TOBB University of Economics and Technology

Department of Computer Engineering

Department of Artificial Intelligence Engineering

AI Health System for Doctors and Patients

Low-Level Design Report

Abdullah Çöplü

Abdurrahman Doğru

Ece Derya

Zeynep Ergül

Table of Content

1. Introduction

1.1 Object design trade-offs

The object design of the **AI Health System for Doctors and Patients** balances extensibility, modularity, and performance under strict healthcare data protection and ethical constraints.
Key trade-offs include:

- **Modularity vs. Performance:** The system adopts a micro-modular design where each subsystem (User Management, Image Processing, Symptom Analysis, etc.) operates independently. Although this introduces minor communication overhead, it ensures scalability, parallel development, and maintainability.

- **Abstraction vs. Transparency:** Abstraction layers are used to isolate AI model execution and data persistence from business logic. While this increases implementation complexity, it provides flexibility for future model updates and storage migration.

- **Security vs. Usability:** Strong authentication and encryption mechanisms slightly increase latency, but they are essential to maintain compliance with **GDPR**, **KVKK**, and **HIPAA**.

- **AI Accuracy vs. Explainability:** Deep learning models are optimized for diagnostic accuracy, but additional post-processing and visualization components are added to enhance interpretability and ensure ethical use.

Overall, the object design prioritizes **safety, maintainability, and compliance** over raw performance, aligning with the ethical and professional requirements of medical software systems.

1.2 Interface documentation guidelines

All subsystem interfaces are designed following clear and traceable documentation standards to ensure consistent integration across development teams.
Each interface must specify:

- **Input and output data types** (DTOs, JSON structures, or database entities)
- **Preconditions and postconditions** for method execution
- **Error and exception handling behavior**
- **Versioning and backward compatibility notes**

UML **Sequence Diagrams** and **Activity Diagrams** are provided to illustrate control flow across modules such as login authentication, image upload, and inference workflows.
For each interface, documentation includes both textual API definitions and corresponding UML representations to maintain design–implementation traceability.

1.3 Engineering standards (e.g., UML and IEEE)

The design and documentation of this system follow widely accepted **software engineering standards**:

- **UML (Unified Modeling Language):** Used to express class, sequence, and activity diagrams in accordance with **UML 2.5** notation.
- **IEEE 1016–2009:** Followed as the reference for software design description (SDD) structure and terminology.
- **IEEE 830–1998:** Referenced for maintaining consistency with software requirement specifications (SRS).
- **PEP 8 (Python Enhancement Proposal):** Adopted as the coding style standard for all backend modules.
- **ISO/IEC 25010:** Considered for defining system quality attributes such as reliability, performance efficiency, and security.

These standards ensure that the design is both **professionally structured and implementation-ready**, promoting collaboration and long-term maintainability.

1.4 Definitions, acronyms, and abbreviations

Term	Definition
AI	Artificial Intelligence
LLM	Large Language Model
CNN	Convolutional Neural Network
DTO	Data Transfer Object
RBAC	Role-Based Access Control
REST API	Representational State Transfer Application Programming Interface
HLD	High Level Design
LLD	Low Level Design

GDPR	General Data Protection Regulation
KVKK	Kişisel Verilerin Korunması Kanunu (Turkish Personal Data Protection Law)
HIPAA	Health Insurance Portability and Accountability Act
UML	Unified Modeling Language
IEEE	Institute of Electrical and Electronics Engineers

2. Packages

2.1. Server

2.1.1. Model

- ***User Management***

  User: This class represents a system user, either a doctor or a patient. It stores core identity information, authentication credentials, and role metadata. It serves as the foundation of role-based access control (RBAC) and secure access management across all system modules.

  SessionToken: This class represents an active authentication session associated with a user. It maintains secure login sessions using time-bound tokens, enabling authorized API access while preventing replay attacks and unauthorized reuse.

  Role (Enum): Represents the predefined access privilege levels within the system to ensure consistent authorization control. Values: DOCTOR - PATIENT

  LoginRequest: This class represents the data transfer object used during authentication requests. It carries user login credentials from the client to the authentication service for validation.

  LoginResponse: This class represents the authentication response returned after successful login. It contains the issued session token, the authenticated user role, and the token expiration duration.

  PasswordPolicy: This class represents a value object that defines standardized password validation constraints. It enforces complexity requirements such as minimum length, special characters, uppercase letters, and numeric inclusion to maintain credential security consistency.

- ***Image Processing***

MedicalImage: This class represents a diagnostic medical image uploaded by a doctor within the system. It serves as the core entity for image-based AI inference workflows and maintains the association between the uploaded file, the responsible physician, and the related patient case.

AIResult: This class represents the AI-generated diagnostic output derived from a medical image. It stores the predicted label, confidence score, lifecycle status, and creation timestamp, enabling physician review, auditing, and potential override operations.

ResultStatus (Enum): Represents the lifecycle state of an AI inference result to ensure controlled management of prediction validity. Values: ACTIVE - OVERRIDDEN

ImageUploadRequest: This class represents the data transfer object used during image upload operations. It carries the uploader identity, associated case identifier, and file metadata required for server-side validation and persistent storage.

InferenceResponse: This class represents the structured response returned to the doctor dashboard after AI inference is completed. It contains the prediction result, confidence score, model version information, and the current result status.

ModelVersion: This class represents a deployed AI model version used for medical image inference. It stores architectural details, release information, performance metrics, and notes to ensure full traceability between predictions and specific model deployments.

- ***Symptom Analysis***

SymptomReport: This class represents a patient-submitted symptom report captured as free-text input. It stores the patient identifier, optional linkage to a clinical case, the original symptom text, and the language context required for prompt generation and downstream safety processing.

SymptomAnalysisResult: This class represents the finalized, patient-safe output produced by the symptom analysis pipeline. It contains the computed risk level, the sanitized and simplified safe text, a safe list of possible conditions, the moderation action applied to the generated content, and the disclaimer text that must accompany all patient-facing outputs.

ConsentRecord: This class represents a persisted consent tracking record used to verify whether a user has viewed and/or accepted the required consent version. It stores the user identity, consent version, language, timestamps for viewed/accepted actions, and optional IP address metadata for auditability.

RiskLevel (Enum): Represents the standardized risk classification returned by symptom analysis to ensure consistent triage communication in patient-facing results. Values: LOW - MEDIUM - HIGH

ModerationAction (Enum): Represents the moderation enforcement decision applied to patient-facing content after safety checks. It determines whether the content can be returned as-is, must be

rewritten, or must be blocked. Values: ALLOW - REWRITE - BLOCK

- ***Feedback and Moderation***

FeedbackEntry: This class represents a structured feedback record submitted by a doctor for a specific AI output. It links the feedback to the related clinical case and the target AI result, enabling the system to capture physician feedback for traceability and quality improvement.

DoctorAnnotation: This class represents a doctor-written note attached to a specific target within a case (such as an image result or symptom analysis). It is used to store additional clinical context and clarifications that support review and documentation.

DoctorOverride: This class represents an explicit doctor override action applied to an AI-generated result. It provides a controlled mechanism for physicians to correct an AI label while keeping a traceable record of the change.

FeedbackType: This type defines standardized categories for feedback so that doctor responses can be consistently stored and analyzed instead of relying only on free-text.

- ***Monitoring and Logging***

AuditLog: This class represents an immutable audit event generated by the system for traceability. It captures key information about system actions such as access events, inference executions, and feedback-related actions to support compliance review and incident investigation.

- ***Localization and UI***

Language (Enum): Represents the supported languages of the system. This enumeration defines the set of languages that can be selected for user interface rendering and backend response localization. It is used consistently across localization services, UI components, and language negotiation logic to ensure type-safe language handling within the system. Currently, the system supports Turkish (TR) and English (EN).

CoverageReport (DTO): This class is responsible for managing multilingual UI text resources for the entire system. It maintains language-specific translation bundles and provides key-based translation lookup functionality used by both frontend UI rendering and backend-generated responses. In addition to translation retrieval, the service ensures that all required UI keys are defined for each supported language by performing translation coverage validation. This mechanism guarantees 100% localization completeness and prevents missing-text issues in production environments.

- ***Training and Model Development***

DatasetVersion (Entity): Represents a versioned dataset used for training machine learning models. This class ensures reproducibility and traceability by storing dataset metadata, integrity hash values, source information, and dataset split configurations (train/validation/test). Each dataset version acts as a fixed reference point in the training pipeline, enabling auditability and controlled experimentation within the medical AI system.

TrainingJob (Entity): Represents a single training workflow instance within the system. It manages the lifecycle of a model training execution, including its task type, associated dataset version, hyperparameters, execution timestamps, and current status. This class enables tracking of training processes from queued state through execution to completion or failure, ensuring controlled orchestration and reproducibility of model development activities.

ModelArtifact (Entity): Represents a trained model produced by a training job. It encapsulates model metadata such as version identifier, framework information, artifact storage location, and performance metrics obtained during evaluation. This class acts as the persistent representation of deployable machine learning models and supports safe promotion to production environments after validation.

TrainingTaskType (Enum): Defines the category of training operation being executed. It distinguishes between image-based model training, NLP-based model training, and moderation model training. This enumeration enables the system to route training jobs to appropriate trainer components in a type-safe and structured manner. Values: IMAGE, NLP, MODERATION

TrainingStatus (Enum): Represents the lifecycle state of a training job. It is used to monitor and control execution flow within the training orchestration pipeline. Values: QUEUED, RUNNING, FAILED, SUCCEEDED

MetricsReport (DTO): Represents structured performance metrics generated during model evaluation. It encapsulates evaluation statistics required for performance validation and decision-making in the training pipeline.

ConfusionMatrix (Value Object): Represents the classification distribution results of a model evaluation, including true positives, false positives, true negatives, and false negatives. It serves as a foundational analytical structure for computing higher-level metrics such as accuracy and FNR.

ConstraintCheck (DTO): Represents the result of validating model performance against predefined safety and regulatory constraints. It indicates whether the evaluated model satisfies required thresholds and is eligible for further lifecycle progression.

2.1.2. Controller

- ***User Management***

AuthController: This class provides the API endpoints for user authentication and session management. It handles login, logout, and token refresh requests from both doctors and patients.

The controller validates credentials, issues secure tokens, and enforces access policies by delegating authentication logic to the corresponding service layer.

UserController: This class provides the API endpoints for managing user profiles and role-specific information. It enables users to retrieve or update their profile data and allows doctors to access their activity or upload history. The controller enforces role-based access control (RBAC) and delegates data handling to the user service.

- ***Image Processing***

ImageController: This class provides the API endpoints for uploading, validating, and managing medical images. It accepts image uploads from doctors, verifies file integrity and format, and stores metadata in the database. The controller triggers AI inference workflows by delegating execution to the image processing service.

ResultController: This class provides the API endpoints for retrieving, reviewing, and managing AI-generated diagnostic results. It allows doctors to view or override inference outputs, ensuring that updates are securely recorded for auditability. The controller delegates result handling and persistence tasks to the result service.

- ***Symptom Analysis***

PatientSymptomController: This class provides the API endpoints for the patient symptom analysis workflow, including consent tracking, symptom submission, and patient-safe result retrieval. It acts as the boundary layer that validates incoming requests, delegates consent state changes to the consent service, and triggers analysis execution through the symptom analysis service before returning a moderated and disclaimer-covered response.

- ***Feedback and Moderation***

FeedbackController: This class provides the API endpoints for doctors to submit structured feedback and to query previously submitted feedback records. It acts as the boundary layer that accepts requests and delegates processing to the feedback service.

AnnotationController: This class provides the API endpoints for adding doctor annotations to a case and for overriding AI results. It enables annotation and override workflows through the annotation service.

- ***Monitoring and Logging***

MonitoringController: This class provides operational endpoints used by monitoring tools to check system availability and basic service health. It enables visibility into uptime and dependency status.

AuditController: This class provides endpoints for querying audit records for traceability and compliance review. It supports retrieving audit information in a controlled way for administrators and investigative workflows.

- ***Localization and UI***

LocalizationController: Provides API endpoints for retrieving localized text and triggering coverage checks.

Responsibilities: Serve localized text responses, trigger translation coverage validation, provide diagnostic information for administrators

LanguageController: Handles language resolution and session language assignment.

Responsibilities: Resolve effective language, Store language in session context

- ***Training and Model Development***

TrainingOrchestrator: This class acts as the central coordinator for all training workflows. It manages job creation, execution, evaluation, logging, and artifact registration. In the absence of a separate "TrainingController" class in your allowed list, this class serves as the main server-side entry point for training operations.

2.1.3. Service

- ***User Management***

UserService: This class implements the business logic for managing user accounts, including registration, profile updates, and role-based access control. It validates input data, applies password policies, and ensures that sensitive information such as credentials is handled securely. The service also interacts with the persistence layer to create and update user entities.

AuthService: This class implements the core authentication logic for the system. It verifies user credentials, generates and refreshes tokens, and ensures that authentication processes comply with the system's security standards. It interacts with UserService for credential validation and SessionService for session tracking.

SessionService: This class manages the lifecycle of active user sessions. It creates, validates, refreshes, and terminates session tokens. The service ensures that expired or invalid tokens cannot be reused and that all session-related actions are logged for traceability.

PasswordService: This class provides password-related utilities, including hashing, verification, and strength validation. It enforces password complexity rules defined in the PasswordPolicy and integrates with AuthService to maintain consistent security across authentication workflows.

- ***Image Processing***

ImageService: This class implements the core business logic for handling medical image uploads, validation, and storage. It verifies image format, size, and metadata before persisting the image record in the database. The service also manages secure file storage and triggers inference operations through the InferenceService.

InferenceService: This class coordinates the execution of AI inference tasks on uploaded medical images. It selects the appropriate model version, invokes the model runner, and handles timeout, retry, or failure conditions. The service ensures that each inference request is processed safely and consistently, storing results in the AIResult repository.

ResultService: This class manages the lifecycle of AI inference results. It handles result retrieval, updates, and override operations initiated by doctors. The service enforces data integrity by maintaining consistent status transitions and ensures that all modifications are auditable.

InferenceModelRegistryService: This class maintains references to available AI model versions and their metadata. It ensures that each inference is bound to a valid, approved model release and provides version lookup and validation functions. The service supports controlled model updates to ensure reproducibility and compliance.

StorageService: This class provides abstraction for managing image file storage and retrieval. It implements storage policies for file naming, retention, and deletion. The service ensures that medical images are stored in compliance with healthcare data security standards and remain accessible for authorized users only.

