e)
26. [Maintenance Plan](#X4493a9fd703620444746b2b1d0905f0643b2041)
27. [Glossary](#glossary)
28. [References](#references)
29. [Appendix](#appendix)
30. [Revision History](#revision-history)

## Executive Summary

This technical design document outlines the architecture and implementation details of the Insurance System, a comprehensive solution for automated insurance application processing, risk assessment, and premium calculation. The system combines modern web technologies with AI capabilities to streamline the insurance underwriting process.

Key highlights: - AI-powered risk assessment and premium calculation - Secure and scalable architecture - Compliance with industry standards - Comprehensive monitoring and maintenance - Future-proof design for enhancements

## 1. Project Overview

### 1.1 Purpose

The Insurance System is a comprehensive solution for automated insurance application processing, risk assessment, and premium calculation. It combines modern web technologies with AI capabilities to streamline the insurance underwriting process.

### 1.2 Key Features

* Automated insurance application processing
* AI-powered risk assessment
* Dynamic premium calculation
* Medical history analysis
* Fraud detection
* Real-time quote generation

### 1.3 Technology Stack

* Frontend: Angular (TypeScript)
* Backend: FastAPI (Python)
* AI Processing: Python-based LLM services
* Databases: PostgreSQL, ChromaDB
* Vector Store: ChromaDB for medical condition embeddings

## 2. High-Level Functionality

### 2.1 Application Processing

* **Purpose**: Handle insurance application submissions and processing
* **Key Components**:
  + Application validation
  + Data enrichment
  + Risk assessment triggering
  + Premium calculation
  + Decision making

### 2.2 Risk Assessment

* **Purpose**: Evaluate applicant risk profile
* **Key Components**:
  + Medical history analysis
  + Lifestyle factor assessment
  + Age-based risk calculation
  + Occupation risk evaluation
  + AI-powered insights

### 2.3 Premium Calculation

* **Purpose**: Determine insurance premium based on risk factors
* **Key Components**:
  + Base rate calculation
  + Risk multiplier application
  + Age factor consideration
  + Medical history impact
  + Lifestyle factor adjustment

## 3. Detailed Functionality Implementation

### 3.1 Application Processing

#### 3.1.1 Application Validation

* **Location**: backend/app/api/endpoints/insurance.py
* **Key Functions**:
* @router.post("/applications")  
  async def create\_application(application: ApplicationCreate):  
   # Validates application data  
   validated\_data = await validate\_application\_data(application)  
   # Processes medical history  
   medical\_history = await process\_medical\_history(validated\_data.medical\_history)  
   # Triggers risk assessment  
   risk\_assessment = await ai\_service.assess\_risk(validated\_data)

#### 3.1.2 Data Processing

* **Location**: backend/app/services/application\_processor.py
* **Key Functions**:
* class ApplicationProcessor:  
   async def process\_application(self, application\_data: Dict[str, Any]):  
   # Data enrichment  
   enriched\_data = await self.enrich\_application\_data(application\_data)  
   # Medical history processing  
   medical\_analysis = await self.process\_medical\_history(enriched\_data)  
   # Risk assessment  
   risk\_score = await self.calculate\_risk\_score(medical\_analysis)

### 3.2 Risk Assessment Implementation

#### 3.2.1 Medical Risk Analysis

* **Location**: backend/app/services/medical\_risk\_analysis.py
* **Key Functions**:
* def analyze\_medical\_risk(medical\_history: Dict[str, Any]) -> Dict[str, Any]:  
   # Vector similarity search  
   matches = vector\_store.search(medical\_history["conditions"])  
   # Risk score calculation  
   risk\_score = calculate\_risk\_score(matches)  
   # Assessment generation  
   assessment = generate\_risk\_assessment(risk\_score)

#### 3.2.2 Lifestyle Risk Assessment

* **Location**: backend/app/schemas/insurance.py
* **Key Functions**:
* class RiskFactorsBase(BaseModel):  
   def calculate\_risk\_contribution(self) -> float:  
   # Smoking impact  
   if self.smoking:  
   base\_risk += 0.3  
   # Alcohol impact  
   if self.alcohol\_consumption:  
   base\_risk += 0.15  
   # Activity risk  
   activity\_risk = min(0.4, len(self.dangerous\_activities) \* 0.1)

### 3.3 Premium Calculation Implementation

#### 3.3.1 Base Rate Calculation

* **Location**: backend/app/services/premium\_calculator.py
* **Key Functions**:
* class PremiumCalculatorService:  
   def get\_base\_rate(self, coverage\_type: str) -> float:  
   # Load base rates from configuration  
   base\_rates = self.load\_base\_rates()  
   # Return appropriate rate  
   return base\_rates.get(coverage\_type, 0.0)

#### 3.3.2 Risk Factor Application

* **Location**: backend/app/services/premium\_calculator.py
* **Key Functions**:
* async def calculate\_premium(self, application\_data: ApplicationData) -> Premium:  
   # Base rate  
   base\_rate = self.get\_base\_rate(application\_data.coverage\_type)  
   # Risk multiplier  
   risk\_multiplier = await self.calculate\_risk\_multiplier(risk\_assessment)  
   # Age factor  
   age\_factor = self.calculate\_age\_factor(application\_data.age)  
   # Final premium  
   final\_premium = base\_rate \* risk\_multiplier \* age\_factor

### 3.4 Frontend Implementation

#### 3.4.1 Application Form Component

* **Location**: frontend/src/app/components/application-form/application-form.component.ts
* **Key Functions**:
* @Component({  
   selector: 'app-application-form',  
   templateUrl: './application-form.component.html'  
  })  
  export class ApplicationFormComponent {  
   async onSubmit(applicationData: ApplicationFormData) {  
   // Form validation  
   if (!this.validateForm(applicationData)) {  
   return;  
   }  
   // API call  
   const response = await this.insuranceService.submitApplication(applicationData);  
   // Handle response  
   this.handleApplicationResponse(response);  
   }  
    
   private validateForm(data: ApplicationFormData): boolean {  
   // Required field validation  
   if (!data.personalInfo || !data.medicalHistory) {  
   this.showError('Missing required fields');  
   return false;  
   }  
   // Age validation  
   if (data.personalInfo.age < 18 || data.personalInfo.age > 75) {  
   this.showError('Age must be between 18 and 75');  
   return false;  
   }  
   return true;  
   }  
  }

#### 3.4.2 Risk Assessment Display

* **Location**: frontend/src/app/components/risk-assessment/risk-assessment.component.ts
* **Key Functions**:
* @Component({  
   selector: 'app-risk-assessment',  
   templateUrl: './risk-assessment.component.html'  
  })  
  export class RiskAssessmentComponent {  
   async loadRiskAssessment(applicationId: string) {  
   // Fetch risk data  
   const riskData = await this.riskService.getRiskAssessment(applicationId);  
   // Update UI  
   this.updateRiskDisplay(riskData);  
   // Generate visualizations  
   this.generateRiskCharts(riskData);  
   }  
    
   private generateRiskCharts(riskData: RiskAssessmentData) {  
   // Medical risk chart  
   this.medicalRiskChart = this.createChart({  
   type: 'radar',  
   data: riskData.medicalFactors  
   });  
   // Lifestyle risk chart  
   this.lifestyleRiskChart = this.createChart({  
   type: 'bar',  
   data: riskData.lifestyleFactors  
   });  
   }  
  }

## 4. Service Implementation Details

### 4.1 AI Processing Services

#### 4.1.1 LLM Service

* **Location**: backend/app/services/llm\_service.py
* **Key Components**:
* class LLMService:  
   async def process\_application(self, application\_data: ApplicationData):  
   # Context preparation  
   context = self.prepare\_application\_context(application\_data)  
   # Embedding generation  
   embeddings = await self.generate\_embeddings(context)  
   # Analysis generation  
   analysis = await self.generate\_analysis(embeddings)  
   # Insight extraction  
   insights = self.extract\_insights(analysis)

#### 4.1.2 Vector Store Service

* **Location**: backend/app/services/vector\_store.py
* **Key Components**:
* class VectorStore:  
   async def similarity\_search(self, query\_vector: np.ndarray):  
   # Vector preprocessing  
   processed\_vector = self.preprocess\_vector(query\_vector)  
   # Search execution  
   results = await self.collection.query(query\_embeddings=processed\_vector)  
   # Result processing  
   return self.process\_search\_results(results)

### 4.2 Database Services

#### 4.2.1 Application Storage

* **Location**: backend/app/models/insurance.py
* **Key Models**:
* class InsuranceApplication(Base):  
   \_\_tablename\_\_ = "insurance\_applications"  
   id = Column(Integer, primary\_key=True)  
   applicant\_name = Column(String)  
   applicant\_age = Column(Integer)  
   medical\_history = Column(JSON)  
   risk\_factors = Column(JSON)  
   premium\_amount = Column(Float)

#### 4.2.2 Risk Score Storage

* **Location**: backend/app/models/insurance.py
* **Key Models**:
* class RiskScore(Base):  
   \_\_tablename\_\_ = "risk\_scores"  
   id = Column(Integer, primary\_key=True)  
   application\_id = Column(Integer, ForeignKey("insurance\_applications.id"))  
   overall\_score = Column(Float)  
   medical\_factor = Column(Float)  
   age\_factor = Column(Float)  
   lifestyle\_factor = Column(Float)

### 4.3 Frontend Services

#### 4.3.1 Insurance Service

* **Location**: frontend/src/app/services/insurance.service.ts
* **Key Components**:
* @Injectable({  
   providedIn: 'root'  
  })  
  export class InsuranceService {  
   constructor(private http: HttpClient) {}  
    
   async submitApplication(application: ApplicationData): Promise<ApplicationResponse> {  
   // API call  
   return this.http.post<ApplicationResponse>(  
   '/api/applications',  
   application,  
   { headers: this.getAuthHeaders() }  
   ).toPromise();  
   }  
    
   async getQuote(applicationId: string): Promise<Quote> {  
   // Quote retrieval  
   return this.http.get<Quote>(  
   `/api/quotes/${applicationId}`,  
   { headers: this.getAuthHeaders() }  
   ).toPromise();  
   }  
  }

