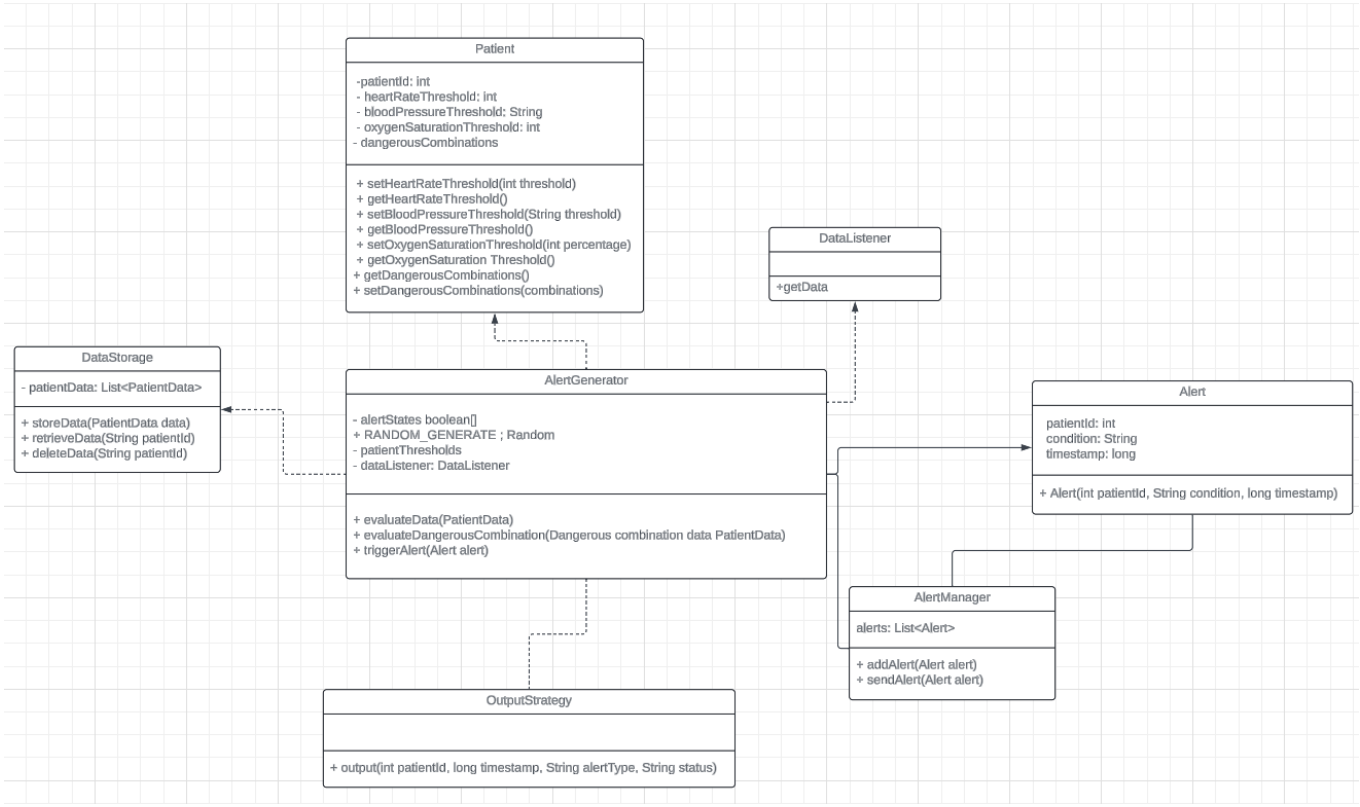
Alert Generation System



The diagram above shows how AlertGenerator and other classes works.