- ***Symptom Analysis***

ConsentService: This class implements the business logic for consent tracking and enforcement within the symptom analysis workflow. It records consent view/accept events for a given consent version and language, persists them through the consent repository, and logs consent-related actions via the audit service to support compliance traceability. It also enforces consent prerequisites by blocking symptom submission when consent has not been accepted.

PatientTextModerationService: This class implements the business logic for moderating and rewriting LLM-generated content for patient display. It evaluates generated text using the moderation model client, determines whether the content should be allowed, rewritten, or blocked, and records moderation decisions through the audit service to ensure moderation actions are traceable.

PatientOutputPolicy: This class applies patient-facing output constraints to reduce anxiety-inducing or overly technical language. It sanitizes forbidden medical terms and simplifies explanation text to ensure the response remains understandable and appropriate for non-clinical users.

DisclaimerPolicy: This class is responsible for enforcing mandatory disclaimer coverage on all patient-facing outputs. It retrieves the correct disclaimer for the selected language and appends it to generated content to guarantee that medical decision responsibility is clearly communicated.

SymptomAnalysisService: This class implements the end-to-end symptom analysis orchestration under the platform's latency target (5 seconds). It enforces role and consent requirements, persists symptom reports and results, calls the prompt builder and LLM client, applies moderation and patient output policies, attaches disclaimers, records audit events, and publishes latency metrics. It exposes methods for initial symptom submission, report analysis execution, and applying safety transformations to raw model output.

LLMClient (Adapter): This class provides an abstraction for invoking the external LLM provider under a defined timeout budget. It generates raw completions from prepared prompts and supports parsing outputs into a structured form for reliable downstream processing.

- ***Data Security and Compliance***

EncryptionService: This class provides at-rest encryption utilities used across subsystems to protect sensitive persisted values and binary payloads. It supports field-level encryption/decryption and byte-level encryption/decryption, using the configured algorithm and the active key reference provided by the key management component.

AnonymizationService: This class provides PII minimization utilities used to reduce exposure of identifiable information. It anonymizes image metadata and generates pseudonymous user identifiers to support safer processing and storage while limiting linkage risk.

- ***Feedback and Moderation***

FeedbackService: This class implements the core business logic for feedback submission and listing. It enforces authorization rules and ensures feedback actions are recorded for traceability through the logging subsystem.

AnnotationService: This class implements the business logic for managing doctor annotations and override actions. It validates doctor permissions, persists the operations, and records them as traceable actions.

OutputModerationService: This class is responsible for evaluating AI-generated outputs to detect misleading or unsafe content before presentation. It applies predefined moderation rules to decide whether an output should be allowed, sanitized, or suppressed as part of the safety workflow.

- ***Monitoring and Logging***

AuditService: This class generates audit events for critical platform actions including user access, AI inference operations, and feedback-related actions. It centralizes audit creation so that traceability is consistently applied across the system.

SystemHealthService: This class performs system health evaluations and dependency checks to determine whether the platform is operating normally. It provides structured health results that can be consumed by monitoring dashboards and operational tooling.

MetricsService: This class collects runtime performance indicators such as latency, uptime signals, and concurrency measures. It supports operational monitoring and performance analytics required by dashboards.

- ***Localization and UI***

LocalizationService: This class manages language-specific text bundles used throughout the system interface. It provides key-based translation lookup functionality for UI components and backend responses, and verifies translation completeness to ensure full coverage across supported languages.

LanguageNegotiationService: This class resolves the effective language for a user session. It determines the preferred language by evaluating stored user preferences and HTTP Accept-Language headers, and applies a predefined fallback strategy when necessary to guarantee consistent localization behavior.

2.1.4. Repository

- ***Symptom Analysis***

ConsentRepository: This class encapsulates persistence operations for consent records. It abstracts database access for storing consent view/accept events and retrieving the latest consent state for a given user to support compliance enforcement in the symptom workflow.

SymptomRepository: This class encapsulates persistence operations for symptom reports submitted by patients. It enables storing new symptom reports, retrieving a report by identifier, and listing prior reports for a patient without coupling the symptom analysis business logic to storage details.

SymptomResultRepository: This class encapsulates persistence operations for symptom analysis results. It supports storing finalized patient-safe outputs and retrieving a specific result by identifier for patient-facing result rendering and traceability.

ModerationModelClient (Adapter): This class abstracts access to the underlying moderation model used to evaluate generated text. It provides a stable interface for obtaining moderation

decisions without coupling the moderation service to a specific model implementation or deployment.

- ***Feedback and Moderation***

  FeedbackRepository: This class encapsulates persistence operations for feedback records. It abstracts database access so feedback data can be stored and retrieved without coupling business logic to storage details.

  AnnotationRepository: This class encapsulates persistence operations for doctor annotations linked to cases. It supports storing and retrieving annotations for later review.

  OverrideRepository: This class encapsulates persistence operations for doctor override actions on AI results. It supports retrieving overrides for a given AI result when needed.

- ***Monitoring and Logging***

  AuditRepository: This class encapsulates persistence operations for audit logs. It enables saving immutable audit events and retrieving them by time range for compliance and investigation purposes.

- ***Localization and UI***

  LocalizationRepository: This class manages the persistence and retrieval of localization bundles when they are stored externally (e.g., database or file storage). It is responsible for loading language-specific bundles, persisting updates, and providing bundle access by language to support the LocalizationService.

2.2. Client

The client layer is responsible for rendering user interfaces and communicating with the backend services. It provides role-based views for different user types and integrates localization mechanisms to ensure multilingual support across the platform.

The client is divided into two main parts:

- View Layer
- Components Layer

2.2.1. View

The View layer is responsible for rendering role-specific user interfaces and presenting localized content to end users. Each view consumes translation keys provided by the Localization component and communicates with backend services through the API component.

Doctor

This view provides the interface used by doctors to review AI-generated outputs, add annotations, submit feedback, and override AI results.

Responsibilities:

- Render localized UI text
- Allow language selection
- Display AI outputs and feedback forms
- Consume backend APIs for moderation and training-related actions

Patient

This view provides the interface used by patients to submit symptoms, upload medical images, and receive AI-supported insights.

Responsibilities:

- Render localized UI text
- Allow language selection
- Display AI responses and medical disclaimers
- Communicate with backend inference endpoints

Admin

This view provides administrative functionality including system monitoring, audit access, and model lifecycle management.

Responsibilities:

- Render localized UI text
- Display system status information
- Trigger training workflows via API
- View production model information
- Access localization coverage results (retrieved from server)

2.2.2. Components

The Components layer contains reusable frontend modules that support rendering, state management, localization, and backend communication.

Localization

This component is responsible for handling client-side multilingual support within the application. It manages the active language state (Turkish / English), resolves translation keys for UI elements, and ensures a reliable default language fallback when necessary. The component integrates with backend language negotiation mechanisms by attaching the user's selected language preference to API requests, allowing consistent localization across frontend and backend interactions. It stores the selected language in session or local storage to preserve user preference across sessions and guarantees uniform language usage throughout the user interface. While it collaborates closely with the backend 'LocalizationService', it is implemented independently to handle frontend rendering and dynamic UI translation efficiently.

APIClient

This component handles communication between the client and the backend server.

Responsibilities:

- Send HTTP requests to backend services
- Attach Accept-Language header
- Handle authentication tokens
- Parse localized backend responses
- Trigger training-related operations
- Fetch model and dataset information

All business logic remains on the server; the API component acts only as a communication bridge.

3. Class Interfaces

3.1. User Management Subsystem

class User
This class represents a system user, either a doctor or a patient. It stores credentials, roles, and registration metadata to support secure authentication and authorization.
Attributes
- user_id: UUID - email: str

- password_hash: str - role: Role - created_at: datetime - last_login: datetime
Methods
- verify_password(password: str) -> bool - update_last_login() -> None - to_dict() -> dict

class SessionToken
This class represents an active authentication session. It maintains issued token metadata and lifecycle state.
Attributes
- token_id: UUID - user_id: UUID - issued_at: datetime - expires_at: datetime - ip_address: str - is_active: bool
Methods
- is_valid(current_time: datetime) -> bool - invalidate() -> None - remaining_time() -> int

class AuthController
This class provides API endpoints for user authentication and token lifecycle management.
Attributes
- auth_service: AuthService - session_service: SessionService
Methods
- login(request: LoginRequest, ip_address: str | None = None) -> LoginResponse - logout(token_id: UUID, ip_address: str | None = None) -> None - refresh(token_id: UUID, ip_address: str | None = None) -> LoginResponse

class UserController
This class exposes user profile management endpoints for both patients and doctors.
Attributes
- user_service: UserService
Methods
- get_user(user_id: UUID) -> dict - update_user(user_id: UUID, payload: dict) -> dict - get_activity(user_id: UUID) -> list

class UserService
This class manages user-related operations, including registration, profile updates, and access control.
Attributes
- user_repo: UserRepository - password_service: PasswordService
Methods
- register(email: str, password: str, role: Role) -> User - update_profile(user_id: UUID, data: dict) -> User - get_user(user_id: UUID) -> User

class AuthService
This class manages authentication workflows such as login, token issuance, and validation.
Attributes
- user_service: UserService - session_service: SessionService - password_service: PasswordService - audit_service: AuditService
Methods
- authenticate(email: str, password: str) -> SessionToken - validate_token(token_id: UUID) -> bool - logout(token_id: UUID) -> None

class SessionService
This class maintains a session lifecycle and ensures valid access tokens across the system.
Attributes
- session_repo: SessionRepository
Methods
- create_session(user_id: UUID, ip_address: str) -> SessionToken - validate_session(token_id: UUID) -> bool - terminate_session(token_id: UUID) -> None

class PasswordService
This class provides secure password hashing, verification, and strength validation logic.
Attributes
- policy: PasswordPolicy
Methods
- hash(password: str) -> str - verify(password: str, password_hash: str) -> bool - check_strength(password: str) -> bool

3.2. Image Processing Subsystem

class MedicalImage
This class represents a diagnostic image uploaded by a doctor. It serves as the main data object for AI-based inference workflows.
Attributes
- image_id: UUID - case_id: UUID - uploader_id: UUID - file_path: str - upload_timestamp: datetime - model_version: str
Methods
- validate_format() -> bool - get_storage_path() -> str

- assign_model_version(version: str) -> None

class AIResult
This class stores AI inference outputs produced for a medical image. It provides diagnostic labels, confidence scores, and result status.
Attributes
- result_id: UUID - image_id: UUID - prediction_label: str - confidence_score: float - created_at: datetime - status: ResultStatus
Methods
- is_low_confidence(threshold: float) -> bool - override(label: str) -> None - to_dict() -> dict

class ResultStatus (ENUM)
Defines possible states of an inference result.
Values
- ACTIVE - OVERRIDDEN

class ImageUploadRequest
This class represents a data transfer object for image upload requests from the frontend.
Attributes
- uploader_id: UUID - case_id: UUID - file_path: str
Methods
- to_dict() -> dict

class InferenceResponse
This class encapsulates structured inference results returned to clients or doctor dashboards.
Attributes
- result_id: UUID - prediction_label: str - confidence_score: float - model_version: str - status: str
Methods
- to_dict() -> dict

class ModelVersion
This class represents a deployed and traceable AI model version used for inference.
Attributes
- version_id: str - architecture: str - release_date: datetime - accuracy: float - notes: str
Methods
- is_valid() -> bool - get_metadata() -> dict

class ImageController
This class provides the API endpoints for uploading, validating, and managing medical images.
Attributes
- image_service: ImageService - inference_service: InferenceService - storage_service: StorageService - max_upload_size: int - allowed_formats: list[str]
Methods
- upload_image(request: ImageUploadRequest) -> InferenceResponse - validate_image(file_path: str) -> bool - get_image_metadata(image_id: UUID) -> dict

- delete_image(image_id: UUID) -> bool

class ResultController
This class provides the API endpoints for retrieving, reviewing, and managing AI-generated results.
Attributes
- result_service: ResultService - audit_service: AuditService - confidence_threshold: float - low_confidence_policy: str
Methods
- get_result(image_id: UUID) -> dict - list_results_by_doctor(doctor_id: UUID) -> list[dict] - override_result(result_id: UUID, new_label: str) -> None - filter_low_confidence(results: list[dict]) -> list[dict]

class ImageService
This class manages medical image upload and storage workflows.
Attributes
- storage_service: StorageService - inference_service: InferenceService - image_repo: ImageRepository
Methods
- upload_image(request: ImageUploadRequest) -> MedicalImage - validate_image(file_path: str) -> bool - store_metadata(image: MedicalImage) -> None

class InferenceService
This class coordinates inference execution using registered AI models.
Attributes
- model_registry: InferenceModelRegistryService - result_repo: ResultRepository - audit_service: AuditService
Methods

- run_inference(image_id: UUID) -> AIResult
- handle_timeout(image_id: UUID) -> None
- retry(image_id: UUID) -> AIResult

class ResultService

This class manages AI inference results, including retrieval and override actions.

Attributes

- result_repo: ResultRepository
- audit_service: AuditService
- confidence_threshold: float

Methods

- get_result(image_id: UUID) -> AIResult
- override_result(result_id: UUID, new_label: str) -> None
- list_results_by_doctor(doctor_id: UUID) -> list[AIResult]
- low_confidence_policy: LowConfidencePolicy (ENUM)

class InferenceModelRegistryService

This class maintains references to deployed AI model versions and ensures controlled model updates.

Attributes

- models: dict[str, ModelVersion]
- active_version: str

Methods

- get_active_model() -> ModelVersion
- get_model_by_version(version: str) -> ModelVersion
- register_model(model: ModelVersion) -> None

class StorageService

This class provides abstraction for file storage, retrieval, and retention of medical images.