#### 4.3.2 Risk Service

* **Location**: frontend/src/app/services/risk.service.ts
* **Key Components**:
* @Injectable({  
   providedIn: 'root'  
  })  
  export class RiskService {  
   async getRiskAssessment(applicationId: string): Promise<RiskAssessment> {  
   // Risk data retrieval  
   return this.http.get<RiskAssessment>(  
   `/api/risk-assessment/${applicationId}`,  
   { headers: this.getAuthHeaders() }  
   ).toPromise();  
   }  
    
   async getRiskHistory(applicationId: string): Promise<RiskHistory[]> {  
   // Historical risk data  
   return this.http.get<RiskHistory[]>(  
   `/api/risk-history/${applicationId}`,  
   { headers: this.getAuthHeaders() }  
   ).toPromise();  
   }  
  }

## 5. Integration Points

### 5.1 API Endpoints

* **Location**: backend/app/api/endpoints/insurance.py
* **Key Endpoints**:
* @router.post("/calculate-premium")  
  async def premium\_calculation(request: PremiumCalculationRequest):  
   # Risk analysis  
   risk\_analysis = analyze\_medical\_risk(request.medical\_history)  
   # Premium calculation  
   premium\_data = calculate\_premium({  
   "applicant\_age": request.applicant\_age,  
   "coverage\_amount": request.coverage\_amount,  
   "risk\_analysis": risk\_analysis  
   })

### 5.2 Service Integration

* **Location**: backend/app/services/integration.py
* **Key Components**:
* class ServiceIntegrator:  
   async def process\_application(self, application\_data: Dict[str, Any]):  
   # AI processing  
   ai\_analysis = await self.llm\_service.process\_application(application\_data)  
   # Risk assessment  
   risk\_score = await self.risk\_service.assess\_risk(ai\_analysis)  
   # Premium calculation  
   premium = await self.premium\_service.calculate\_premium(risk\_score)

### 5.3 Frontend-Backend Integration

#### 5.3.1 API Communication

* **Location**: frontend/src/app/core/api.interceptor.ts
* **Key Components**:
* @Injectable()  
  export class ApiInterceptor implements HttpInterceptor {  
   intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {  
   // Add authentication  
   const authReq = req.clone({  
   headers: req.headers.set('Authorization', `Bearer ${this.auth.getToken()}`)  
   });  
    
   // Handle response  
   return next.handle(authReq).pipe(  
   catchError(error => this.handleError(error))  
   );  
   }  
    
   private handleError(error: HttpErrorResponse) {  
   // Error handling logic  
   if (error.status === 401) {  
   this.auth.logout();  
   }  
   return throwError(error);  
   }  
  }

#### 5.3.2 Real-time Updates

* **Location**: frontend/src/app/services/websocket.service.ts
* **Key Components**:
* @Injectable({  
   providedIn: 'root'  
  })  
  export class WebsocketService {  
   private socket: WebSocket;  
    
   connect(applicationId: string) {  
   this.socket = new WebSocket(`ws://api/updates/${applicationId}`);  
    
   this.socket.onmessage = (event) => {  
   const update = JSON.parse(event.data);  
   this.handleUpdate(update);  
   };  
   }  
    
   private handleUpdate(update: UpdateMessage) {  
   // Update application state  
   this.stateService.updateApplicationState(update);  
   // Notify subscribers  
   this.updateSubject.next(update);  
   }  
  }

## 6. Error Handling and Monitoring

### 6.1 Error Handling

#### 6.1.1 Backend Error Handling

* **Location**: backend/app/core/error\_handling.py
* **Key Components**:
* class ErrorHandler:  
   async def handle\_error(self, error: Exception, context: Dict[str, Any]):  
   # Error classification  
   error\_type = self.classify\_error(error)  
   # Error logging  
   await self.log\_error(error, context)  
   # Error response generation  
   response = self.generate\_error\_response(error\_type)  
   # Monitoring notification  
   await self.notify\_monitoring(error\_type, context)  
   return response  
    
   def classify\_error(self, error: Exception) -> str:  
   if isinstance(error, ValidationError):  
   return "VALIDATION\_ERROR"  
   elif isinstance(error, DatabaseError):  
   return "DATABASE\_ERROR"  
   elif isinstance(error, AIProcessingError):  
   return "AI\_PROCESSING\_ERROR"  
   return "UNKNOWN\_ERROR"  
    
   async def log\_error(self, error: Exception, context: Dict[str, Any]):  
   logger.error({  
   "error\_type": type(error).\_\_name\_\_,  
   "error\_message": str(error),  
   "context": context,  
   "timestamp": datetime.now().isoformat()  
   })

#### 6.1.2 Frontend Error Handling

* **Location**: frontend/src/app/core/error-handler.service.ts
* **Key Components**:
* @Injectable({  
   providedIn: 'root'  
  })  
  export class ErrorHandlerService {  
   handleError(error: any) {  
   // Error classification  
   const errorType = this.classifyError(error);  
   // User notification  
   this.notifyUser(errorType);  
   // Error logging  
   this.logError(error);  
   // Error recovery  
   this.handleErrorRecovery(errorType);  
   }  
    
   private classifyError(error: any): ErrorType {  
   if (error instanceof HttpErrorResponse) {  
   return this.classifyHttpError(error);  
   }  
   return ErrorType.UNKNOWN;  
   }  
    
   private handleErrorRecovery(errorType: ErrorType) {  
   switch (errorType) {  
   case ErrorType.NETWORK:  
   this.retryOperation();  
   break;  
   case ErrorType.AUTH:  
   this.handleAuthError();  
   break;  
   case ErrorType.VALIDATION:  
   this.handleValidationError();  
   break;  
   }  
   }  
  }

### 6.2 Monitoring

#### 6.2.1 Backend Monitoring

* **Location**: backend/app/services/monitoring.py
* **Key Components**:
* class MonitoringService:  
   async def monitor\_system(self):  
   # System health check  
   health\_status = await self.check\_system\_health()  
   # Performance metrics  
   performance\_metrics = await self.collect\_performance\_metrics()  
   # Resource usage  
   resource\_usage = await self.monitor\_resources()  
   # Error rates  
   error\_rates = await self.calculate\_error\_rates()  
    
   return {  
   "health\_status": health\_status,  
   "performance\_metrics": performance\_metrics,  
   "resource\_usage": resource\_usage,  
   "error\_rates": error\_rates  
   }  
    
   async def check\_system\_health(self) -> Dict[str, Any]:  
   return {  
   "database": await self.check\_database\_health(),  
   "ai\_services": await self.check\_ai\_services\_health(),  
   "api\_endpoints": await self.check\_api\_health(),  
   "background\_tasks": await self.check\_background\_tasks()  
   }  
    
   async def collect\_performance\_metrics(self) -> Dict[str, Any]:  
   return {  
   "response\_times": await self.measure\_response\_times(),  
   "throughput": await self.calculate\_throughput(),  
   "concurrent\_users": await self.count\_concurrent\_users(),  
   "api\_latency": await self.measure\_api\_latency()  
   }

#### 6.2.2 Frontend Monitoring

* **Location**: frontend/src/app/services/monitoring.service.ts
* **Key Components**:
* @Injectable({  
   providedIn: 'root'  
  })  
  export class MonitoringService {  
   private performanceMetrics: PerformanceMetrics = {  
   pageLoadTimes: new Map<string, number>(),  
   apiResponseTimes: new Map<string, number>(),  
   errorCounts: new Map<string, number>()  
   };  
    
   trackPageLoad(pageName: string) {  
   const loadTime = performance.now();  
   this.performanceMetrics.pageLoadTimes.set(pageName, loadTime);  
   this.sendMetrics('page\_load', { pageName, loadTime });  
   }  
    
   trackApiCall(endpoint: string, duration: number) {  
   this.performanceMetrics.apiResponseTimes.set(endpoint, duration);  
   this.sendMetrics('api\_call', { endpoint, duration });  
   }  
    
   trackError(error: Error, context: any) {  
   const errorType = error.constructor.name;  
   const count = this.performanceMetrics.errorCounts.get(errorType) || 0;  
   this.performanceMetrics.errorCounts.set(errorType, count + 1);  
   this.sendMetrics('error', { errorType, context });  
   }  
    
   private sendMetrics(type: string, data: any) {  
   // Send metrics to monitoring backend  
   this.http.post('/api/monitoring/metrics', {  
   type,  
   data,  
   timestamp: new Date().toISOString()  
   }).subscribe();  
   }  
  }

## 7. System Architecture

### 7.1 High-Level Architecture

graph TB  
 subgraph Client[Client Layer]  
 Web[Web Browser]  
 Mobile[Mobile App]  
 end  
  
 subgraph Frontend[Frontend Layer]  
 Angular[Angular Application]  
 Auth[Authentication Service]  
 Cache[Client-side Cache]  
 end  
  
 subgraph Backend[Backend Layer]  
 API[API Gateway]  
 Services[Microservices]  
 Queue[Message Queue]  
 end  
  
 subgraph AI[AI Processing Layer]  
 LLM[LLM Service]  
 Vector[Vector Store]  
 Crew[Crew AI]  
 end  
  
 subgraph Data[Data Layer]  
 SQL[PostgreSQL]  
 VectorDB[ChromaDB]  
 CacheDB[Redis Cache]  
 end  
  
 Client --> Frontend  
 Frontend --> Backend  
 Backend --> AI  
 Backend --> Data  
 AI --> Data

### 7.2 Component Interaction

* **Frontend-Backend**: RESTful APIs with WebSocket for real-time updates
* **Backend-AI**: Asynchronous message queue for AI processing
* **AI-Data**: Direct database access for vector storage and retrieval
* **Backend-Data**: ORM-based database access with caching