Attributes

- root_path: str
- naming_convention: str
- retention_period_days: int

Methods
- validate_storage_path() -> bool - generate_file_name(case_id: UUID, image_id: UUID, ext: str, timestamp: datetime) -> str - save_image(file_bytes: bytes, case_id: UUID, image_id: UUID, ext: str, filename: str, timestamp: datetime) -> str - delete_image(file_path: str) -> bool - get_image(file_path: str) -> bytes - purge_expired_files(now: datetime | None = None) -> int

3.3. Symptom Analysis Subsystem

class SymptomReport
This class represents a free-text symptom report submitted by a patient. It captures the raw symptom description together with basic context such as patient identity, optional case linkage, and language for downstream processing.
Attributes
id: UUID patient_id: UUID case_id: UUID | None text: str language: Language created_at: datetime
Methods
is_empty() -> bool

class SymptomAnalysisResult
This class represents the finalized symptom analysis output prepared for patient display. It stores the risk level, patient-safe explanation text, possible safe conditions, moderation decision, and mandatory disclaimer content.
Attributes
id: UUID report_id: UUID risk_level: RiskLevel safe_text: str possible_conditions_safe: list[str] moderation_action: ModerationAction disclaimer_text: str created_at: datetime
Methods

as_patient_payload() -> dict

class ConsentRecord
This class represents a persisted consent record used to track whether a user has viewed and accepted the required consent terms. It supports compliance verification and auditability for the symptom submission workflow.
Attributes
id: UUID user_id: UUID version: str language: Language viewed_at: datetime accepted_at: datetime | None ip_address: str | None
Methods
is_accepted() -> bool

class ConsentService
This class manages consent state changes and enforces consent prerequisites for symptom analysis. It records view/accept events through the repository layer and ensures consent has been accepted before symptom submission is allowed.
Attributes
consent_repository: ConsentRepository audit_service: AuditService
Methods
record_viewed(user_id: UUID, version: str, lang: Language, ip: str | None = None) -> None record_accepted(user_id: UUID, version: str, lang: Language, ip: str | None = None) -> None require_accepted(user_id: UUID) -> None

class LLMClient
This class abstracts external LLM invocation for symptom analysis, enforcing a strict timeout budget. It supports generating raw text outputs and parsing them into structured form for downstream processing.
Attributes
provider_name: str timeout_ms: int = 5000

Methods
generate(prompt: str) -> str parse_structured(output: str) -> LLMStructuredOutput

class PatientTextModerationService
This class moderates and, when necessary, rewrites LLM-generated outputs before they are shown to patients. It delegates moderation scoring to a model client and records moderation outcomes for traceability.
Attributes
moderation_model_client: ModerationModelClient audit_service: AuditService
Methods
moderate(text: str, lang: Language) -> ModerationDecision rewrite(text: str, lang: Language) -> str

class PatientOutputPolicy
This class applies patient-facing safety transformations by removing or masking technical/anxiety-inducing medical terms and simplifying language to ensure outputs are understandable and appropriate.
Attributes
forbidden_terms: set[str]
Methods
sanitize_medical_terms(text: str, lang: Language) -> str simplify(text: str, lang: Language) -> str

class DisclaimerPolicy
This class enforces mandatory disclaimer coverage for all patient-facing symptom analysis outputs. It provides language-specific disclaimer text and appends it to generated content.
Attributes
(stateless / no persistent attributes)
Methods
get_disclaimer(lang: Language) -> str append_disclaimer(text: str, lang: Language) -> str

class SymptomAnalysisService

This class orchestrates the end-to-end symptom analysis workflow under the target latency budget. It enforces RBAC and consent requirements, persists reports/results, invokes the LLM, applies moderation and patient output policies, appends disclaimers, and records audit/metrics.

Attributes

rbac_service: RBACService
consent_service: ConsentService
symptom_repository: SymptomRepository
symptom_result_repository: SymptomResultRepository
llm_client: LLMClient
moderation_service: PatientTextModerationService
patient_output_policy: PatientOutputPolicy
disclaimer_policy: DisclaimerPolicy
audit_service: AuditService
metrics_service: MetricsService

Methods

- submit_symptom(patient: User, text: str, lang: Language, case_id: UUID | None = None) -> SymptomAnalysisResult
- analyze_report(report_id: UUID) -> SymptomAnalysisResult
- apply_safety(raw_text: str, lang: Language) -> tuple[str, ModerationAction]
- _build_symptom_prompt(text: str, lang: Language, context: dict | None = None) -> str

class PatientSymptomController

This controller provides endpoints for consent tracking, symptom submission, and patient-safe result retrieval. It delegates consent operations to the consent service and symptom processing to the symptom analysis service.

Attributes

- consent_service: ConsentService
- symptom_analysis_service: SymptomAnalysisService

Methods

- record_consent_viewed(request: ConsentViewedRequest) -> VoidResponse
- record_consent_accepted(request: ConsentAcceptedRequest) -> VoidResponse
- submit_symptoms(request: SymptomSubmitRequest) -> SymptomResultDTO
- get_patient_result(result_id: UUID) -> PatientResultDTO

class ConsentRepository

This repository manages persistence and retrieval of consent records for compliance enforcement.

Attributes
(implementation-specific database/session dependency)
Methods
- save(record: ConsentRecord) -> ConsentRecord - find_latest_by_user(user_id: UUID) -> ConsentRecord | None

class SymptomRepository
This repository manages persistence operations for patient symptom reports.
Attributes
(implementation-specific database/session dependency)
Methods
- save(report: SymptomReport) -> SymptomReport - find_by_id(report_id: UUID) -> SymptomReport | None - list_by_patient(patient_id: UUID) -> list[SymptomReport]

class SymptomResultRepository
This repository manages persistence operations for symptom analysis results.
Attributes
(implementation-specific database/session dependency)
Methods
- save(result: SymptomAnalysisResult) -> SymptomAnalysisResult - find_by_id(result_id: UUID) -> SymptomAnalysisResult | None

class ModerationModelClient
This adapter provides a stable interface to the moderation model used for evaluating generated text.
Attributes
(implementation-specific client configuration)
Methods
- predict(text: str, lang: Language) -> ModerationDecision

3.4. Data Security and Compliance Subsystem

class EncryptionService
This class provides at-rest encryption utilities to protect sensitive data stored by the platform. It supports field-level encryption for persisted values and byte-level encryption for binary payloads, using a configured cryptographic algorithm and an active key reference.
Attributes
- algorithm: str - key_ref: str
Methods
- encrypt_field(value: str) -> str - decrypt_field(value: str) -> str - encrypt_bytes(bytes: bytes) -> bytes - decrypt_bytes(bytes: bytes) -> bytes

class AnonymizationService
This class provides PII minimization utilities to reduce exposure of identifiable information across subsystems. It supports anonymizing image metadata and generating pseudonymous identifiers for users when direct identifiers should not be stored or propagated.
Attributes
(stateless / no persistent attributes)
Methods
- anonymize_image_metadata(meta: dict) -> dict - pseudonymize_user_id(user_id: UUID) -> str

3.5. Feedback & Moderation Subsystem

class FeedbackEntry
This class represents a feedback record submitted by a doctor regarding an AI-generated result. It is used to capture structured feedback for model improvement, auditing, and quality monitoring purposes.
Attributes
- id: UUID

- doctor_id: UUID - case_id: UUID - target_result_id: UUID - feedback_type: FeedbackType - comment: str - created_at: datetime
Methods
- short() -> str

class DoctorAnnotation
This class represents textual annotations added by doctors to cases or AI outputs. It allows medical professionals to attach contextual explanations or notes to specific targets such as images, symptoms, or reports.
Attributes
- id: UUID - case_id: UUID - doctor_id: UUID - target_type: str - target_id: UUID - text: str - created_at: datetime
Methods
- preview(n: int) -> str

class DoctorOverride
This class represents an explicit override action performed by a doctor on an AI-generated decision. It records corrected labels and justification notes to ensure traceability and accountability.
Attributes
- id: UUID - doctor_id: UUID - target_result_id: UUID - override_label: str | None - note: str - created_at: datetime
Methods
- has_label() -> bool

class FeedbackService
This class manages the submission and retrieval of doctor feedback. It enforces authorization rules, records audit events, and tracks latency metrics related to feedback operations.
Attributes
- feedback_repository: FeedbackRepository - rbac_service: RBACService - audit_service: AuditService - metrics_service: MetricsService
Methods
- submit(doctor: User, case_id: UUID, target_result_id: UUID, feedback_type: FeedbackType, comment: str) -> FeedbackEntry - list(filters: dict | None = None) -> list[FeedbackEntry]

class AnnotationService
This class handles doctor annotations and override workflows. It ensures only authorized users can annotate or override AI results and logs all critical actions for compliance.
Attributes
- annotation_repository: AnnotationRepository - override_repository: OverrideRepository - rbac_service: RBACService - audit_service: AuditService
Methods
- add_annotation(doctor: User, case_id: UUID, target_type: str, target_id: UUID, text: str) -> DoctorAnnotation - override_result(doctor: User, target_result_id: UUID, override_label: str | None, note: str) -> DoctorOverride

class OutputModerationService
This class evaluates AI-generated outputs to detect unsafe or misleading content before they are shown to users. It applies moderation rules to allow, sanitize, or suppress outputs.
Attributes
- audit_service: AuditService
Methods
- evaluate(output_text: str, context: dict | None = None) -> ModerationDecision - sanitize(output_text: str, decision: ModerationDecision) -> str

class FeedbackController
This controller exposes API endpoints for submitting and querying doctor feedback related to AI results.
Attributes
- feedback_service: FeedbackService
Methods
- submit_feedback(request: FeedbackSubmitRequest) -> FeedbackDTO - list_feedback(filters: dict | None = None) -> list[FeedbackDTO]

class AnnotationController
This controller provides endpoints for adding annotations to cases and overriding AI-generated results.
Attributes
- annotation_service: AnnotationService
Methods
- add_annotation(case_id: UUID, request: AnnotationCreateRequest) -> AnnotationDTO - override_result(result_id: UUID, request: OverrideRequest) -> OverrideDTO

class FeedbackRepository
This repository handles persistence operations for feedback entries.
Attributes
- (implementation-specific database/session dependency)
Methods
- save(entry: FeedbackEntry) -> FeedbackEntry - list(filters: dict | None = None) -> list[FeedbackEntry]

class AnnotationRepository
This repository manages storage and retrieval of doctor annotations.
Attributes
- (implementation-specific database/session dependency)

Methods
- save(annotation: DoctorAnnotation) -> DoctorAnnotation - list_by_case(case_id: UUID) -> list[DoctorAnnotation]

class OverrideRepository
This repository manages persistence of doctor override decisions.
Attributes
- (implementation-specific database/session dependency)
Methods
- save(override: DoctorOverride) -> DoctorOverride - find_by_target(target_result_id: UUID) -> DoctorOverride | None

3.6. Monitoring & Logging Subsystem

class AuditLog
This class represents an immutable audit record for system actions such as authentication events (LOGIN/LOGOUT), AI inference executions, moderation decisions, and override operations. The details_json field stores structured metadata specific to the action type.
Attributes
- id: UUID - user_id: UUID - action: str - entity_type: str - entity_id: UUID - status: str - timestamp: datetime - ip_address: str | None - details_json: dict | None
Methods
- to_summary() -> str

class MonitoringController
This controller exposes operational endpoints used by dashboards and monitoring tools to retrieve system health status, dependency status, and runtime performance metrics. It acts as a thin API layer over the monitoring services to support uptime monitoring and performance analytics.
Attributes
- system_health_service: SystemHealthService - metrics_service: MetricsService
Methods
- get_health() -> HealthStatus - get_dependencies() -> list[DependencyStatus] - get_metrics() -> MetricsSnapshot

class AuditController
This controller provides endpoints for querying audit log records for traceability, compliance review, and incident investigation. It accepts filter parameters, delegates query logic to the audit service, and returns log DTOs/summaries appropriate for administrative views.
Attributes
- audit_service: AuditService
Methods
- list_audit_logs(filters: dict | None = None) -> list[AuditLog] - list_audit_logs_by_range(start: datetime, end: datetime) -> list[AuditLog]

class AuditService
This class is responsible for generating and storing audit logs to meet compliance, traceability, and regulatory requirements.
Attributes
- audit_repository: AuditRepository
Methods
- log_access(user_id: UUID, action: str, entity_type: str, entity_id: UUID, status: str, details: dict) -> None - log_inference(model_name: str, model_version: str, latency_ms: int, input_meta: dict, output_meta: dict) -> None - log_moderation(action: ModerationAction, reasons: list[str], confidence: float) -> None

class SystemHealthService
This class monitors the operational health of the system and checks the availability of dependent services.
Attributes
- (implementation-specific dependency check configuration)
Methods
- health_check() -> HealthStatus - dependency_check() -> list[DependencyStatus]

class MetricsService
This class collects and records runtime metrics such as latency, uptime, and concurrent user counts for monitoring and optimization.
Attributes
- (implementation-specific metrics backend or in-memory store)
Methods
- record_latency(name: str, ms: int) -> None - record_uptime(is_up: bool) -> None - record_concurrent_users(count: int) -> None

class AuditRepository
This repository manages persistence and retrieval of audit log records.
Attributes
- (implementation-specific database/session dependency)
Methods
- save(log: AuditLog) -> AuditLog - list_by_range(start: datetime, end: datetime) -> list[AuditLog]

3.7. Localization & UI Subsystem

class LocalizationService
This class is responsible for managing multilingual UI text resources for the system. It provides key-based translation functionality for Turkish and English and verifies 100% translation coverage by detecting missing keys in localization bundles.
Attributes
- bundles: dict[Language, dict[str, str]] - default_language: Language
Methods
- t(key: str, lang: Language): str -assertCoverage(lang: Language): CoverageReport

class LanguageNegotiationService
This class is responsible for selecting the effective language of the system for a user session. It resolves the language by prioritizing the user's saved preference (if provided) and otherwise using the HTTP Accept-Language header. If neither is available or valid, it falls back to a default language (e.g., TR). This service ensures consistent language selection for all user-facing outputs (UI text, messages, disclaimers).
Attributes
(No persistent attributes.)
Methods
- resolve(userPref?: Language, acceptLanguageHeader?: str): Language

3.8. Training & Model Development Subsystem

class DatasetVersion
This class represents a versioned dataset used for model training and evaluation. It stores dataset metadata, version information, integrity hash, and train/validation/test split configuration to ensure reproducibility.
Attributes
- id: UUID - name: str - version: str - source: str

- hash: str - splitInfo: JSON - createdAt: datetime
Methods
- getIdTag(): str

class TrainingJob
This class represents a training job and its lifecycle (queued → running → done). It stores the dataset reference, task type, hyperparameters, timestamps, and execution status.
Attributes
- id: UUID - taskType: TrainingTaskType - datasetVersionId: UUID - status: TrainingStatus - hyperparamsJson: JSON - startedAt: datetime - endedAt: datetime
Methods
- markRunning(): void - markFailed(): void - markSucceeded(): void

class ModelArtifact
This class represents a trained model artifact produced by a training job. It stores model metadata, framework information, storage location, and evaluation metrics.
Attributes
- id: UUID - modelName: str - version: str - framework: str - artifactUri: str - metricsJson: JSON - createdAt: datetime
Methods
- metric(name: str): float

class TrainingOrchestrator
This class is responsible for creating and managing training jobs. It coordinates trainers, evaluation, experiment tracking, and model registration.
Attributes
(No persistent attributes defined. Dependencies are injected externally.)
Methods
- createJob(taskType: TrainingTaskType, datasetVersionId: UUID, hyperparams: any): TrainingJob. - run(jobId: UUID): ModelArtifact - cancel(jobId: UUID): void

class ImageTrainer
This class is responsible for training computer vision models such as lung X-ray, brain CT, and ultrasound models.
Attributes
(No persistent attributes.)
Methods
- train(dataset: DatasetVersion, hyperparams: any): ModelArtifact - resume(checkpointUri: str): ModelArtifact

class NLPTrainer
This class is responsible for training NLP-based models including symptom analysis and moderation models.
Attributes
(No persistent attributes.)
Methods
- trainSymptomModel(dataset: DatasetVersion, hyperparams: any): ModelArtifact - trainModerationModel(dataset: DatasetVersion, hyperparams: any): ModelArtifact

class EvaluationService
This class evaluates trained model artifacts using a given test dataset and produces structured performance metrics. It computes evaluation outputs such as confusion matrices and verifies whether the model satisfies predefined

performance and safety constraints before deployment.
Attributes
(No persistent attributes.)
Methods
- evaluate(model: ModelArtifact, testSet: any): MetricsReport - computeConfusionMatrix(yTrue, yPred): ConfusionMatrix - checkConstraints(metrics: MetricsReport): ConstraintCheck

class TrainingModelRegistryService
This class is responsible for registering trained models and promoting selected versions to production.
Attributes
(No persistent attributes defined.)
Methods
- register(artifact: ModelArtifact): void - promoteToProduction(modelName: str, version: str): void - getProductionModel(modelName: str): ModelArtifact

class ExperimentTracker
This class logs experiment parameters, metrics, and artifacts for reproducibility.
Attributes
(No persistent attributes defined.)
Methods
- logParams(jobId: UUID, params: any): void - logMetrics(jobId: UUID, metrics: any): void - logArtifact(jobId: UUID, uri: str): void

class DatasetRepository
This repository manages persistence operations for dataset versions.
Attributes
(No persistent attributes defined.)

Methods
- save(ds: DatasetVersion): DatasetVersion - findById(id: UUID): DatasetVersion

class TrainingJobRepository
This repository manages persistence operations for training jobs.
Attributes
(No persistent attributes defined.)
Methods
- save(job: TrainingJob): TrainingJob - findById(id: UUID): TrainingJob - update(job: TrainingJob): TrainingJob

class ModelArtifactRepository
This repository manages persistence operations for model artifacts.
Attributes
(No persistent attributes defined.)
Methods
- save(m: ModelArtifact): ModelArtifact - findByNameVersion(name: str, version: str): ModelArtifact - findProduction(name: str): ModelArtifact

3.9. Client / Views

class DoctorView
This class renders the doctor interface for reviewing AI outputs, annotating results, and submitting feedback.
Attributes
- current_language: Language - api_client: APIClient - current_case_id: UUID

- ai_output: AIResult
- feedback_form: FeedbackData
- is_loading: bool
- last_error: ApiError

Methods

- render() -> None
- fetch_case(case_id: UUID) -> AIResult
- submit_feedback(data: FeedbackData) -> ApiResponse
- override_prediction(new_label: str) -> ApiResponse
- change_language(lang: Language) -> None
- handle_api_error(error: ApiError) -> None

class PatientView

This class renders the patient interface for symptom submission, image upload, and displaying AI-supported insights with disclaimers.

Attributes

- current_language: Language
- api_client: APIClient
- uploaded_image: File
- symptom_text: str
- ai_response: AIResponse
- disclaimer_visible: bool
- is_submitting: bool
- last_error: ApiError

Methods

- render() -> None
- submit_symptoms(symptom_text: str) -> AIResponse
- upload_image(image: File) -> AIResponse
- show_disclaimer() -> None
- change_language(lang: Language) -> None
- handle_api_error(error: ApiError) -> None

class AdminView

This class renders the administrative interface for monitoring, model lifecycle management, and training workflow control.

Attributes

- current_language: Language
- api_client: APIClient
- system_status: SystemStatus
- model_list: list[ModelInfo]

- dataset_list: list[DatasetInfo] - localization_coverage: LocalizationReport - is_loading: bool - last_error: ApiError
Methods
- render() -> None - fetch_system_status() -> SystemStatus - fetch_models() -> list[ModelInfo] - fetch_datasets() -> list[DatasetInfo] - trigger_training(dataset_id: UUID) -> ApiResponse - fetch_localization_report() -> LocalizationReport - change_language(lang: Language) -> None - handle_api_error(error: ApiError) -> None

3.10. Client / Components

class Localization
This component manages client-side localization state, resolves translation keys for UI rendering, persists user language preference, and provides safe fallback behavior.
Attributes
- current_language: Language - default_language: Language - storage_key: str - bundles: dict[Language, dict[str, str]]
Methods
- get_language() -> Language - set_language(lang: Language) -> None - t(key: str, lang: Language | None = None) -> str - load_from_storage() -> Language | None - save_to_storage(lang: Language) -> None - ensure_fallback(lang: Language | None = None) -> Language

class APIClient
This component provides a thin communication layer between the frontend and backend services. It attaches authentication and localization headers, sends HTTP requests, parses responses, and exposes typed client methods for core backend operations.
Attributes
- base_url: str

- auth_token: str | None - refresh_token: str | None - timeout_ms: int - default_headers: dict[str, str] - localization: Localization
Methods
- set_auth_token(token: str | None) -> None - set_refresh_token(token: str | None) -> None - build_headers() -> dict[str, str] - get(path: str, params: dict | None = None) -> dict - post(path: str, payload: dict | None = None) -> dict - put(path: str, payload: dict | None = None) -> dict - delete(path: str) -> dict - refresh_auth_if_needed() -> bool - parse_response(raw: dict) -> dict

4. UML Diagrams

4.1. Class Diagrams

This section presents the structural view of the AI Health System through four UML class diagrams, each representing a distinct architectural perspective:

- Core Domain Diagram
- Application Layer (Controllers & Services) Diagram
- Persistence & Infrastructure Diagram
- Training & Model Development Diagram

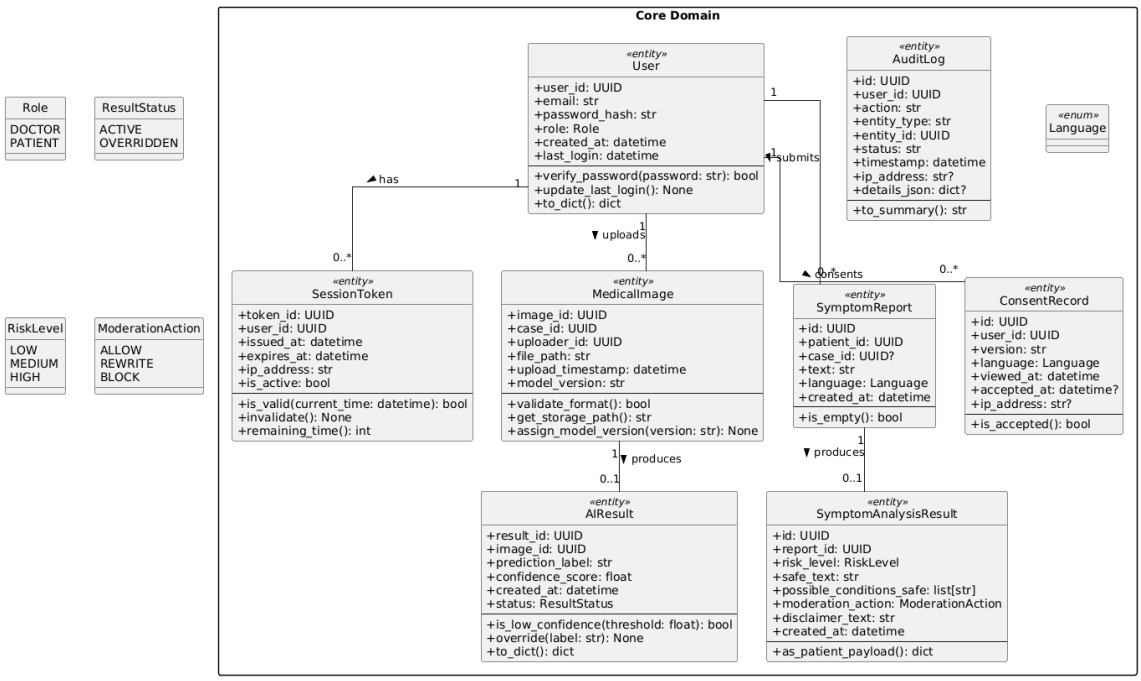
Due to the architectural scale and the number of domain entities, the complete system model has been divided into these logically grouped diagrams in order to improve readability and maintain visual clarity.

The diagrams are intentionally presented in a compact form, focusing on structural relationships, subsystem boundaries, and dependency flows rather than exhaustive member-level detail. Multiplicity (cardinality) is explicitly represented only in the Domain Model diagram where structural entity relationships are relevant. In the Application and Infrastructure diagrams, relationships represent dependency or usage rather than ownership or aggregation.

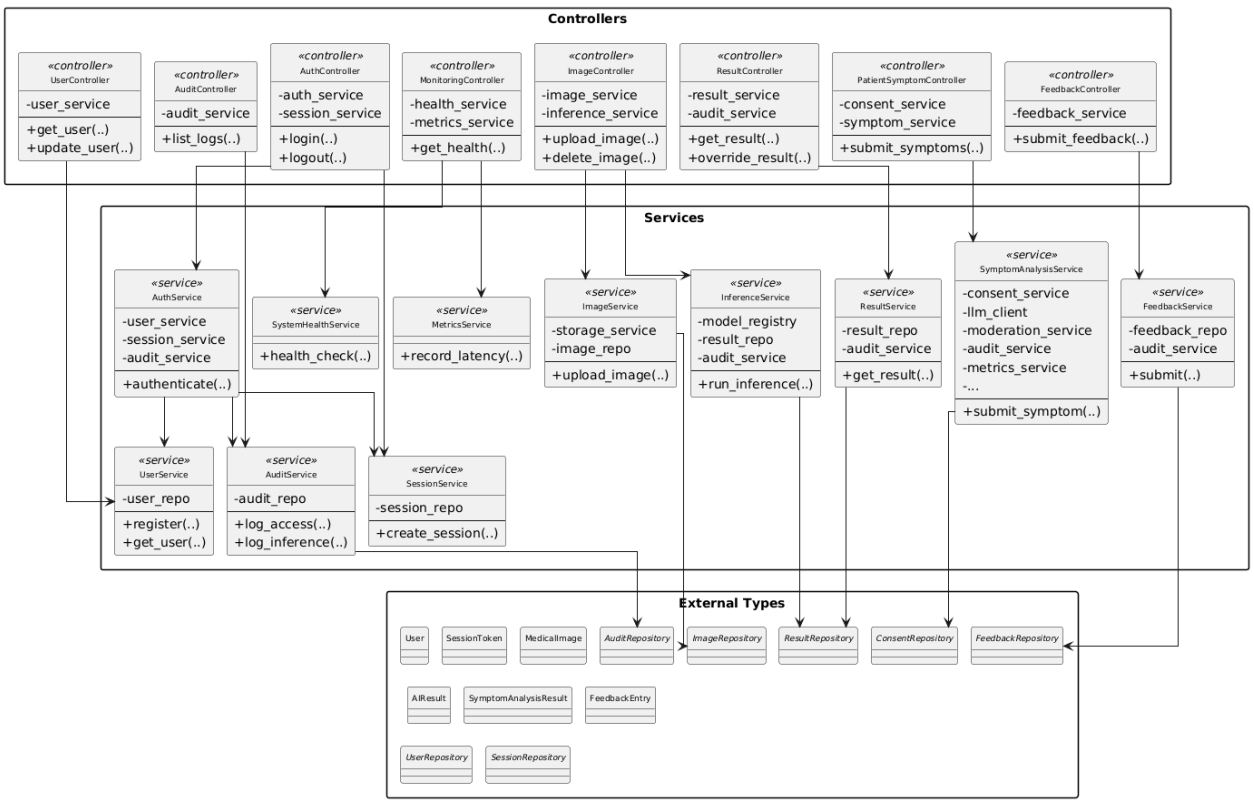
All classes shown in the diagrams correspond directly to the definitions provided in Section 3 (Class Interfaces). While method signatures and attributes may appear abbreviated in the diagrams for layout optimization, the full and precise specifications - including complete attribute lists, method signatures, parameter types, and return types - are formally documented in the Class Interfaces section.

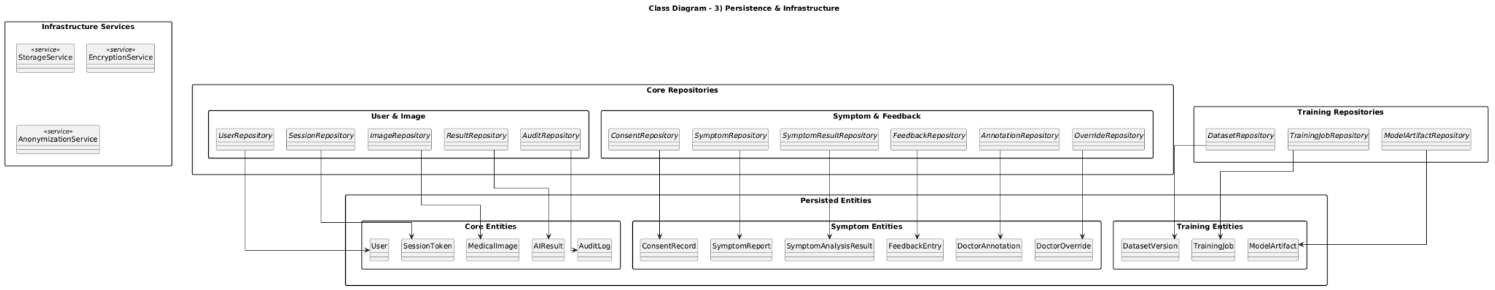
Therefore, the diagrams serve as a structural abstraction layer, whereas Section 3 provides the authoritative low-level implementation detail.