## 8. Data Flow

### 8.1 Application Processing Flow

sequenceDiagram  
 participant Client  
 participant Frontend  
 participant API  
 participant Processor  
 participant AI  
 participant DB  
  
 Client->>Frontend: Submit Application  
 Frontend->>API: POST /applications  
 API->>Processor: Process Application  
 Processor->>AI: Request Risk Assessment  
 AI->>DB: Query Medical History  
 DB-->>AI: Return Results  
 AI-->>Processor: Risk Assessment  
 Processor->>DB: Store Results  
 DB-->>Processor: Confirmation  
 Processor-->>API: Processed Application  
 API-->>Frontend: Application Response  
 Frontend-->>Client: Display Results

### 8.2 Data Models

# Application Data Model  
class ApplicationData(BaseModel):  
 personal\_info: PersonalInfo  
 medical\_history: MedicalHistory  
 lifestyle\_factors: LifestyleFactors  
 risk\_assessment: Optional[RiskAssessment]  
 premium: Optional[Premium]  
  
# Risk Assessment Model  
class RiskAssessment(BaseModel):  
 overall\_score: float  
 medical\_factors: Dict[str, float]  
 lifestyle\_factors: Dict[str, float]  
 recommendations: List[str]  
 confidence\_score: float

## 9. Security Considerations

### 9.1 Authentication & Authorization

* **JWT-based Authentication**
  + Token expiration: 1 hour
  + Refresh token rotation
  + Role-based access control
* **API Security**
  + Rate limiting
  + Request validation
  + CORS configuration
* **Data Security**
  + Encryption at rest
  + TLS for data in transit
  + PII data masking

### 9.2 Security Implementation

# Authentication Middleware  
class AuthMiddleware:  
 async def verify\_token(self, token: str) -> User:  
 try:  
 payload = jwt.decode(token, SECRET\_KEY, algorithms=[ALGORITHM])  
 return await self.get\_user(payload["sub"])  
 except JWTError:  
 raise HTTPException(status\_code=401)  
  
# Role-based Access Control  
class RBAC:  
 async def check\_permission(self, user: User, resource: str, action: str):  
 role = await self.get\_user\_role(user.id)  
 return await self.verify\_permission(role, resource, action)

## 10. Deployment Architecture

### 10.1 Infrastructure

* **Containerization**: Docker containers for all services
* **Orchestration**: Kubernetes for container management
* **CI/CD**: GitHub Actions for automated deployment
* **Monitoring**: Prometheus + Grafana for metrics
* **Logging**: ELK stack for log management

### 10.2 Deployment Configuration

# Kubernetes Deployment  
apiVersion: apps/v1  
kind: Deployment  
metadata:  
 name: insurance-api  
spec:  
 replicas: 3  
 template:  
 spec:  
 containers:  
 - name: api  
 image: insurance-api:latest  
 resources:  
 limits:  
 cpu: "1"  
 memory: "1Gi"  
 env:  
 - name: DB\_HOST  
 valueFrom:  
 secretKeyRef:  
 name: db-secret  
 key: host

## 11. Performance Considerations

### 11.1 Caching Strategy

* **Client-side Caching**
  + Application data: 5 minutes
  + Risk assessments: 1 hour
* **Server-side Caching**
  + Database queries: Redis
  + Vector embeddings: In-memory cache
* **CDN Integration**
  + Static assets
  + API responses

### 11.2 Performance Optimization

# Query Optimization  
class OptimizedQuery:  
 async def get\_application\_data(self, application\_id: str):  
 # Use materialized views  
 return await self.db.execute(  
 """  
 SELECT \* FROM application\_materialized\_view  
 WHERE id = :id  
 """,  
 {"id": application\_id}  
 )  
  
# Batch Processing  
class BatchProcessor:  
 async def process\_applications(self, applications: List[Application]):  
 # Process in batches of 100  
 for batch in self.chunk(applications, 100):  
 await self.process\_batch(batch)

## 12. Testing Strategy

### 12.1 Test Types

* **Unit Tests**: Component-level testing
* **Integration Tests**: Service interaction testing
* **E2E Tests**: Full workflow testing
* **Performance Tests**: Load and stress testing
* **Security Tests**: Vulnerability scanning

### 12.2 Test Implementation

# Unit Test Example  
class TestRiskAssessment(unittest.TestCase):  
 def test\_medical\_risk\_calculation(self):  
 assessment = RiskAssessment()  
 result = assessment.calculate\_medical\_risk(test\_data)  
 self.assertGreater(result.score, 0)  
 self.assertLess(result.score, 1)  
  
# Integration Test Example  
class TestApplicationFlow(IntegrationTest):  
 async def test\_complete\_flow(self):  
 application = await self.create\_application()  
 assessment = await self.get\_risk\_assessment(application.id)  
 premium = await self.calculate\_premium(assessment)  
 self.assertIsNotNone(premium.amount)

## 13. API Documentation

### 13.1 Endpoint Specifications

# Application Submission  
/applications:  
 post:  
 summary: Submit new application  
 requestBody:  
 content:  
 application/json:  
 schema:  
 type: object  
 properties:  
 personal\_info:  
 type: object  
 properties:  
 name: {type: string}  
 age: {type: integer}  
 gender: {type: string}  
 medical\_history:  
 type: object  
 properties:  
 conditions: {type: array}  
 medications: {type: array}  
 lifestyle\_factors:  
 type: object  
 properties:  
 smoking: {type: boolean}  
 alcohol: {type: boolean}  
 activities: {type: array}  
 responses:  
 201:  
 description: Application created  
 content:  
 application/json:  
 schema:  
 type: object  
 properties:  
 application\_id: {type: string}  
 status: {type: string}  
 risk\_score: {type: number}  
 premium: {type: number}

## 14. Scalability Considerations

### 14.1 Horizontal Scaling

* **Frontend**: Stateless design for easy scaling
* **Backend**: Microservices architecture
* **Database**: Read replicas for PostgreSQL
* **AI Services**: Containerized deployment

### 14.2 Load Balancing

# Load Balancer Configuration  
class LoadBalancer:  
 def \_\_init\_\_(self):  
 self.servers = []  
 self.current = 0  
  
 def get\_server(self):  
 server = self.servers[self.current]  
 self.current = (self.current + 1) % len(self.servers)  
 return server

## 15. Future Enhancements

### 15.1 Planned Features

* **AI Model Improvements**
  + Enhanced medical condition analysis
  + Better fraud detection
  + Automated document processing
* **User Experience**
  + Mobile app development
  + Real-time chat support
  + Enhanced visualization
* **Integration**
  + Third-party medical data providers
  + Insurance comparison tools
  + Payment gateway integration

### 15.2 Technical Roadmap

* **Q1 2024**: Mobile app development
* **Q2 2024**: Enhanced AI models
* **Q3 2024**: Payment integration
* **Q4 2024**: Advanced analytics

This technical design document provides a comprehensive overview of the system’s architecture, implementation details, and integration points. It serves as a reference for developers working on the system and helps maintain consistency in implementation.

## 16. Non-Functional Requirements

### 16.1 Performance Requirements

* **Response Time**
  + API endpoints: < 200ms for 95% of requests
  + Risk assessment: < 5 seconds
  + Premium calculation: < 2 seconds
* **Throughput**
  + Peak load: 1000 requests/second
  + Average load: 100 requests/second
* **Availability**
  + System uptime: 99.99%
  + Scheduled maintenance window: 2 hours/month

### 16.2 Scalability Requirements

* **User Load**
  + Concurrent users: 10,000
  + Daily active users: 100,000
* **Data Volume**
  + Applications per day: 50,000
  + Storage growth: 1TB/year
* **Processing Capacity**
  + Risk assessments per hour: 10,000
  + Premium calculations per hour: 20,000

## 17. Compliance Requirements

### 17.1 Data Protection

* **GDPR Compliance**
  + Data retention policies
  + Right to be forgotten
  + Data portability
* **HIPAA Compliance**
  + PHI data handling
  + Encryption standards
  + Access controls
* **PCI DSS Compliance**
  + Payment data security
  + Transaction logging
  + Security monitoring

### 17.2 Audit Requirements

# Audit Logging  
class AuditLogger:  
 async def log\_operation(self, operation: str, user: User, data: Dict[str, Any]):  
 await self.db.execute(  
 """  
 INSERT INTO audit\_logs   
 (operation, user\_id, timestamp, data)  
 VALUES (:operation, :user\_id, :timestamp, :data)  
 """,  
 {  
 "operation": operation,  
 "user\_id": user.id,  
 "timestamp": datetime.now(),  
 "data": json.dumps(data)  
 }  
 )

## 18. Data Migration Strategy

### 18.1 Migration Plan

* **Phase 1**: Schema Migration
  + Database structure updates
  + Index optimization
  + Constraint implementation
* **Phase 2**: Data Migration
  + Batch processing
  + Data validation
  + Error handling
* **Phase 3**: Verification
  + Data integrity checks
  + Performance validation
  + Rollback testing

### 18.2 Migration Tools

# Data Migration Script  
class DataMigrator:  
 async def migrate\_data(self, source\_db: Database, target\_db: Database):  
 # Batch size configuration  
 batch\_size = 1000  
 # Migration progress tracking  
 progress = await self.get\_migration\_progress()  
   
 while not progress.complete:  
 batch = await source\_db.get\_batch(batch\_size)  
 await target\_db.insert\_batch(batch)  
 await self.update\_progress(progress)

## 19. Disaster Recovery Plan

### 19.1 Recovery Objectives

* **RTO (Recovery Time Objective)**: 4 hours
* **RPO (Recovery Point Objective)**: 1 hour
* **Critical Systems Priority**
  1. Database services
  2. API services
  3. AI processing
  4. Frontend services

### 19.2 Backup Strategy

# Automated Backup  
class BackupManager:  
 async def perform\_backup(self):  
 # Database backup  
 await self.backup\_database()  
 # Configuration backup  
 await self.backup\_config()  
 # Vector store backup  
 await self.backup\_vector\_store()  
   
 async def verify\_backup(self):  
 # Integrity check  
 integrity = await self.check\_backup\_integrity()  
 # Restore test  
 await self.test\_restore()

## 20. Cost Estimation

### 20.1 Infrastructure Costs

* **Cloud Services**
  + Compute: $5,000/month
  + Storage: $1,000/month
  + Networking: $500/month
* **AI Services**
  + LLM API: $2,000/month
  + Vector Database: $1,000/month
* **Monitoring & Logging**
  + Prometheus: $500/month
  + ELK Stack: $1,000/month

### 20.2 Development Costs

* **Team Composition**
  + Backend Developers: 4
  + Frontend Developers: 3
  + DevOps Engineers: 2
  + AI Specialists: 2
* **Timeline**
  + Development: 6 months
  + Testing: 2 months
  + Deployment: 1 month

## 21. Team Structure and Responsibilities

### 21.1 Development Team

* **Backend Team**
  + API development
  + Database management
  + Integration services
* **Frontend Team**
  + UI/UX development
  + State management
  + Performance optimization
* **AI Team**
  + Model development
  + Data processing
  + Algorithm optimization

### 21.2 Operations Team

* **DevOps**
  + Infrastructure management
  + CI/CD pipeline
  + Monitoring setup
* **QA Team**
  + Test automation
  + Performance testing
  + Security testing

## 22. Third-Party Dependencies

### 22.1 External Services

* **AI Services**
  + OpenAI API
  + Vector database providers
  + Medical data APIs
* **Infrastructure**
  + Cloud providers (AWS/GCP)
  + CDN services
  + Monitoring tools

### 22.2 Dependency Management

# Dependency Version Control  
class DependencyManager:  
 def check\_dependencies(self):  
 # Security updates  
 security\_updates = await self.check\_security\_updates()  
 # Version compatibility  
 compatibility = await self.check\_version\_compatibility()  
 # License compliance  
 licenses = await self.verify\_licenses()

## 23. API Versioning Strategy

### 23.1 Version Control

* **Version Format**: v1.0.0
* **Deprecation Policy**: 6 months notice
* **Backward Compatibility**: Maintained for 1 year
* **Version Endpoints**
  + /api/v1/…
  + /api/v2/…

### 23.2 Version Implementation

# API Version Router  
class VersionRouter:  
 def \_\_init\_\_(self):  
 self.versions = {  
 "v1": V1Router(),  
 "v2": V2Router()  
 }  
   
 async def route\_request(self, request: Request):  
 version = request.headers.get("API-Version", "v1")  
 return await self.versions[version].handle\_request(request)

## 24. Documentation Standards

### 24.1 Code Documentation

* **Function Documentation**
  + Purpose
  + Parameters
  + Return values
  + Examples
* **Class Documentation**
  + Responsibilities
  + Dependencies
  + Usage examples

### 24.2 API Documentation

# API Documentation Generator  
class APIDocGenerator:  
 def generate\_docs(self, endpoint: Endpoint):  
 return {  
 "path": endpoint.path,  
 "method": endpoint.method,  
 "parameters": self.get\_parameters(endpoint),  
 "responses": self.get\_responses(endpoint),  
 "examples": self.get\_examples(endpoint)  
 }

## 25. Maintenance Plan

### 25.1 Regular Maintenance

* **Daily Tasks**
  + Log monitoring
  + Performance checks
  + Backup verification
* **Weekly Tasks**
  + Security updates
  + Performance optimization
  + Cache cleanup
* **Monthly Tasks**
  + System updates
  + Capacity planning
  + Cost optimization

### 25.2 Maintenance Procedures

# Maintenance Manager  
class MaintenanceManager:  
 async def perform\_maintenance(self):  
 # Pre-maintenance checks  
 await self.verify\_system\_state()  
 # Backup current state  
 await self.create\_snapshot()  
 # Execute maintenance tasks  
 await self.execute\_maintenance\_tasks()  
 # Post-maintenance verification  
 await self.verify\_system\_health()

## Glossary

* **API**: Application Programming Interface
* **LLM**: Large Language Model
* **RTO**: Recovery Time Objective
* **RPO**: Recovery Point Objective
* **PHI**: Protected Health Information
* **PII**: Personally Identifiable Information
* **CDN**: Content Delivery Network
* **CI/CD**: Continuous Integration/Continuous Deployment
* **ORM**: Object-Relational Mapping
* **JWT**: JSON Web Token

## References

1. **Industry Standards**
   * HIPAA Security Rule
   * GDPR Compliance Guidelines
   * PCI DSS Requirements
   * ISO 27001 Security Standards
2. **Technical References**
   * Angular Documentation
   * FastAPI Documentation
   * PostgreSQL Documentation
   * Kubernetes Documentation
   * OpenAI API Documentation
3. **Best Practices**
   * OWASP Security Guidelines
   * REST API Design Principles
   * Microservices Architecture Patterns
   * Cloud Security Best Practices

## Appendix

### A. System Diagrams

1. Component Architecture
2. Data Flow Diagrams
3. Deployment Architecture
4. Security Architecture

### B. API Specifications

1. Endpoint Details
2. Request/Response Examples
3. Error Codes
4. Authentication Flow

### C. Database Schema

1. Entity Relationship Diagram
2. Table Definitions
3. Index Specifications
4. Constraint Details

### D. Security Controls

1. Authentication Flow
2. Authorization Matrix
3. Encryption Standards
4. Audit Requirements

## Revision History

| Version | Date | Author | Changes | Reviewed By |
| --- | --- | --- | --- | --- |
| 0.1 | 2024-03-01 | Initial Draft | Initial structure | John Doe |
| 0.2 | 2024-03-05 | Added Architecture | Architecture details | Jane Smith |
| 0.3 | 2024-03-10 | Added Security | Security requirements | Mike Johnson |
| 0.4 | 2024-03-12 | Added Implementation | Implementation details | Sarah Williams |
| 1.0 | 2024-03-15 | Final Review | All sections complete | All Reviewers |

## Document Status

* **Current Status**: Draft
* **Next Review Date**: 2024-04-15
* **Approval Status**: Pending
* **Distribution List**:
  + Development Team
  + Architecture Team
  + Security Team
  + Operations Team
  + Product Management

## Formatting Guidelines

1. **Headings**
   * Use hierarchical structure (H1, H2, H3)
   * Maintain consistent formatting
   * Include in table of contents
2. **Code Blocks**
   * Use appropriate language highlighting
   * Include comments for clarity
   * Maintain consistent indentation
3. **Diagrams**
   * Use Mermaid notation
   * Include captions
   * Reference in text
4. **Tables**
   * Use markdown table syntax
   * Include headers
   * Align columns appropriately
5. **Lists**
   * Use appropriate bullet points
   * Maintain consistent indentation
   * Use numbered lists for sequences

## Document Control

* **Document Owner**: Technical Design Team
* **Document Location**: Confluence/SharePoint
* **Access Control**:
  + Read: All team members
  + Edit: Technical leads
  + Approve: Architecture team
* **Backup Location**: GitHub repository
* **Retention Policy**: 5 years
* **Review Cycle**: Quarterly

## Change Request Process

1. **Request Submission**
   * Use JIRA ticket template
   * Include impact analysis
   * Attach relevant documentation
2. **Review Process**
   * Technical review by architects
   * Security review by security team
   * Business review by product owners
3. **Approval Workflow**

* graph TD  
   A[Submit Change Request] --> B[Technical Review]  
   B --> C[Security Review]  
   C --> D[Business Review]  
   D --> E[Architecture Approval]  
   E --> F[Implement Change]  
   F --> G[Document Update]

1. **Implementation Guidelines**
   * Follow version control process
   * Update documentation
   * Conduct impact testing
   * Update regression tests

## Risk Assessment Matrix

### Technical Risks

| Risk | Probability | Impact | Mitigation Strategy |
| --- | --- | --- | --- |
| AI Model Accuracy | Medium | High | Regular model retraining |
| System Scalability | Low | High | Load testing and optimization |
| Data Security | Medium | Critical | Regular security audits |
| Integration Failures | Medium | High | Circuit breakers and fallbacks |

### Operational Risks

| Risk | Probability | Impact | Mitigation Strategy |
| --- | --- | --- | --- |
| Service Downtime | Low | High | High availability setup |
| Data Loss | Low | Critical | Regular backups |
| Performance Issues | Medium | High | Monitoring and alerts |
| Compliance Violations | Low | Critical | Regular audits |

## Dependencies Timeline

gantt  
 title Project Dependencies  
 dateFormat YYYY-MM-DD  
 section Infrastructure  
 Cloud Setup :2024-03-01, 30d  
 Database Setup :2024-03-15, 15d  
 section Development  
 API Development :2024-04-01, 45d  
 Frontend Development:2024-04-15, 60d  
 section AI  
 Model Training :2024-05-01, 30d  
 Integration :2024-06-01, 30d  
 section Testing  
 Unit Testing :2024-06-15, 30d  
 Integration Testing:2024-07-01, 30d

## Stakeholder Communication Plan

* **Development Team**
  + Daily standups
  + Weekly technical reviews
  + Sprint planning meetings
* **Business Stakeholders**
  + Bi-weekly status updates
  + Monthly demos
  + Quarterly roadmap reviews
* **Security Team**
  + Weekly security reviews
  + Monthly compliance checks
  + Quarterly security audits
* **Operations Team**
  + Daily monitoring reports
  + Weekly performance reviews
  + Monthly capacity planning

## Training Requirements

1. **Technical Training**
   * Angular development
   * FastAPI implementation
   * AI model management
   * Database optimization
   * Security best practices
2. **Operational Training**
   * System monitoring
   * Incident response
   * Performance optimization
   * Backup and recovery
   * Compliance procedures
3. **Business Training**
   * System functionality
   * Risk assessment process
   * Premium calculation
   * Reporting capabilities
   * User interface navigation