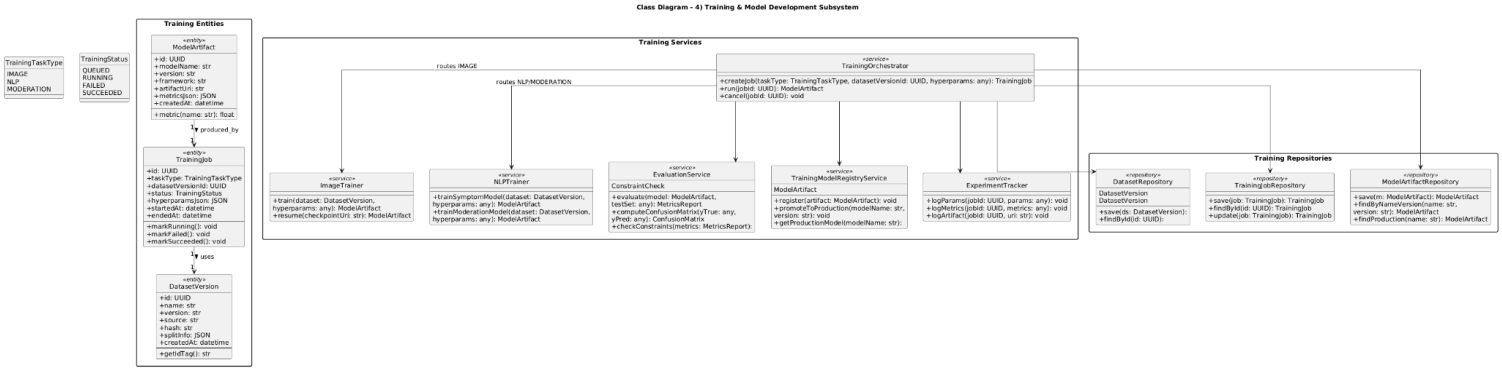
Class Diagram - 1) Core Domain (Entities + Enums)



Class Diagram - 2) Application Layer



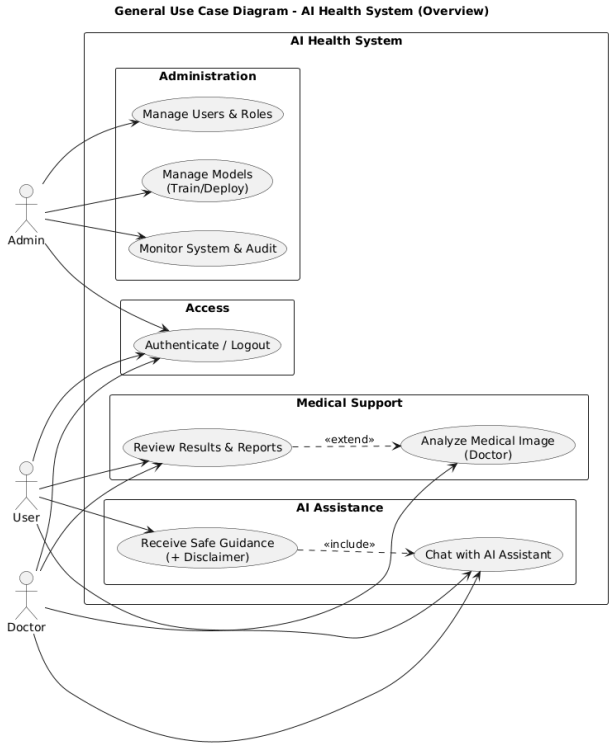

Class Diagram - 3) Persistence & Infrastructure
Infrastructure Services
«service»
StorageService
«service»
EncryptionService
«service»
AnonymizationService
Core Repositories
User & Image
UserRepository
SessionRepository
ImageRepository
ResultRepository
AuditRepository
Symptom & Feedback
ConsentRepository
SymptomRepository
SymptomResultRepository
FeedbackRepository
AnnotationRepository
OverrideRepository
Training Repositories
DatasetRepository
TrainingJobRepository
ModelArtifactRepository
Persisted Entities
Core Entities
User
SessionToken
MedicalImage
AIResult
AuditLog
Symptom Entities
ConsentRecord
SymptomReport
SymptomAnalysisResult
FeedbackEntry
DoctorAnnotation
DoctorOverride
Training Entities
DatasetVersion
TrainingJob
ModelArtifact


Class Diagram - 4) Training & Model Development Subsystem
TrainingTaskType
IMAGE
NLP
MODERATION
TrainingStatus
QUEUED
RUNNING
FAILED
SUCCEEDED
Training Entities
«entity»
ModelArtifact
+id: UUID
+modelName: str
+version: str
+framework: str
+artifactUri: str
+metricsJson: JSON
+createdAt: datetime
+metric(name: str): float
produced_by
«entity»
TrainingJob
+id: UUID
+taskType: TrainingTaskType
+datasetVersionId: UUID
+status: TrainingStatus
+hyperparamsJson: JSON
+startedAt: datetime
+endedAt: datetime
+markRunning(): void
+markFailed(): void
+markSucceeded(): void
uses
«entity»
DatasetVersion
+id: UUID
+name: str
+version: str
+source: str
+hash: str
+splitInfo: JSON
+createdAt: datetime
+getIdTag(): str
Training Services
routes IMAGE
routes NLP/MODERATION
«service»
TrainingOrchestrator
+createJob(taskType: TrainingTaskType, datasetVersionId: UUID, hyperparams: any): TrainingJob
+runJob(jobId: UUID): ModelArtifact
+cancelJob(jobId: UUID): void
«service»
ImageTrainer
+train(dataset: DatasetVersion, hyperparams: any): ModelArtifact
+resume(checkpointUri: str): ModelArtifact
«service»
NLPTrainer
+trainSymptomModel(dataset: DatasetVersion, hyperparams: any): ModelArtifact
+trainModerationModel(dataset: DatasetVersion, hyperparams: any): ModelArtifact
«service»
EvaluationService
ConstraintCheck
+evaluate(model: ModelArtifact, testSet: any): MetricsReport
+computeConfusionMatrix(yTrue: any, yPred: any): ConfusionMatrix
+checkConstraints(metrics: MetricsReport)
«service»
TrainingModelRegistryService
ModelArtifact
+register(artifact: ModelArtifact): void
+promoteToProduction(modelName: str, version: str): void
+getProductionModel(modelName: str):
«service»
ExperimentTracker
+logParams(jobId: UUID, params: any): void
+logMetrics(jobId: UUID, metrics: any): void
+logArtifact(jobId: UUID, uri: str): void
Training Repositories
«repository»
DatasetRepository
DatasetVersion
DatasetVersion
+save(ds: DatasetVersion): +findById(id: UUID):
«repository»
TrainingJobRepository
+save(job: TrainingJob): TrainingJob
+findById(id: UUID): TrainingJob
+update(job: TrainingJob): TrainingJob
«repository»
ModelArtifactRepository
+save(m: ModelArtifact): ModelArtifact
+findByNameVersion(name: str, version: str): ModelArtifact
+findProduction(name: str): ModelArtifact

4.2. Use Case Diagrams

General Use Case Diagram of the AI Health System

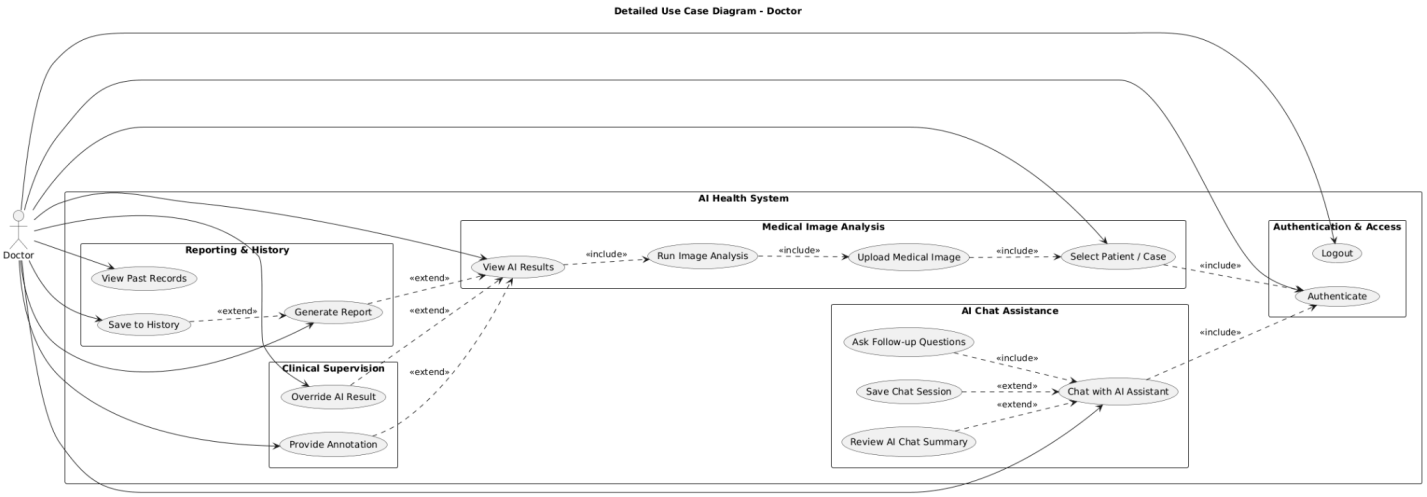
This diagram provides a high-level overview of the AI Health System by showing the three primary actors (User, Doctor, and Admin) and the main system capabilities. Users and doctors authenticate and interact with the AI assistant via chat to receive safety-controlled guidance. Doctors additionally use the system for medical image analysis and reviewing results and reports. Administrators manage users and roles, operate the model lifecycle, and monitor system health and audit information. Detailed role-specific use case diagrams further describe each workflow.



Detailed Use Case Diagram of Doctor

This use case diagram describes the detailed workflow of the Doctor role within the AI Health System. After authentication, the doctor can interact with the AI assistant via chat, ask follow-up questions, review generated summaries, and optionally save chat sessions. The doctor can also perform AI-assisted medical image analysis by selecting a patient case, uploading medical images, running image analysis, and reviewing the AI-generated results.

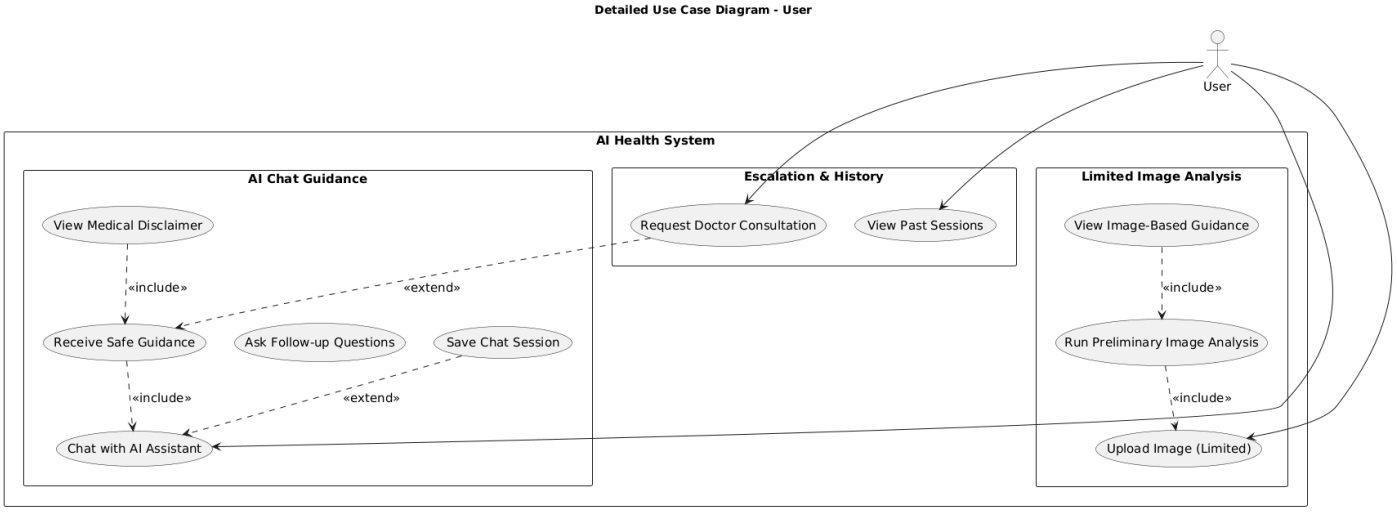
Clinical supervision capabilities such as providing annotations and overriding AI decisions are modeled as optional extensions after reviewing results, ensuring human-in-the-loop control. Reporting functions allow the doctor to generate reports and optionally store them in the system history. This diagram captures the full functional interaction scope of the Doctor role at the system boundary level.



Detailed Use Case Diagram of User

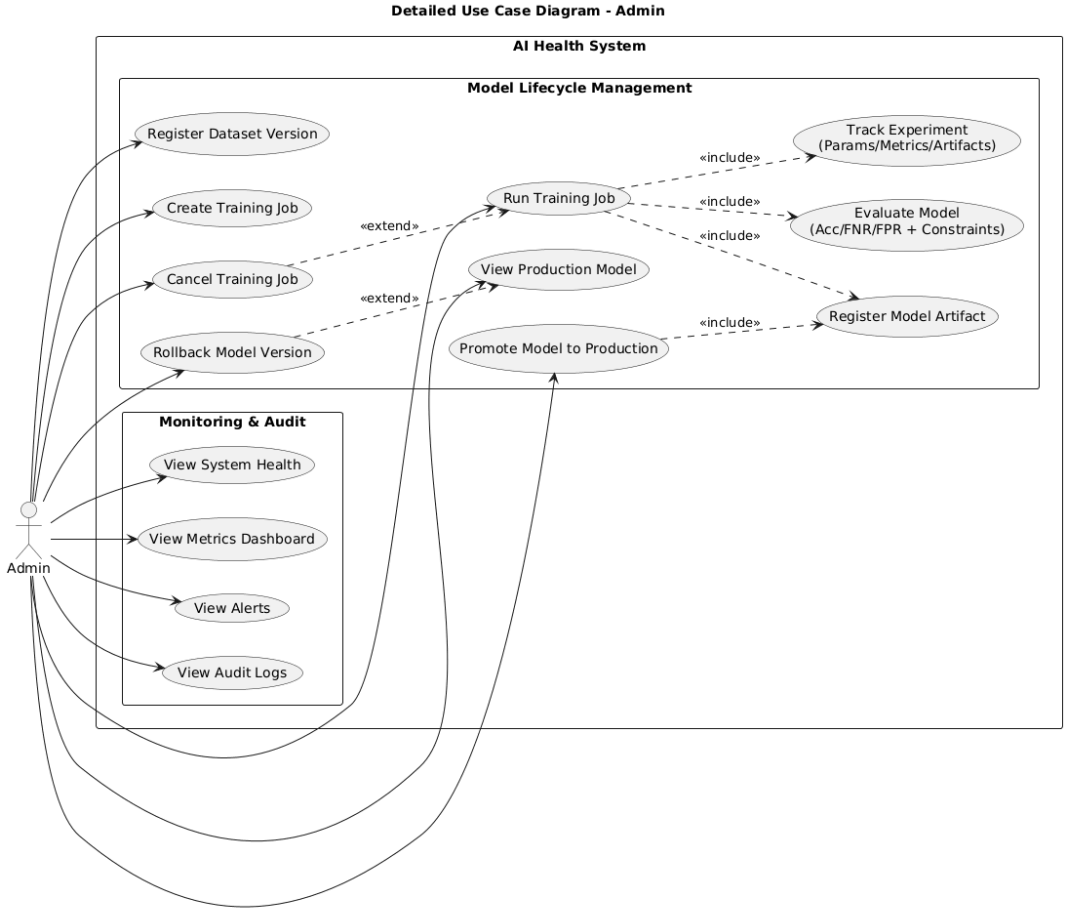
This use case diagram describes the functional interactions of the User role within the AI Health System. After authentication, the user interacts with the AI assistant via chat and receives safety-controlled guidance accompanied by a mandatory medical disclaimer. The user may ask follow-up questions and optionally save chat sessions.