## Acceptance Criteria

1. **Functional Requirements**
   * All API endpoints working
   * Risk assessment accuracy > 95%
   * Premium calculation accuracy > 99%
   * System response time < 200ms
   * Data validation 100% accurate
2. **Non-Functional Requirements**
   * System availability 99.99%
   * Security compliance 100%
   * Documentation complete
   * Test coverage > 90%
   * Performance benchmarks met
3. **Quality Gates**
   * Code review passed
   * Security scan passed
   * Performance tests passed
   * Integration tests passed
   * User acceptance passed

## Quality Assurance Plan

1. **Testing Strategy**
   * Unit testing
   * Integration testing
   * Performance testing
   * Security testing
   * User acceptance testing
2. **Quality Metrics**
   * Code coverage
   * Bug density
   * Test pass rate
   * Performance metrics
   * Security compliance
3. **Review Process**
   * Code reviews
   * Design reviews
   * Security reviews
   * Performance reviews
   * Documentation reviews

## Resource Allocation

1. **Development Team**
   * 4 Backend Developers
   * 3 Frontend Developers
   * 2 DevOps Engineers
   * 2 AI Specialists
   * 1 Database Administrator
2. **Testing Team**
   * 2 QA Engineers
   * 1 Performance Tester
   * 1 Security Tester
   * 1 Automation Engineer
3. **Operations Team**
   * 2 System Administrators
   * 1 Security Officer
   * 1 Database Administrator
   * 1 Monitoring Specialist

## Project Timeline

gantt  
 title Project Timeline  
 dateFormat YYYY-MM-DD  
 section Phase 1  
 Requirements Gathering :2024-03-01, 30d  
 Architecture Design :2024-04-01, 30d  
 section Phase 2  
 Development :2024-05-01, 90d  
 Testing :2024-08-01, 60d  
 section Phase 3  
 Deployment :2024-10-01, 30d  
 Stabilization :2024-11-01, 30d

## Technical Specifications

### System Architecture Details

graph TB  
 subgraph Frontend[Angular Frontend]  
 FC[Quote Component]  
 FAD[Application Detail]  
 FQ[Quote Generation]  
 FRA[Risk Assessment Display]  
 FS[State Management]  
 FR[Routing Module]  
 FA[Auth Module]  
 end  
  
 subgraph Backend[FastAPI Backend]  
 BA[Application Router]  
 BPC[Premium Calculator]  
 BRA[Risk Assessment]  
 BDB[Database Service]  
 BC[Cache Service]  
 BMQ[Message Queue]  
 BAUTH[Auth Service]  
 end  
  
 subgraph AI[AI Processing Layer]  
 AILS[LLM Service]  
 AIVS[Vector Store]  
 AICA[Crew AI Orchestrator]  
 AIRA[Risk Assessment Pipeline]  
 AIM[Model Training]  
 AIV[Validation Service]  
 end  
  
 subgraph DB[Databases]  
 VDB[(Vector DB)]  
 SQL[(PostgreSQL)]  
 CACHE[(Redis)]  
 QUEUE[(RabbitMQ)]  
 end

### API Endpoints Specification

# Application API  
/applications:  
 post:  
 summary: Create new application  
 requestBody:  
 content:  
 application/json:  
 schema:  
 type: object  
 properties:  
 personal\_info:  
 type: object  
 properties:  
 name: {type: string}  
 age: {type: integer}  
 gender: {type: string}  
 medical\_history:  
 type: object  
 properties:  
 conditions: {type: array}  
 medications: {type: array}  
 lifestyle\_factors:  
 type: object  
 properties:  
 smoking: {type: boolean}  
 alcohol: {type: boolean}  
 activities: {type: array}  
 responses:  
 201:  
 description: Application created  
 content:  
 application/json:  
 schema:  
 type: object  
 properties:  
 application\_id: {type: string}  
 status: {type: string}  
 risk\_score: {type: number}  
 premium: {type: number}

### Database Schema Details

-- Applications Table  
CREATE TABLE insurance\_applications (  
 id UUID PRIMARY KEY,  
 applicant\_name VARCHAR(255) NOT NULL,  
 applicant\_age INTEGER NOT NULL,  
 medical\_history JSONB NOT NULL,  
 lifestyle\_factors JSONB NOT NULL,  
 risk\_score FLOAT,  
 premium\_amount DECIMAL(10,2),  
 status VARCHAR(50) NOT NULL,  
 created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,  
 updated\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP  
);  
  
-- Risk Scores Table  
CREATE TABLE risk\_scores (  
 id UUID PRIMARY KEY,  
 application\_id UUID REFERENCES insurance\_applications(id),  
 overall\_score FLOAT NOT NULL,  
 medical\_factor FLOAT NOT NULL,  
 age\_factor FLOAT NOT NULL,  
 lifestyle\_factor FLOAT NOT NULL,  
 confidence\_score FLOAT NOT NULL,  
 created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP  
);  
  
-- Vector Store Schema  
CREATE TABLE medical\_conditions (  
 id UUID PRIMARY KEY,  
 condition\_name VARCHAR(255) NOT NULL,  
 embedding VECTOR(1536) NOT NULL,  
 risk\_factor FLOAT NOT NULL,  
 metadata JSONB  
);

### AI Model Specifications

# LLM Service Configuration  
class LLMConfig:  
 MODEL\_NAME = "gpt-4"  
 MAX\_TOKENS = 4000  
 TEMPERATURE = 0.7  
 TOP\_P = 0.9  
 FREQUENCY\_PENALTY = 0.5  
 PRESENCE\_PENALTY = 0.5  
  
# Vector Store Configuration  
class VectorStoreConfig:  
 DIMENSION = 1536  
 DISTANCE\_METRIC = "cosine"  
 INDEX\_TYPE = "HNSW"  
 EF\_CONSTRUCTION = 200  
 M = 16  
  
# Risk Assessment Model  
class RiskAssessmentModel:  
 def \_\_init\_\_(self):  
 self.medical\_weights = {  
 "chronic\_conditions": 0.4,  
 "acute\_conditions": 0.3,  
 "medications": 0.2,  
 "family\_history": 0.1  
 }  
   
 self.lifestyle\_weights = {  
 "smoking": 0.3,  
 "alcohol": 0.2,  
 "exercise": 0.2,  
 "diet": 0.2,  
 "stress": 0.1  
 }

### Performance Specifications

# Performance Benchmarks  
class PerformanceMetrics:  
 # API Response Times  
 API\_RESPONSE\_TIME = {  
 "p95": 200, # milliseconds  
 "p99": 500, # milliseconds  
 "max": 1000 # milliseconds  
 }  
   
 # Database Query Times  
 DB\_QUERY\_TIME = {  
 "simple": 50, # milliseconds  
 "complex": 200, # milliseconds  
 "batch": 500 # milliseconds  
 }  
   
 # AI Processing Times  
 AI\_PROCESSING\_TIME = {  
 "risk\_assessment": 5000, # milliseconds  
 "premium\_calculation": 2000, # milliseconds  
 "document\_processing": 3000 # milliseconds  
 }

### Security Specifications

# Security Configuration  
class SecurityConfig:  
 # JWT Configuration  
 JWT\_CONFIG = {  
 "algorithm": "HS256",  
 "access\_token\_expire\_minutes": 60,  
 "refresh\_token\_expire\_days": 30,  
 "secret\_key": "your-secret-key" # Should be in environment variables  
 }  
   
 # Password Policy  
 PASSWORD\_POLICY = {  
 "min\_length": 12,  
 "require\_uppercase": True,  
 "require\_lowercase": True,  
 "require\_numbers": True,  
 "require\_special\_chars": True,  
 "max\_age\_days": 90  
 }  
   
 # API Security  
 API\_SECURITY = {  
 "rate\_limit": 100, # requests per minute  
 "max\_retries": 3,  
 "timeout": 30 # seconds  
 }

### Monitoring Specifications

class InsuranceMonitor:  
 CRITICAL\_METRICS = {  
 "claim\_approval\_rate": (0.85, 0.95), # acceptable range  
 "premium\_accuracy": 0.99, # minimum required  
 "risk\_assessment\_time": 5.0, # seconds (max)  
 "policy\_issuance\_time": 30.0 # seconds (max)  
 }  
  
 def check\_underwriting\_compliance(self):  
 return {  
 "age\_limits": self.validate\_age\_ranges(),  
 "risk\_thresholds": self.check\_risk\_distribution(),  
 "coverage\_limits": self.verify\_coverage\_application()  
 }

### Insurance Data Models

# Medical Condition Classification  
class MedicalCondition:  
 CATEGORIES = {  
 "chronic": {  
 "diabetes": {"risk\_factor": 0.4, "severity": "high"},  
 "hypertension": {"risk\_factor": 0.3, "severity": "medium"},  
 "asthma": {"risk\_factor": 0.2, "severity": "low"}  
 },  
 "acute": {  
 "heart\_attack": {"risk\_factor": 0.5, "severity": "high"},  
 "stroke": {"risk\_factor": 0.5, "severity": "high"},  
 "fracture": {"risk\_factor": 0.1, "severity": "low"}  
 }  
 }  
  
# Coverage Types  
class CoverageType:  
 TYPES = {  
 "basic": {  
 "coverage\_limit": 100000,  
 "deductible": 1000,  
 "co\_payment": 0.2  
 },  
 "premium": {  
 "coverage\_limit": 500000,  
 "deductible": 500,  
 "co\_payment": 0.1  
 },  
 "comprehensive": {  
 "coverage\_limit": 1000000,  
 "deductible": 0,  
 "co\_payment": 0.05  
 }  
 }  
  
# Risk Factor Weightings  
class RiskWeights:  
 FACTORS = {  
 "medical": {  
 "chronic\_conditions": 0.4,  
 "acute\_conditions": 0.3,  
 "family\_history": 0.2,  
 "medications": 0.1  
 },  
 "lifestyle": {  
 "smoking": 0.3,  
 "alcohol": 0.2,  
 "exercise": 0.2,  
 "diet": 0.2,  
 "stress": 0.1  
 },  
 "occupation": {  
 "hazardous": 0.4,  
 "sedentary": 0.1,  
 "moderate": 0.2  
 }  
 }