Additionally, the system allows limited image uploads for preliminary AI analysis, after which image-based guidance is provided. Since this workflow does not replace clinical diagnosis, the user can optionally request a doctor consultation. Historical session viewing is also supported to maintain interaction continuity.



Detailed Use Case Diagram of Admin

This use case diagram describes the administrative capabilities of the AI Health System. The Admin role manages authentication-protected operations including user and role management (creating, updating, deactivating users, assigning roles, and password resets). In addition, the Admin controls the model lifecycle by registering dataset versions, creating and running training jobs, tracking experiments, evaluating models, registering model artifacts, and promoting validated models to production; rollback is supported as an optional safety measure. The Admin also monitors system operation through health status, metrics dashboards, alerts, and audit log access to support reliability and traceability.

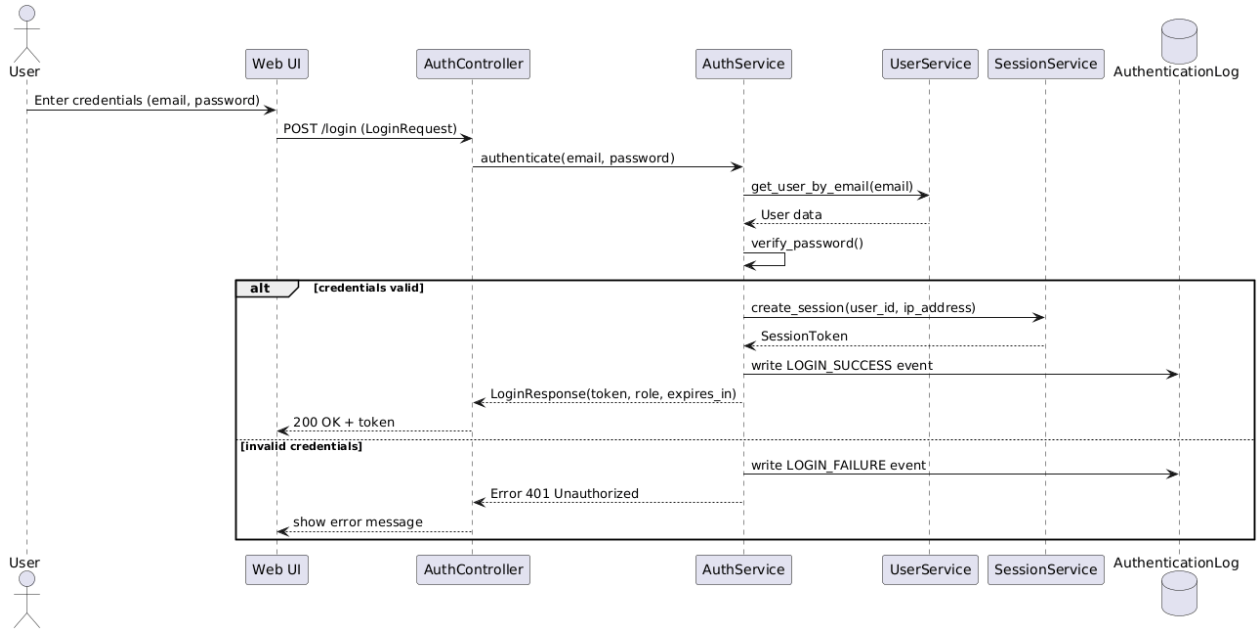


4.3. Sequence Diagrams

User Login & Session Creation Sequence Diagram

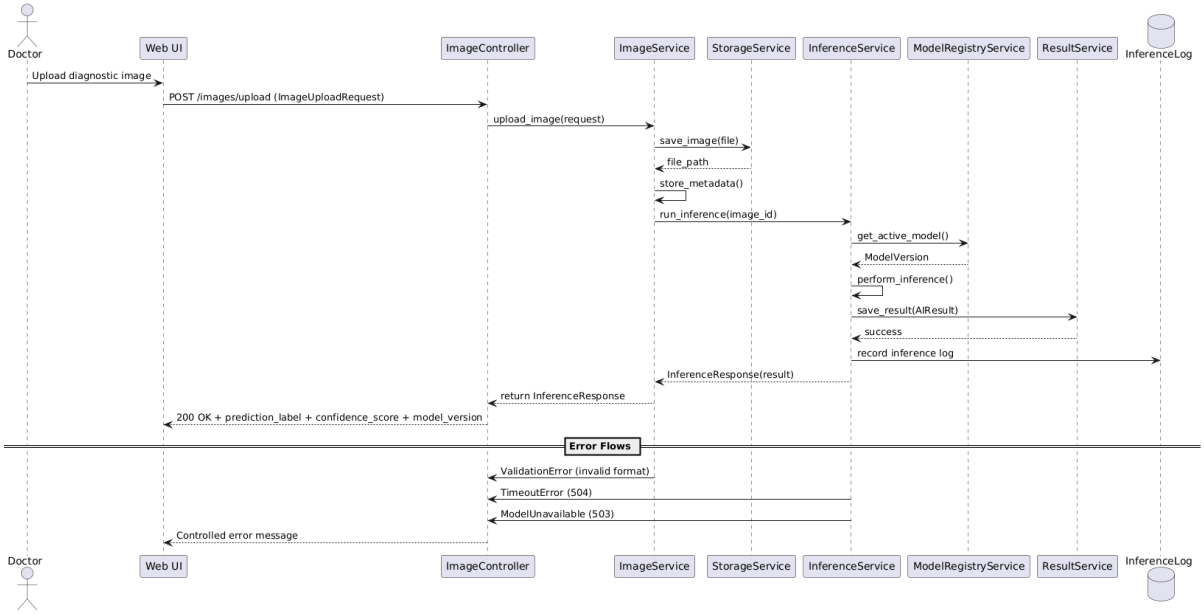
This sequence diagram shows the workflow when a user (doctor or patient) logs into the system and a secure session token is created. The request is received by the **AuthController**, validated by the **AuthService**, and checked against stored credentials via the **UserService**. Upon successful authentication,

the **SessionService** issues a new session token and records the event. Finally, a login response containing the token and expiration time is returned to the user.



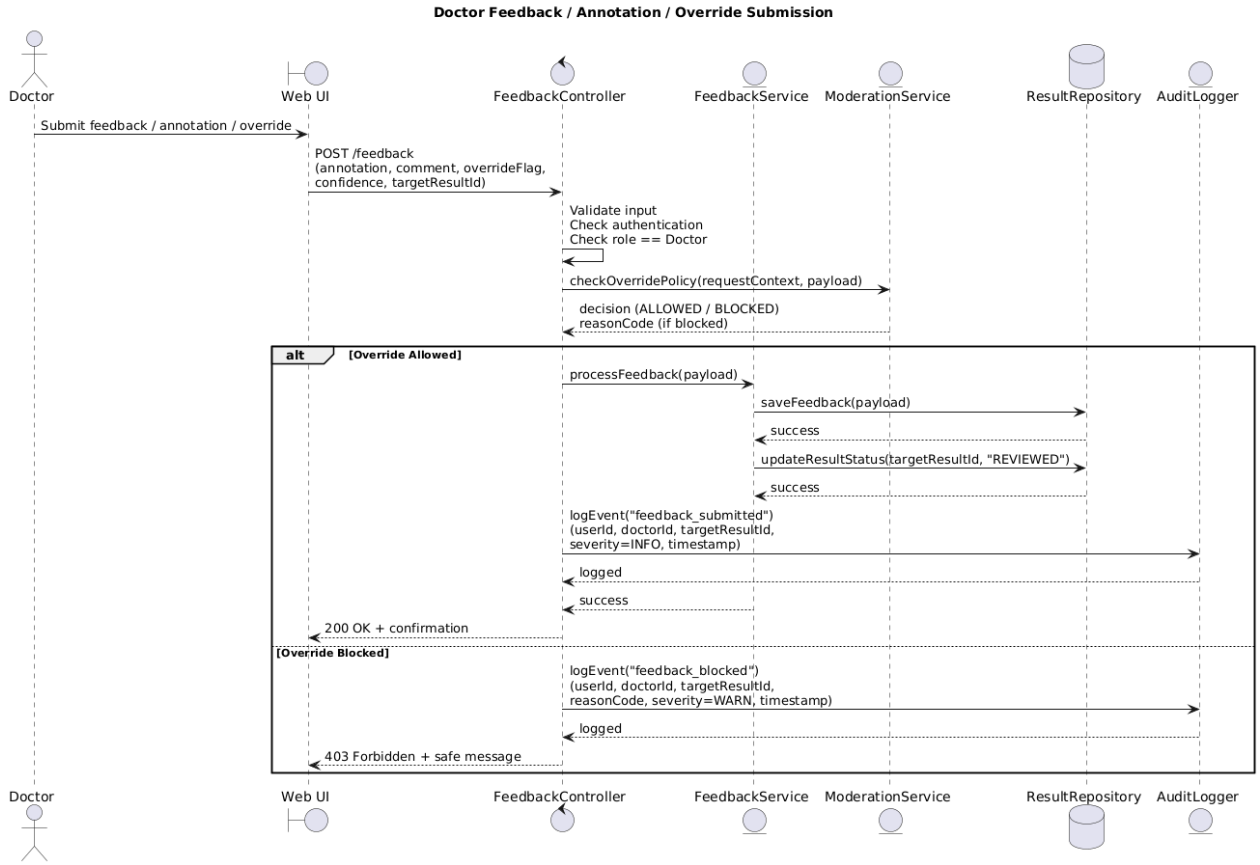
Doctor Upload Image → AI Inference → Result Stored Sequence Diagram

This sequence diagram illustrates the workflow when a doctor uploads a diagnostic image that is processed by the AI inference pipeline. The request is received by the **ImageController**, validated, and forwarded to the **ImageService** for file storage. Once stored, the **InferenceService** triggers a model inference using the latest model version registered in **InferenceModelRegistryService**. The generated results are persisted in the **ResultRepository**. A structured **InferenceResponse** is then returned to the doctor.



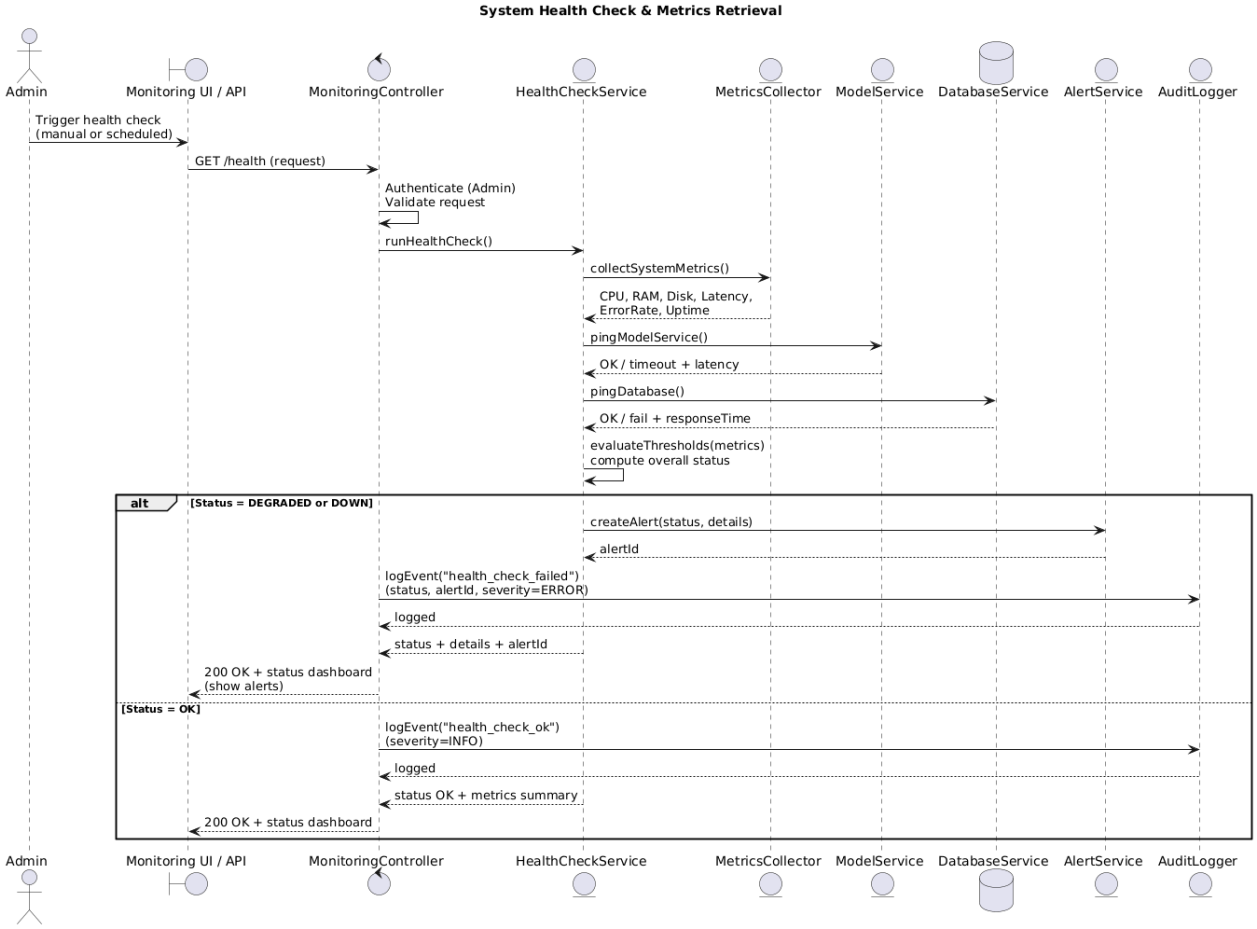
Doctor Feedback / Annotation / Override Submission Sequence Diagram

This sequence diagram shows the workflow when a doctor submits feedback, annotations, or an override decision for an AI-generated result. The request is received by the FeedbackController, validated, and evaluated by the PatientTextModerationService to ensure policy compliance. If approved, the feedback is processed and stored in the database, and the result status is updated accordingly. All actions are recorded by the AuditLogger to maintain traceability. If the request is blocked, a controlled response is returned and the event is logged.