### Insurance Processing Logic

# Underwriting Rules  
class UnderwritingRules:  
 RULES = {  
 "age\_limits": {  
 "min": 18,  
 "max": 75  
 },  
 "risk\_thresholds": {  
 "low": 0.3,  
 "medium": 0.6,  
 "high": 0.8  
 },  
 "coverage\_limits": {  
 "basic": 100000,  
 "premium": 500000,  
 "comprehensive": 1000000  
 }  
 }  
  
# Premium Calculation  
class PremiumCalculator:  
 def calculate\_premium(self, risk\_score: float, coverage\_type: str, age: int) -> float:  
 # Base premium based on coverage type  
 base\_premium = self.get\_base\_premium(coverage\_type)  
   
 # Risk multiplier  
 risk\_multiplier = self.calculate\_risk\_multiplier(risk\_score)  
   
 # Age factor  
 age\_factor = self.calculate\_age\_factor(age)  
   
 # Final premium calculation  
 return base\_premium \* risk\_multiplier \* age\_factor  
  
 def calculate\_risk\_multiplier(self, risk\_score: float) -> float:  
 if risk\_score < 0.3:  
 return 1.0  
 elif risk\_score < 0.6:  
 return 1.5  
 elif risk\_score < 0.8:  
 return 2.0  
 else:  
 return 3.0  
  
# Claim Processing  
class ClaimProcessor:  
 def process\_claim(self, claim\_data: Dict[str, Any]) -> Dict[str, Any]:  
 # Validate claim  
 if not self.validate\_claim(claim\_data):  
 return {"status": "rejected", "reason": "invalid\_claim"}  
   
 # Check coverage  
 coverage = self.check\_coverage(claim\_data)  
 if not coverage["covered"]:  
 return {"status": "rejected", "reason": coverage["reason"]}  
   
 # Calculate payout  
 payout = self.calculate\_payout(claim\_data)  
   
 return {  
 "status": "approved",  
 "payout\_amount": payout,  
 "processing\_time": datetime.now().isoformat()  
 }

### Insurance Compliance

# Regulatory Requirements  
class RegulatoryCompliance:  
 REQUIREMENTS = {  
 "documentation": {  
 "application\_form": True,  
 "medical\_history": True,  
 "identity\_proof": True,  
 "income\_proof": True  
 },  
 "timelines": {  
 "application\_processing": "48h",  
 "claim\_processing": "15d",  
 "premium\_payment": "30d",  
 "policy\_renewal": "30d"  
 },  
 "notifications": {  
 "policy\_issuance": True,  
 "premium\_due": True,  
 "claim\_status": True,  
 "policy\_renewal": True  
 }  
 }  
  
# Policy Documentation  
class PolicyDocumentation:  
 REQUIRED\_DOCUMENTS = {  
 "application": [  
 "identity\_proof",  
 "address\_proof",  
 "income\_proof",  
 "medical\_reports"  
 ],  
 "claim": [  
 "claim\_form",  
 "medical\_bills",  
 "hospital\_discharge",  
 "doctor\_prescription"  
 ]  
 }  
  
# Customer Communication  
class CustomerCommunication:  
 NOTIFICATIONS = {  
 "policy\_issuance": {  
 "channel": ["email", "sms"],  
 "template": "policy\_issuance\_template",  
 "timing": "immediate"  
 },  
 "premium\_due": {  
 "channel": ["email", "sms"],  
 "template": "premium\_due\_template",  
 "timing": "7d\_before"  
 },  
 "claim\_status": {  
 "channel": ["email", "sms"],  
 "template": "claim\_status\_template",  
 "timing": "status\_change"  
 }  
 }

### Age Factor Calculation

def calculate\_age\_factor(age: int) -> float: if 18 <= age <= 30: return 1.0 elif 31 <= age <= 45: return 1.2 elif 46 <= age <= 60: return 1.5 else: return 2.0

### Risk Score Composition

RISK\_WEIGHTS = { “medical\_history”: 0.4, “lifestyle”: 0.3, “occupation”: 0.2, “family\_history”: 0.1 }

### Premium Adjustment Factors

PREMIUM\_ADJUSTMENTS = { “non\_smoker\_discount”: 0.15, “annual\_payment\_discount”: 0.05, “high\_risk\_occupation\_surcharge”: 0.25, “pre\_existing\_condition\_surcharge”: 0.35 }

class PolicyManager: POLICY\_STATES = [“active”, “lapsed”, “surrendered”, “claimed”]

def calculate\_surrender\_value(self, policy\_duration: int, premiums\_paid: float) -> float:  
 if policy\_duration < 1: return 0.0  
 return premiums\_paid \* min(0.7, 0.3 + (policy\_duration \* 0.05))  
  
def handle\_renewal(self, policy: Policy) -> Policy:  
 if policy.claims\_history:  
 new\_premium = policy.premium \* 1.15  
 else:  
 new\_premium = policy.premium \* 0.98  
 return policy.update(premium=new\_premium)

class ClaimsEngine: CLAIM\_VALIDATION\_RULES = { “submission\_deadline”: 90, # days “document\_verification”: [“hospital\_bills”, “police\_report”], “fraud\_indicators”: { “late\_reporting”: 0.3, “discrepant\_docs”: 0.4, “history\_of\_claims”: 0.3 } }

def assess\_claim\_legitimacy(self, claim: Claim) -> float:  
 fraud\_score = sum(  
 self.CLAIM\_VALIDATION\_RULES["fraud\_indicators"][indicator]   
 for indicator in claim.red\_flags  
 )  
 return min(1.0, fraud\_score)

### Policy Management Database Schema

-- Policy Table  
CREATE TABLE insurance\_policies (  
 id UUID PRIMARY KEY,  
 application\_id UUID REFERENCES insurance\_applications(id),  
 policy\_number VARCHAR(20) NOT NULL UNIQUE,  
 status VARCHAR(50) NOT NULL, -- active, lapsed, canceled, expired  
 coverage\_type VARCHAR(50) NOT NULL,  
 start\_date DATE NOT NULL,  
 end\_date DATE NOT NULL,  
 premium\_amount DECIMAL(10,2) NOT NULL,  
 payment\_frequency VARCHAR(20) NOT NULL, -- monthly, quarterly, annual  
 total\_coverage\_amount DECIMAL(15,2) NOT NULL,  
 created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,  
 updated\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP  
);  
  
-- Policy History Table  
CREATE TABLE policy\_history (  
 id UUID PRIMARY KEY,  
 policy\_id UUID REFERENCES insurance\_policies(id),  
 status\_from VARCHAR(50) NOT NULL,  
 status\_to VARCHAR(50) NOT NULL,  
 change\_date TIMESTAMP WITH TIME ZONE NOT NULL,  
 change\_reason VARCHAR(255) NOT NULL,  
 changed\_by UUID NOT NULL, -- reference to users  
 premium\_before DECIMAL(10,2),  
 premium\_after DECIMAL(10,2),  
 created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP  
);  
  
-- Claims Table  
CREATE TABLE insurance\_claims (  
 id UUID PRIMARY KEY,  
 policy\_id UUID REFERENCES insurance\_policies(id),  
 claim\_number VARCHAR(20) NOT NULL UNIQUE,  
 claim\_date DATE NOT NULL,  
 incident\_date DATE NOT NULL,  
 claim\_type VARCHAR(50) NOT NULL,  
 claim\_amount DECIMAL(15,2) NOT NULL,  
 status VARCHAR(50) NOT NULL, -- submitted, reviewing, approved, rejected, paid  
 description TEXT NOT NULL,  
 supporting\_documents JSONB,  
 fraud\_score FLOAT,  
 processing\_notes TEXT,  
 created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP,  
 updated\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP  
);  
  
-- Payments Table  
CREATE TABLE premium\_payments (  
 id UUID PRIMARY KEY,  
 policy\_id UUID REFERENCES insurance\_policies(id),  
 payment\_number VARCHAR(20) NOT NULL,  
 amount DECIMAL(10,2) NOT NULL,  
 payment\_date DATE NOT NULL,  
 due\_date DATE NOT NULL,  
 payment\_method VARCHAR(50) NOT NULL, -- credit\_card, direct\_debit, bank\_transfer  
 status VARCHAR(50) NOT NULL, -- pending, completed, failed, refunded  
 transaction\_id VARCHAR(100),  
 created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP  
);

### Claims API Endpoints

# Claims API  
/claims:  
 post:  
 summary: Submit a new claim  
 requestBody:  
 content:  
 application/json:  
 schema:  
 type: object  
 properties:  
 policy\_id: {type: string}  
 incident\_date: {type: string, format: date}  
 claim\_type: {type: string}  
 claim\_amount: {type: number}  
 description: {type: string}  
 supporting\_documents: {type: array}  
 responses:  
 201:  
 description: Claim submitted successfully  
 content:  
 application/json:  
 schema:  
 type: object  
 properties:  
 claim\_id: {type: string}  
 claim\_number: {type: string}  
 status: {type: string}  
 submission\_date: {type: string, format: date}  
  
 get:  
 summary: Get list of claims for a policy  
 parameters:  
 - name: policy\_id  
 in: query  
 required: true  
 schema:  
 type: string  
 responses:  
 200:  
 description: List of claims  
 content:  
 application/json:  
 schema:  
 type: array  
 items:  
 type: object  
 properties:  
 claim\_id: {type: string}  
 claim\_number: {type: string}  
 claim\_date: {type: string, format: date}  
 claim\_amount: {type: number}  
 status: {type: string}  
  
/claims/{claim\_id}:  
 get:  
 summary: Get claim details  
 parameters:  
 - name: claim\_id  
 in: path  
 required: true  
 schema:  
 type: string  
 responses:  
 200:  
 description: Claim details  
 content:  
 application/json:  
 schema:  
 type: object  
 properties:  
 claim\_id: {type: string}  
 policy\_id: {type: string}  
 claim\_number: {type: string}  
 incident\_date: {type: string, format: date}  
 claim\_type: {type: string}  
 claim\_amount: {type: number}  
 status: {type: string}  
 description: {type: string}  
 supporting\_documents: {type: array}  
 created\_at: {type: string, format: date-time}  
 updated\_at: {type: string, format: date-time}  
  
 patch:  
 summary: Update claim status  
 parameters:  
 - name: claim\_id  
 in: path  
 required: true  
 schema:  
 type: string  
 requestBody:  
 content:  
 application/json:  
 schema:  
 type: object  
 properties:  
 status: {type: string}  
 processing\_notes: {type: string}  
 responses:  
 200:  
 description: Claim updated successfully

### Policy Lifecycle State Machine

# Policy State Machine  
class PolicyStateMachine:  
 # Valid state transitions  
 STATE\_TRANSITIONS = {  
 "draft": ["active", "rejected"],  
 "active": ["lapsed", "canceled", "expired"],  
 "lapsed": ["active", "canceled"],  
 "canceled": [],  
 "expired": ["renewed"],  
 "renewed": ["active"],  
 "rejected": []  
 }  
   
 # Handlers for state transitions  
 TRANSITION\_HANDLERS = {  
 "draft->active": "activate\_policy",  
 "draft->rejected": "reject\_policy",  
 "active->lapsed": "lapse\_policy",  
 "active->canceled": "cancel\_policy",  
 "active->expired": "expire\_policy",  
 "lapsed->active": "reinstate\_policy",  
 "lapsed->canceled": "cancel\_lapsed\_policy",  
 "expired->renewed": "renew\_policy"  
 }  
   
 def transition\_policy(self, policy: Policy, to\_state: str) -> Policy:  
 """  
 Transition policy to a new state if the transition is valid  
 """  
 from\_state = policy.status  
   
 # Check if transition is valid  
 if to\_state not in self.STATE\_TRANSITIONS.get(from\_state, []):  
 raise InvalidStateTransitionError(  
 f"Cannot transition from {from\_state} to {to\_state}"  
 )  
   
 # Get transition handler  
 handler\_name = self.TRANSITION\_HANDLERS.get(f"{from\_state}->{to\_state}")  
 if not handler\_name:  
 raise MissingTransitionHandlerError(  
 f"No handler for transition from {from\_state} to {to\_state}"  
 )  
   
 # Execute transition handler  
 handler = getattr(self, handler\_name)  
 updated\_policy = handler(policy)  
   
 # Log the transition  
 self.log\_policy\_transition(policy.id, from\_state, to\_state)  
   
 return updated\_policy  
   
 def activate\_policy(self, policy: Policy) -> Policy:  
 """Activate a new policy"""  
 policy.status = "active"  
 policy.start\_date = datetime.now().date()  
 policy.end\_date = policy.start\_date + relativedelta(years=1)  
 return policy  
   
 def lapse\_policy(self, policy: Policy) -> Policy:  
 """Mark policy as lapsed due to non-payment"""  
 policy.status = "lapsed"  
 # Additional lapse logic  
 return policy  
   
 def reinstate\_policy(self, policy: Policy) -> Policy:  
 """Reinstate a lapsed policy after payment"""  
 policy.status = "active"  
 # Recalculate end date if needed  
 return policy

### Payment Processing Integration

# Payment Gateway Interface  
class PaymentGateway:  
 def \_\_init\_\_(self, config: Dict[str, Any]):  
 self.api\_key = config["api\_key"]  
 self.merchant\_id = config["merchant\_id"]  
 self.base\_url = config["base\_url"]  
 self.timeout = config.get("timeout", 30)  
   
 async def process\_payment(self, payment\_data: Dict[str, Any]) -> Dict[str, Any]:  
 """  
 Process a payment through the payment gateway  
 """  
 payload = {  
 "merchant\_id": self.merchant\_id,  
 "amount": payment\_data["amount"],  
 "currency": payment\_data["currency"],  
 "payment\_method": {  
 "type": payment\_data["payment\_method"]["type"],  
 "details": payment\_data["payment\_method"]["details"]  
 },  
 "description": payment\_data["description"],  
 "customer": {  
 "email": payment\_data["customer"]["email"],  
 "name": payment\_data["customer"]["name"]  
 },  
 "metadata": {  
 "policy\_id": payment\_data["metadata"]["policy\_id"],  
 "payment\_number": payment\_data["metadata"]["payment\_number"]  
 }  
 }  
   
 headers = {  
 "Authorization": f"Bearer {self.api\_key}",  
 "Content-Type": "application/json"  
 }  
   
 try:  
 async with httpx.AsyncClient(timeout=self.timeout) as client:  
 response = await client.post(  
 f"{self.base\_url}/v1/payments",  
 json=payload,  
 headers=headers  
 )  
 response.raise\_for\_status()  
 return response.json()  
 except httpx.HTTPStatusError as e:  
 logger.error(f"Payment gateway error: {str(e)}")  
 raise PaymentProcessingError(f"Payment failed: {str(e)}")  
 except httpx.RequestError as e:  
 logger.error(f"Payment gateway request error: {str(e)}")  
 raise PaymentGatewayConnectionError(f"Failed to connect to payment gateway: {str(e)}")  
  
# Payment Service  
class PaymentService:  
 def \_\_init\_\_(self, db: Database, payment\_gateway: PaymentGateway):  
 self.db = db  
 self.payment\_gateway = payment\_gateway  
  
 async def create\_payment(self, policy\_id: str, amount: float, due\_date: date) -> Dict[str, Any]:  
 """  
 Create a new payment record  
 """  
 # Get policy details  
 policy = await self.db.get\_policy(policy\_id)  
   
 # Generate payment number  
 payment\_number = f"PMT-{uuid.uuid4().hex[:8].upper()}"  
   
 # Create payment record  
 payment = {  
 "id": str(uuid.uuid4()),  
 "policy\_id": policy\_id,  
 "payment\_number": payment\_number,  
 "amount": amount,  
 "payment\_date": None,  
 "due\_date": due\_date,  
 "payment\_method": policy.payment\_method,  
 "status": "pending",  
 "created\_at": datetime.now()  
 }  
   
 # Store in database  
 await self.db.create\_payment(payment)  
   
 return payment  
  
 async def process\_scheduled\_payment(self, payment\_id: str) -> Dict[str, Any]:  
 """  
 Process a scheduled payment  
 """  
 # Get payment details  
 payment = await self.db.get\_payment(payment\_id)  
 policy = await self.db.get\_policy(payment["policy\_id"])  
 customer = await self.db.get\_customer(policy.customer\_id)  
   
 # Prepare payment data  
 payment\_data = {  
 "amount": payment["amount"],  
 "currency": "USD",  
 "payment\_method": {  
 "type": payment["payment\_method"],  
 "details": await self.get\_payment\_method\_details(policy.customer\_id, payment["payment\_method"])  
 },  
 "description": f"Premium payment {payment['payment\_number']} for policy {policy.policy\_number}",  
 "customer": {  
 "email": customer.email,  
 "name": customer.full\_name  
 },  
 "metadata": {  
 "policy\_id": payment["policy\_id"],  
 "payment\_number": payment["payment\_number"]  
 }  
 }  
   
 # Process payment through gateway  
 try:  
 result = await self.payment\_gateway.process\_payment(payment\_data)  
   
 # Update payment record  
 payment["status"] = "completed"  
 payment["payment\_date"] = datetime.now().date()  
 payment["transaction\_id"] = result["transaction\_id"]  
 await self.db.update\_payment(payment\_id, payment)  
   
 # Check if this was a reinstatement payment  
 if policy.status == "lapsed":  
 # Reinstate policy  
 policy\_service = PolicyService(self.db)  
 await policy\_service.reinstate\_policy(policy.id)  
   
 return payment  
 except (PaymentProcessingError, PaymentGatewayConnectionError) as e:  
 # Update payment record with failure  
 payment["status"] = "failed"  
 await self.db.update\_payment(payment\_id, payment)  
   
 # Log failure  
 logger.error(f"Payment {payment\_id} failed: {str(e)}")  
   
 # Check if policy should be lapsed  
 if policy.status == "active":  
 # Get count of recent failed payments  
 recent\_failed\_count = await self.db.count\_recent\_failed\_payments(  
 policy\_id=policy.id,  
 days=30  
 )  
   
 # Lapse policy if multiple failures  
 if recent\_failed\_count >= 2:  
 policy\_service = PolicyService(self.db)  
 await policy\_service.lapse\_policy(policy.id, reason="payment\_failure")  
   
 raise

### Document Management Implementation

# Document Storage Service  
class DocumentService:  
 def \_\_init\_\_(self, storage\_client, config: Dict[str, Any]):  
 self.storage\_client = storage\_client  
 self.bucket\_name = config["bucket\_name"]  
 self.expiration\_time = config["signed\_url\_expiration"]  
 self.allowed\_extensions = config["allowed\_extensions"]  
 self.max\_file\_size = config["max\_file\_size"]  
   
 async def upload\_document(self, document\_data: Dict[str, Any], file: UploadFile) -> Dict[str, Any]:  
 """  
 Upload a document to the storage service  
 """  
 # Validate file extension  
 file\_extension = self.\_get\_file\_extension(file.filename)  
 if file\_extension not in self.allowed\_extensions:  
 raise InvalidFileTypeError(f"File type {file\_extension} not allowed")  
   
 # Validate file size  
 content = await file.read()  
 if len(content) > self.max\_file\_size:  
 raise FileTooLargeError(f"File size exceeds maximum of {self.max\_file\_size} bytes")  
   