System Health Check & Metrics Retrieval Sequence Diagram

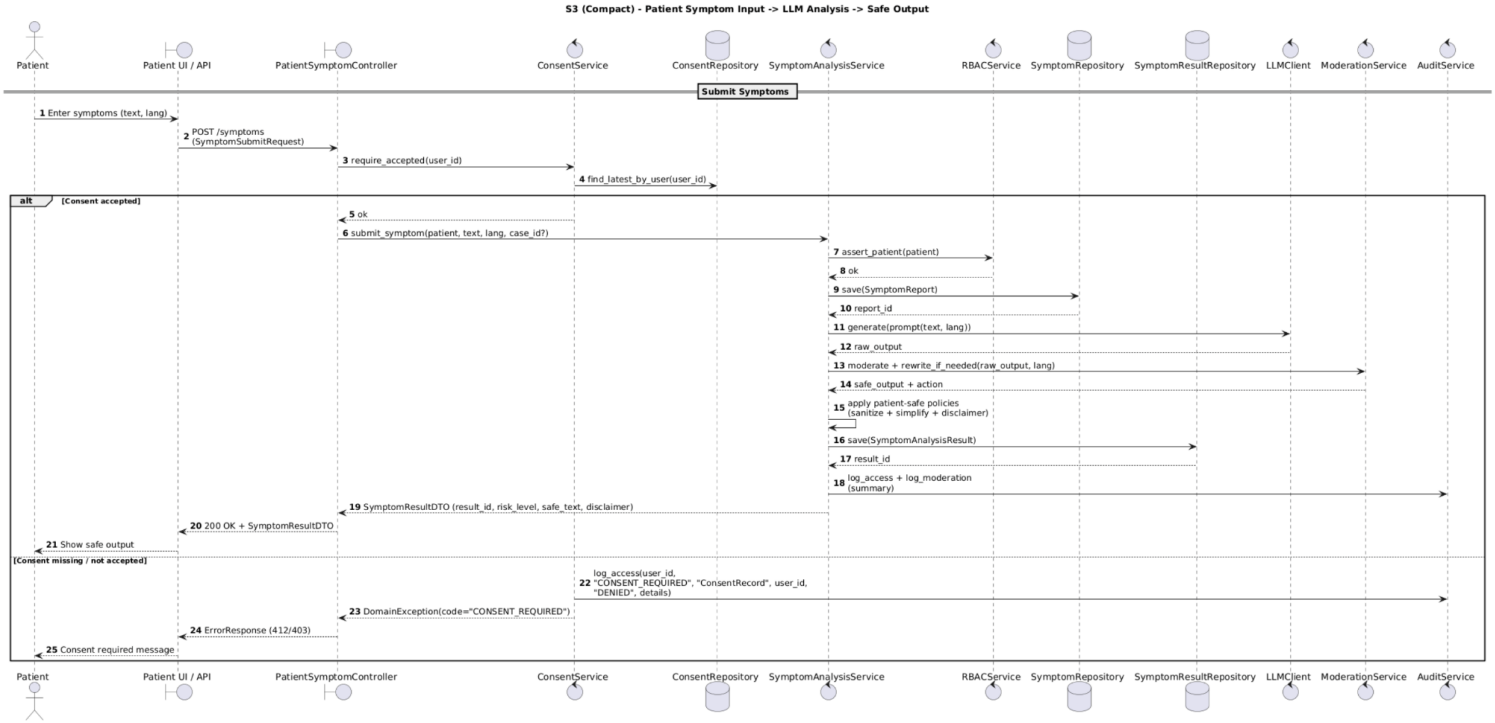
This sequence diagram represents the system health monitoring process. A health check request is handled by the MonitoringController, which invokes the HealthCheckService. The service collects metrics from critical components such as the model service and database, evaluates them against predefined thresholds, and determines the overall system status. If a failure or degradation is detected, an alert is generated and logged. Otherwise, the system status is recorded as healthy.



Patient Symptom Input → LLM Analysis → Safe Output (Compact) Sequence Diagram

This compact sequence diagram summarizes the patient symptom analysis workflow at a high level while preserving the core safety and compliance steps. A patient submits symptoms through the Patient UI, which forwards the request to the PatientSymptomController. Before any analysis is performed, the controller invokes the ConsentService to verify that the user has accepted the required consent; if consent is missing, the request is rejected and an audit event is recorded. When consent is valid, the SymptomAnalysisService orchestrates the process by validating patient authorization, persisting the SymptomReport, and calling the LLMClient to generate an initial analysis output. The generated content is then passed to the PatientTextModerationService, which decides whether the output can be allowed as-

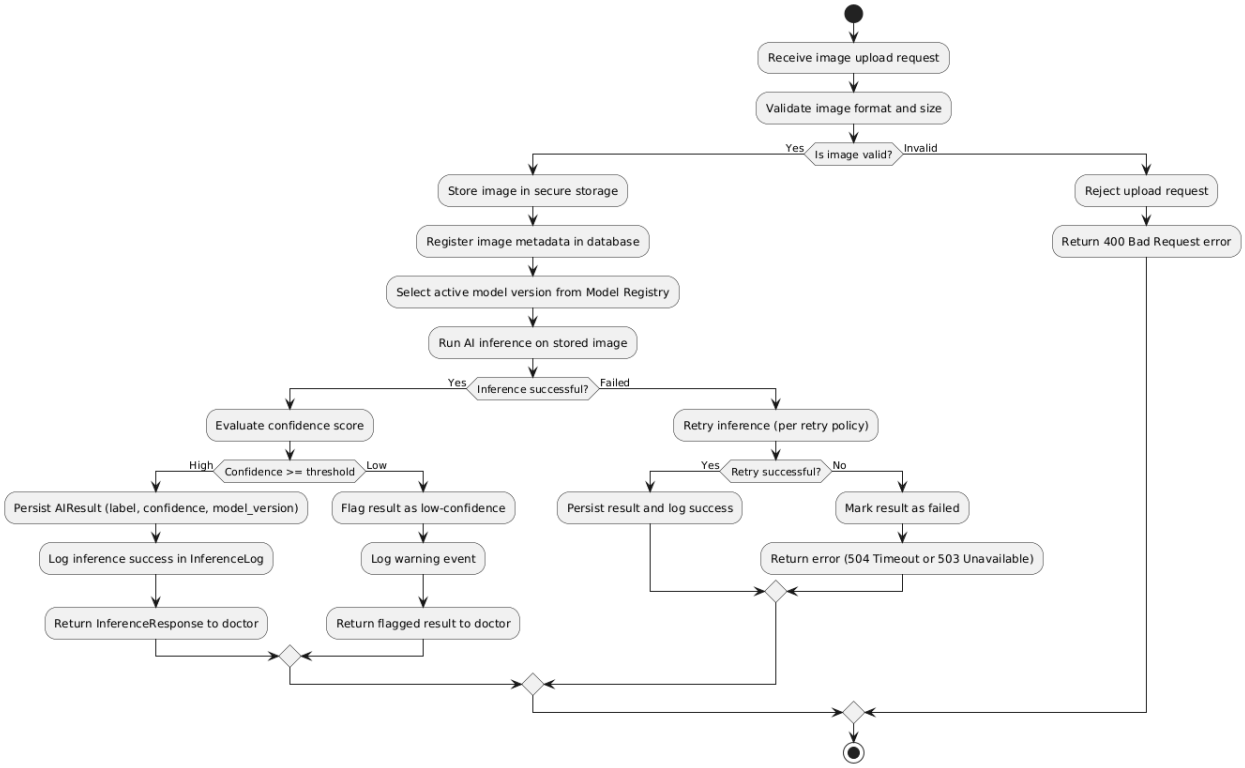
is, must be rewritten, or should be blocked. After moderation, the service applies patient-safe presentation rules (sanitization, simplification, and disclaimer attachment), stores the final SymptomAnalysisResult, logs a summary audit entry, and returns a safe response to the UI for display to the patient.



4.4. Activity Diagrams

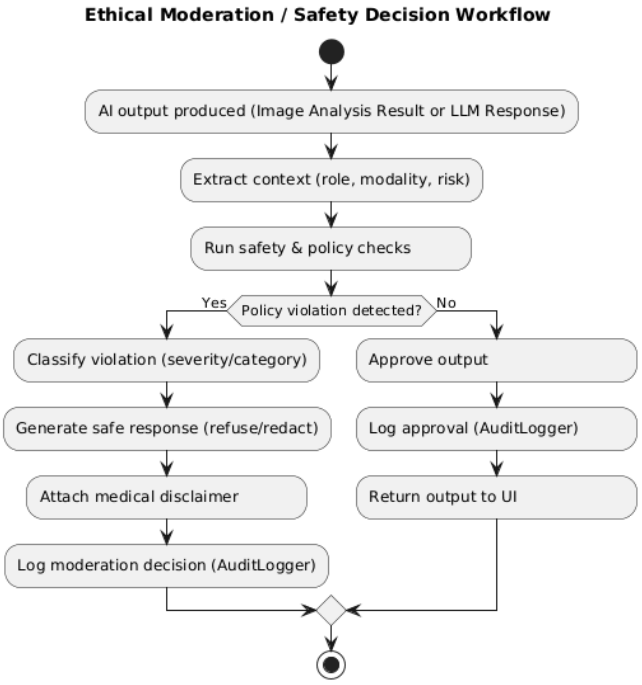
AI Inference Workflow

This activity diagram describes the operational flow for executing AI inference on a medical image uploaded by a doctor. The system receives the upload request, validates the image, and stores it securely. Then, the inference service selects the latest approved model version from the registry and performs the AI prediction. If the inference completes successfully, the result is persisted and returned to the doctor with a confidence score. If the model execution times out or the confidence level is too low, the system applies retry or flagging logic according to safety policies. All steps and errors are logged for traceability and monitoring.



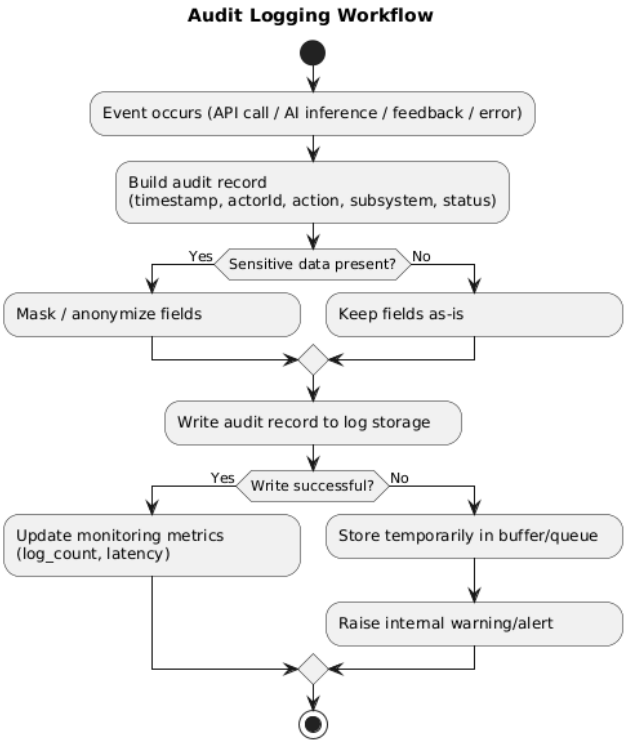
Ethical Moderation / Safety Decision Workflow

This activity diagram describes the safety decision flow applied before presenting AI-generated outputs to end users. The system evaluates the AI output with policy checks and risk-based rules using the request context. If a violation is detected, the response is redacted or replaced with a safe message and a medical disclaimer, and the decision is logged for traceability. If no violation is found, the output is approved and delivered to the UI while still being recorded in audit logs.



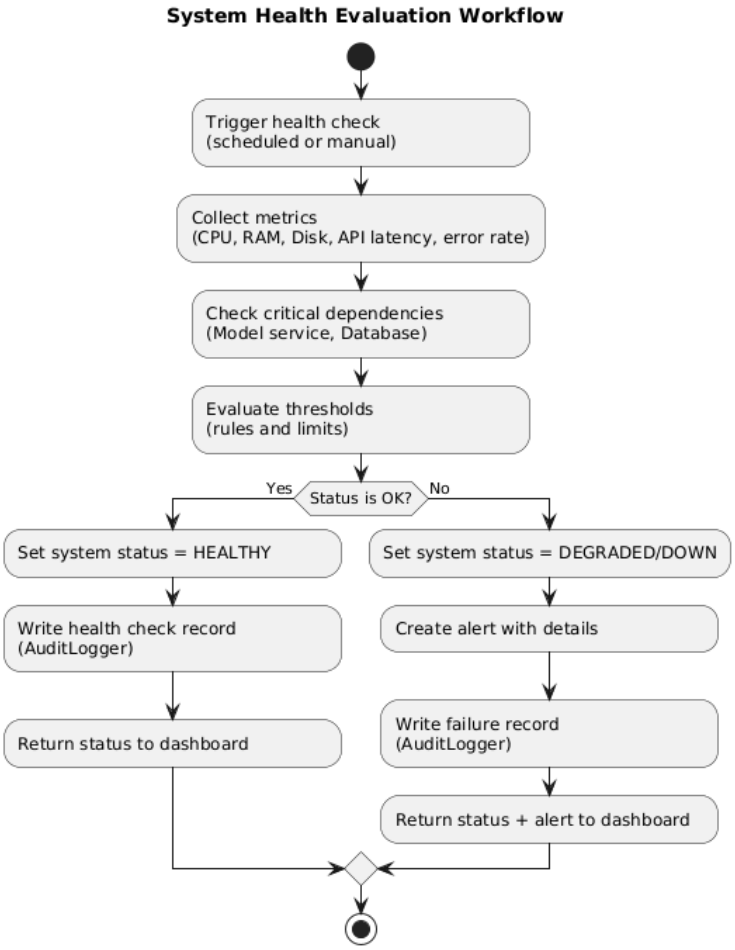
Audit Logging Workflow

This activity diagram describes how the system records auditable events for traceability and compliance. When a critical event occurs, an audit record is constructed with metadata such as timestamp, actor identity, action type, and subsystem context. If sensitive information is detected, the data is masked or anonymized before persistence. The record is then written to the log storage; in case of failure, it is buffered and an internal warning is generated to prevent loss of audit trails.



System Health Evaluation Workflow

This activity diagram describes how the system evaluates its operational health. A health check is triggered manually or on a schedule, after which runtime metrics and dependency statuses are collected. The metrics are compared against predefined thresholds to compute an overall system status. If the system is healthy, the status is recorded and returned to the monitoring dashboard. If degradation or failure is detected, an alert is created and the incident is logged to support rapid detection and troubleshooting.



5. Workflows

This section describes the low-level operational workflows implemented in the backend at the controller–service–repository level. The purpose of this section is not to replace the UML sequence and activity diagrams provided earlier, but to provide a concise textual explanation of the runtime execution logic reflected in those diagrams. The detailed interaction order, message passing, and alternative paths are explicitly visualized in the corresponding Sequence and Activity Diagrams; this section summarizes their implementation-level interpretation.

All workflows follow a structured execution pattern:

Controller method invocation → Input validation → Authorization & consent checks (if required) → Service-layer processing → Repository interaction (persistence / retrieval) → Moderation and safety enforcement (where applicable) → Audit logging and metrics recording → Exception handling (fail-safe) → Response generation

The system adopts a fail-safe design principle, especially for safety-sensitive subsystems such as medical inference and symptom analysis. Under no circumstance is unsafe model output returned directly to the end user.

At runtime, workflows operate as follows:

- Image Processing flows handle medical image uploads, trigger AI inference, apply confidence threshold evaluation, persist inference results, record inference logs, and generate structured responses for the doctor dashboard. Performance metrics and execution logs are recorded for traceability and monitoring.

- Symptom Analysis & LLM flows enforce consent validation prior to symptom submission, apply RBAC authorization, persist symptom reports, call the LLM provider within defined timeout constraints, execute the moderation and safety pipeline, append mandatory disclaimers, persist safe results, and record audit and latency metrics. In case of LLM failure, moderation block, or timeout, the system returns a conservative safe response without exposing raw model output.

- Result Retrieval flows validate identifiers, retrieve persisted entities via repository access, map domain entities to safe response DTOs, and log access actions for audit compliance.

- Feedback and Override flows validate doctor permissions, apply override policies, update AI result status, persist feedback records, and log override actions for full traceability. Unauthorized override attempts result in immediate rejection and security logging.

- Moderation flows evaluate generated outputs against predefined rule sets. If violations are detected, outputs are blocked or rewritten according to policy. If the moderation service itself fails, a conservative fallback response is returned to ensure safety.

- Audit and Logging flows record critical actions such as authentication events, overrides, moderation decisions, and inference executions. Persistence failures trigger retry mechanisms and fallback logging strategies.

- Monitoring and Alert workflows periodically collect system metrics such as inference latency, error rates, and LLM response times. Threshold breaches trigger alert creation and administrative notification through the AlertService.

All workflows described above directly correspond to the previously defined Sequence and Activity Diagrams, which formally illustrate the interaction order, branching logic, and exception paths. This section provides the method-level operational summary of those diagrams while preserving traceability between design artifacts and implementation logic.

6. Data Structures & Persistence

This section defines the persistence model of the system at a structural level. While the detailed attribute definitions and data types are specified in Section 3 (Class Interfaces), this section focuses on how those domain objects are stored, related, indexed, and constrained in the database layer.

The goal of this section is to clarify:

- Which entities are persisted
- How entities are related
- Which constraints enforce data integrity
- How indexing supports performance requirements
- How immutability and traceability principles are applied

All timestamps are stored in UTC format.

Audit, moderation, and consent-related records follow an append-only policy to preserve historical traceability.

The system persists entities across the following logical domains:

- Image & AI Inference Data
- Symptom & Chat Data
- Moderation & Logging Data
- User Management & Governance Data

Image & AI Inference Persistence

Persisted entities include: MedicalImage, AIResult, ModelVersion.
MedicalImage records are linked to the uploading user and optionally to a medical case.
AIResult references MedicalImage and stores prediction outputs and lifecycle status.
ModelVersion provides traceability between inference results and deployed model versions.

Indexing focuses on: image_id (PK), case_id, uploader_id, result_id, created_at timestamps
AI results are immutable after creation, except for controlled status updates (e.g., override).
Image files themselves are stored externally (file storage), while metadata is persisted in the database.

Symptom & Chat Persistence

Persisted entities include: SymptomReport, SymptomAnalysisResult, ConsentRecord.
SymptomReport references the patient (User) and optionally a medical case.
SymptomAnalysisResult references SymptomReport and stores only moderated, patient-safe outputs.
ConsentRecord references User and enforces consent acceptance prior to symptom analysis.

Indexing supports: patient history queries (patient_id, created_at), result lookup (report_id), latest consent lookup (user_id, accepted_at)

Symptom reports preserve raw patient input for traceability and are immutable after creation.

Symptom analysis results store only post-moderation, disclaimer-attached content.
Consent records are append-only; acceptance state is derived from the latest record.

Moderation & Logging Persistence

Persisted entities include: ModerationRecord, AuditLog, SystemHealthMetric, AlertEvent.
ModerationRecord stores safety decisions associated with inference or LLM outputs.
AuditLog records critical user and system actions (authentication, overrides, moderation decisions).
SystemHealthMetric captures periodic monitoring data.
AlertEvent represents threshold-based alerts triggered by monitoring logic.

Indexing supports: entity-based traceability (related_entity_id), user-based audit lookup (user_id, timestamp), subsystem-based monitoring queries

Audit logs and moderation records are immutable and append-only, health metrics behave as time-series records and alert events are updated only when resolved.

User Management & Governance Persistence

Persisted entities include: User, SessionToken.
User stores identity and role information used for RBAC enforcement.
SessionToken maintains active authentication sessions with expiration logic.
Indexing focuses on: unique email constraint, user_id foreign key references, token expiration queries, authentication timestamp tracking
Password hashes are stored securely.
Session tokens are invalidated through expiration or explicit logout.
Authentication logs are append-only to ensure forensic traceability.

This persistence model ensures:

- Performance-aware indexing for common query paths
- Referential integrity across subsystems
- Immutable safety-critical records
- Full traceability for compliance and audit requirements

Together with the Class Interfaces section, this defines the complete low-level data structure and storage behavior of the system.

7. Error Handling & Edge Cases

This section summarizes the low-level error handling strategy applied across all subsystems. Detailed validation cases, test scenarios, and failure matrices are documented in the Test Plan Report; therefore, this section provides a concise design-level overview rather than exhaustive case-by-case descriptions.

The system follows a fail-safe principle, especially for safety-critical components such as medical inference, symptom analysis, and moderation. In situations of uncertainty, service failure, or unexpected behavior, the system prioritizes controlled output, traceability, and stability over permissive behavior.

Error handling is implemented at the controller and service levels using:

- Explicit exception classes per subsystem
- Controlled and standardized HTTP response codes
- Bounded retry mechanisms
- Structured audit logging for all critical failures
- Graceful degradation strategies
- Threshold-based alert triggering
- Idempotent recovery procedures where applicable

All error paths generate structured audit entries to ensure traceability and compliance.

The system addresses the following general edge-case categories:

- Invalid or malformed input (e.g., missing fields, unsupported language, invalid identifiers)
- Authorization and role mismatches (RBAC enforcement)
- Consent requirement violations (for symptom workflows)
- External service unavailability (LLM provider, moderation engine)
- Timeout and latency budget exceedance
- Threshold breaches in monitoring metrics
- Concurrent update conflicts (e.g., result overrides)
- Persistence failures (database write/read issues)

Under no condition is raw AI or LLM output returned without passing through the safety and moderation pipeline. If moderation or inference components fail, the system switches to SAFE_MODE behavior, returning a conservative, disclaimer-attached response or a controlled error payload.

For persistence failures, the system avoids returning non-traceable results. If audit logging fails, business workflows continue while fallback logging mechanisms are applied to ensure eventual consistency.

Monitoring-related errors (metrics collection interruption, alert delivery failure, threshold breaches) do not block core functionality. Instead, they trigger alerts, retry mechanisms, or degraded-status marking while preserving system availability.

Access control and governance errors (invalid role, unauthorized override, expired session, consent missing) result in immediate controlled rejection, with full audit trace generation.

This section defines the design philosophy and subsystem-level handling approach. Detailed test cases, validation matrices, retry limits, and edge-condition verification procedures are formally specified in the Test Plan Report.

8. Testability

This section summarizes the testability strategy of the system at the implementation level. Detailed test cases, coverage matrices, failure simulations, and validation tables are formally documented in the Test Plan Report; therefore, this section provides a structured overview of how the system is designed to support reliable and controlled testing.

The architecture follows a Controller–Service–Repository separation pattern, enabling clear isolation of responsibilities and independent verification of components. Testability is ensured through:

- Dependency injection for external services (LLM client, moderation model, database layer)
- Mockable external dependencies
- Explicit exception handling paths
- Configurable thresholds and timeout parameters
- Isolated persistence layers (test schemas)
- Deterministic behavior under controlled test conditions

All subsystems (Image Processing, LLM & Chat, Moderation & Monitoring, Governance) are validated through a combination of unit tests, integration tests, and controlled failure simulations.

The general testing strategy includes:

Unit Tests

Core services such as ConsentService, SymptomAnalysisService, PatientTextModerationService, and policy components (DisclaimerPolicy handling) are tested independently. External dependencies are mocked to ensure deterministic outputs and controlled exception scenarios.

Integration Tests

Controller–Service–Repository interaction flows are validated using isolated test database schemas. These tests verify correct persistence behavior, status transitions, moderation enforcement, and structured response generation.

Failure Simulation Tests

External dependency failures (LLM timeout, moderation failure, persistence errors, logging interruptions) are simulated through mocks. The system is verified to switch to SAFE_MODE behavior, return controlled outputs, and generate appropriate audit entries without exposing unsafe content.

Threshold & Latency Validation

Latency budgets (e.g., ≤5 seconds for LLM calls) and monitoring thresholds are validated using controlled timing mechanisms and mock metrics collectors. Retry policies are bounded and verified to avoid unbounded execution.

Governance & Access Control Tests

RBAC enforcement, consent gating, override authorization, and session validation are tested through role-based access simulations. Unauthorized access attempts must result in controlled HTTP responses and structured audit logs.

Moderation & Logging Validation

Moderation decisions (ALLOW / REWRITE / BLOCK) are validated under both normal and failure scenarios. Audit logging integrity is verified to ensure append-only behavior, correct timestamping, and non-blocking logging fallback mechanisms.

Test isolation principles applied across the system include:

- Mocking of all external providers (LLMClient, ModerationModelClient, notifier services)
- Fixed clock injection for deterministic timing tests
- Isolated database schemas for integration testing
- Verification that failure paths remain traceable via audit logging

Safety-sensitive paths (medical inference, symptom analysis, moderation) require explicit coverage of both success and fail-safe branches. No test scenario allows raw model output to bypass moderation and disclaimer enforcement.

Coverage targets and exhaustive scenario definitions are specified in the Test Plan Report. This section defines the architectural foundations that make those tests feasible and reliable.

9. Glossary

AI (Artificial Intelligence): Computational techniques used to perform automated medical image analysis and symptom text interpretation.

AIResult: The structured output produced by an AI inference process, including prediction label, confidence score, and status.

AuditLog: An immutable, append-only record of critical system actions (e.g., login, inference, moderation, override) used for traceability and compliance.

Consent: The explicit confirmation that a user has accepted required terms before accessing specific features such as symptom analysis.

DTO (Data Transfer Object): A structured object used to transfer data between system layers or across the API boundary without embedding business logic.

Inference: The execution of a trained AI model to generate predictions from medical images or text input.

LLD (Low-Level Design): A detailed system design specification describing class interfaces, data structures, workflows, and error handling logic.

LLM (Large Language Model): A neural language model used to analyze free-text symptom input and generate structured responses.

Moderation: The evaluation and transformation of AI-generated outputs to ensure safety, policy compliance, and appropriate user-facing communication.

RBAC (Role-Based Access Control): An authorization mechanism that restricts system actions based on predefined user roles.

SessionToken: A time-bound authentication token representing an active user session.

UML (Unified Modeling Language): A standardized modeling language used to represent system structure and behavior through diagrams such as class and sequence diagrams.

10. References

IEEE. IEEE Std 1016-2009: IEEE Standard for Information Technology—Systems Design—Software Design Descriptions. IEEE, 2009.

IEEE. IEEE Std 830-1998: IEEE Recommended Practice for Software Requirements Specifications. IEEE, 1998.

Object Management Group (OMG). Unified Modeling Language (UML) Specification (Version 2.5 or later). OMG.

ISO/IEC. ISO/IEC 25010: Systems and software engineering—Systems and software Quality Requirements and Evaluation (SQuaRE)—System and software quality models. ISO/IEC, 2011.

European Union. Regulation (EU) 2016/679 (General Data Protection Regulation—GDPR). Official Journal of the European Union, 2016.

Republic of Türkiye. Law No. 6698 on the Protection of Personal Data (KVKK). Official Gazette of the Republic of Türkiye, 2016.

U.S. Department of Health & Human Services (HHS). Health Insurance Portability and Accountability Act (HIPAA): Privacy, Security, and Breach Notification Rules.

Python Software Foundation. PEP 8 — Style Guide for Python Code.

PlantUML. PlantUML: Open-Source Tool for Generating UML Diagrams from Plain Text. PlantUML Project Documentation.