 # Generate document ID and path  
 document\_id = str(uuid.uuid4())  
 entity\_type = document\_data["entity\_type"] # policy, claim, application  
 entity\_id = document\_data["entity\_id"]  
 document\_type = document\_data["document\_type"]  
 storage\_path = f"{entity\_type}/{entity\_id}/{document\_type}/{document\_id}.{file\_extension}"  
   
 # Upload to storage  
 blob = self.storage\_client.bucket(self.bucket\_name).blob(storage\_path)  
 blob.upload\_from\_string(content, content\_type=file.content\_type)  
   
 # Create document record  
 document = {  
 "id": document\_id,  
 "entity\_type": entity\_type,  
 "entity\_id": entity\_id,  
 "document\_type": document\_type,  
 "filename": file.filename,  
 "storage\_path": storage\_path,  
 "content\_type": file.content\_type,  
 "size\_bytes": len(content),  
 "upload\_date": datetime.now(),  
 "uploaded\_by": document\_data["uploaded\_by"],  
 "metadata": document\_data.get("metadata", {})  
 }  
   
 return document  
   
 async def get\_document\_url(self, document\_id: str) -> str:  
 """  
 Get a temporary signed URL for accessing a document  
 """  
 document = await self.db.get\_document(document\_id)  
 blob = self.storage\_client.bucket(self.bucket\_name).blob(document["storage\_path"])  
   
 # Generate signed URL  
 url = blob.generate\_signed\_url(  
 version="v4",  
 expiration=datetime.timedelta(seconds=self.expiration\_time),  
 method="GET"  
 )  
   
 return url  
   
 async def delete\_document(self, document\_id: str) -> None:  
 """  
 Delete a document  
 """  
 document = await self.db.get\_document(document\_id)  
   
 # Delete from storage  
 blob = self.storage\_client.bucket(self.bucket\_name).blob(document["storage\_path"])  
 blob.delete()  
   
 # Delete document record  
 await self.db.delete\_document(document\_id)  
   
 def \_get\_file\_extension(self, filename: str) -> str:  
 """Get the file extension from a filename"""  
 return filename.rsplit(".", 1)[1].lower() if "." in filename else ""  
  
# Document Validation Service  
class DocumentValidator:  
 def \_\_init\_\_(self, config: Dict[str, Any]):  
 self.required\_documents = config["required\_documents"]  
 self.document\_service = DocumentService(  
 storage\_client=config["storage\_client"],  
 config=config["storage\_config"]  
 )  
   
 async def validate\_documents(self, entity\_type: str, entity\_id: str) -> Dict[str, Any]:  
 """  
 Validate that all required documents are present  
 """  
 # Get required documents for entity type  
 required\_doc\_types = self.required\_documents.get(entity\_type, [])  
   
 # Get existing documents  
 existing\_docs = await self.db.get\_documents\_by\_entity(entity\_type, entity\_id)  
   
 # Group by document type  
 docs\_by\_type = {}  
 for doc in existing\_docs:  
 doc\_type = doc["document\_type"]  
 if doc\_type not in docs\_by\_type:  
 docs\_by\_type[doc\_type] = []  
 docs\_by\_type[doc\_type].append(doc)  
   
 # Check for missing documents  
 missing\_docs = []  
 for doc\_type in required\_doc\_types:  
 if doc\_type not in docs\_by\_type or not docs\_by\_type[doc\_type]:  
 missing\_docs.append(doc\_type)  
   
 return {  
 "is\_complete": len(missing\_docs) == 0,  
 "missing\_documents": missing\_docs,  
 "existing\_documents": docs\_by\_type  
 }

### Batch Processing Jobs

```python # Policy Renewal Job class PolicyRenewalJob: def **init**(self, db: Database, notification\_service: NotificationService): self.db = db self.notification\_service = notification\_service

async def run(self):  
 """  
 Process policies due for renewal  
 """  
 # Get policies expiring in the next 30 days  
 expiring\_soon = await self.db.get\_expiring\_policies(days=30)  
   
 # Group by days until expiration  
 by\_days = {  
 30: [],  
 14: [],  
 7: []  
 }  
   
 today = datetime.now().date()  
 for policy in expiring\_soon:  
 days\_until = (policy.end\_date - today).days  
 if days\_until <= 7:  
 by\_days[7].append(policy)  
 elif days\_until <= 14:  
 by\_days[14].append(policy)  
 elif days\_until <= 30:  
 by\_days[30].append(policy)  
   
 # Send renewal notifications  
 for days, policies in by\_days.items():  
 for policy in policies:  
 customer = await self.db.get\_customer(policy.customer\_id)  
   
 # Skip if notification already sent  
 if await self.db.check\_notification\_sent(  
 policy\_id=policy.id,  
 type=f"renewal\_{days}\_days",  
 days=days  
 ):  
 continue  
   
 # Calculate renewal premium  
 new\_premium = await self.calculate\_renewal\_premium(policy)  
   
 # Send notification  
 await self.notification\_service.send\_notification(  
 recipient=customer.email,  
 template="policy\_renewal",  
 data={  
 "customer\_name": customer.full\_name,  
 "policy\_number": policy.policy\_number,  
 "expiry\_date": policy.end\_date.isoformat(),  
 "days\_remaining": days,  
 "current\_premium": policy.premium\_amount,  
 "renewal\_premium": new\_premium,  
 "renewal\_link": f"/policies/{policy.id}/renew"  
 }  
 )  
   
 # Record notification  
 await self.db.save\_notification(  
 policy\_id=policy.id,  
 customer\_id=customer.id,  
 type=f"renewal\_{days}\_days",  
 sent\_at=datetime.now()  
 )  
  
async def calculate\_renewal\_premium(self, policy: Policy) -> float:  
 """  
 Calculate the premium for policy renewal  
 """  
 # Get claim history  
 claims = await self.db.get\_policy\_claims(policy.id)  
   
 # Base renewal is current premium  
 base\_renewal = policy.premium\_amount  
   
 # Apply claim history adjustment  
 if claims:  
 # 15% increase for policies with claims  
 base\_renewal \*= 1.15  
 else:  
 # 5% loyalty discount for policies without claims  
 base\_renewal \*= 0.95  
   
 # Apply inflation adjustment (3% per year)  
 base\_renewal \*= 1.03  
   
 return round(base\_renewal, 2)

# Payment Reminder Job

class PaymentReminderJob: def **init**(self, db: Database, notification\_service: NotificationService): self.db = db self.notification\_service = notification\_service

async def run(self):  
 """  
 Send reminders for upcoming and overdue payments  
 """  
 today = datetime.now().date()  
   
 # Get payments due in the next 7 days  
 upcoming\_payments = await self.db.get\_upcoming\_payments(days=7)  
   
 # Get overdue payments  
 overdue\_payments = await self.db.get\_overdue\_payments()  
   
 # Process upcoming payment reminders  
 for payment in upcoming\_payments:  
 policy = await self.db.get\_policy(payment.policy\_id)  
 customer = await self.db.get\_customer(policy.customer\_id)  
   
 days\_until\_due = (payment.due\_date - today).days  
   
 # Skip if reminder already sent for this timeframe  
 if await self.db.check\_notification\_sent(  
 payment\_id=payment.id,  
 type=f"payment\_due\_{days\_until\_due}\_days",  
 days=7  
 ):  
 continue  
   
 # Send notification  
 await self.notification\_service.send\_notification(  
 recipient=customer.email,  
 template="payment\_reminder",  
 data={  
 "customer\_name": customer.full\_name,  
 "policy\_number": policy.policy\_number,  
 "payment\_amount": payment.amount,  
 "due\_date": payment.due\_date.isoformat(),  
 "days\_until\_due": days\_until\_due,  
 "payment\_link": f"/payments/{payment.id}/pay"  
 }  
 )  
   
 # Record notification  
 await self.db.save\_notification(  
 policy\_id=policy.id,  
 payment\_id=payment.id,  
 customer\_id=customer.id,  
 type=f"payment\_due\_{days\_until\_due}\_days",  
 sent\_at=datetime.now()  
 )  
   
 # Process overdue payment reminders  
 for payment in overdue\_payments:  
 policy = await self.db.get\_policy(payment.policy\_id)  
 customer = await self.db.get\_customer(policy.customer\_id)  
   
 days\_overdue = (today - payment.due\_date).days  
   
 # Handle different overdue periods  
 if days\_overdue >= 30:  
 reminder\_type = "payment\_overdue\_30\_days"  
 elif days\_overdue >= 14:  
 reminder\_type = "payment\_overdue\_14\_days"  
 elif days\_overdue >= 7:  
 reminder\_type = "payment\_overdue\_7\_days"  
 else:  
 reminder\_type = "payment\_overdue"  
   
 # Skip if reminder already sent for this timeframe  
 if await self.db.check\_notification\_sent(  
 payment\_id=payment.id,  
 type=reminder\_type,  
 days=7  
 ):  
 continue  
   
 # Check if policy already lapsed  
 if policy.status == "lapsed":  
 template = "payment\_overdue\_policy\_lapsed"  
 else:  
 template = "payment\_overdue"  
   
 # Lapse policy if overdue for more than 30 days  
 if days\_overdue >= 30 and policy.status == "active":  
 policy\_service = PolicyService(self.db)  
 await policy\_service.lapse\_policy(policy.id, reason="non\_payment")  
   
 # Send notification  
 await self.notification\_service.send\_notification(  
 recipient=customer.email,  
 template=template,  
 data={  
 "customer\_name": customer.full\_name,  
 "policy\_number": policy.policy\_number,  
 "payment\_amount": payment.amount,  
 "due\_date": payment.due\_date.isoformat(),  
 "days\_overdue": days\_overdue,  
 "payment\_link": f"/payments/{payment.id}/pay",  
 "policy\_status": policy.status  
 }  
 )  
   
 # Record notification  
 await self.db.save\_notification(  
 policy\_id=policy.id,  
 payment\_id=payment.id,  
 customer\_id=customer.id,  
 type=reminder\_type,  
 sent\_at=datetime.now()  
 )