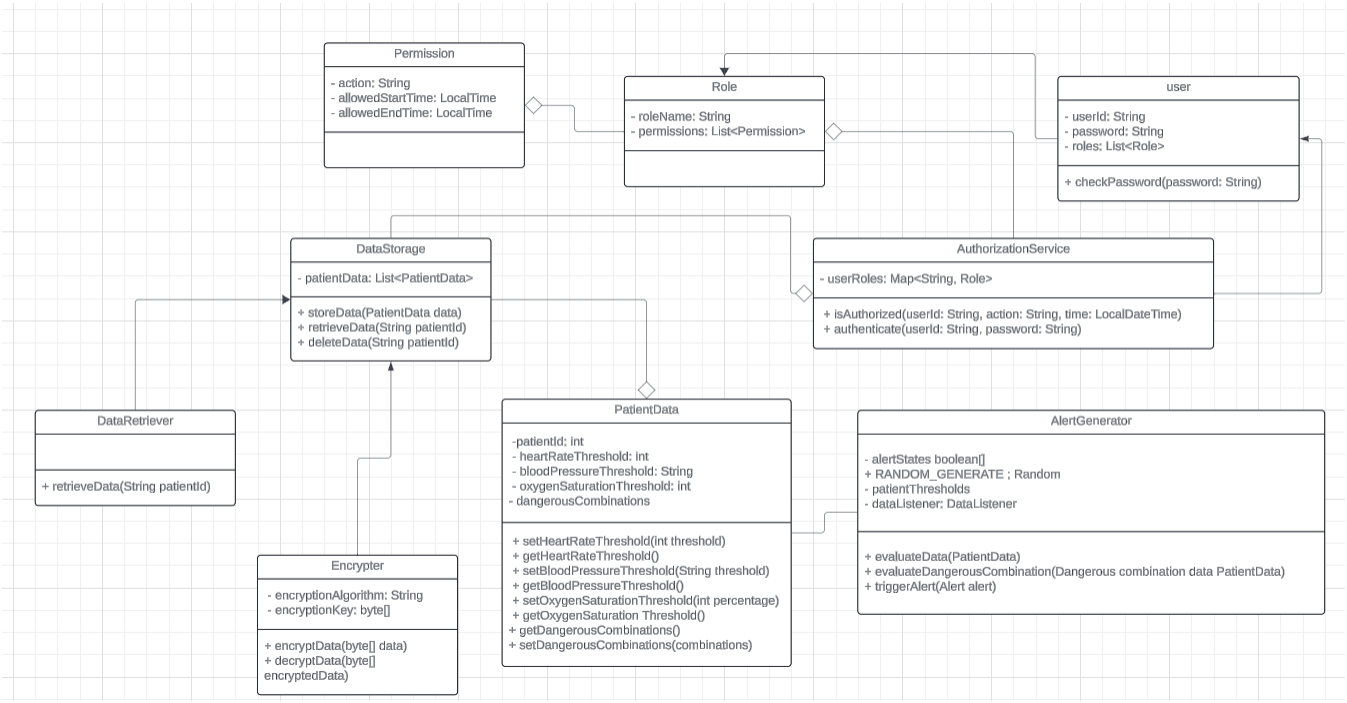
Patient objects are created with unique patient IDs. This way, it is possible to store each patient's heart rate, oxygen saturation, and blood pressure thresholds. Additionally, we can save their special conditions to keep track of them. The AlertGenerator constantly receives data from the DataListener and compares it with the patient's thresholds. It can also perform complex evaluations by accessing past records of the patient through the DataStorage class and by having information about special conditions provided by the Patient class.

After these evaluations, if any of the data points exceed the predefined thresholds or meet the dangerous combinations criteria, an Alert object is created by the AlertGenerator class. The Alert object includes the patient ID, the specific condition that triggered the alert, and a timestamp indicating when the alert was generated. The AlertGenerator then passes the Alert object to the AlertManager class. The AlertManager is responsible for managing these alerts, which ensuring the alert is sent to the appropriate medical staff.

The OutputStrategy class determines how the alert information should be presented and communicated. The OutputStrategy ensures that the alerts are delivered efficiently and effectively to the medical staff, enabling prompt response to the patient's condition.

By integrating these components, the system provides a comprehensive solution for monitoring and responding to patient health metrics, ensuring timely and accurate medical intervention when necessary.

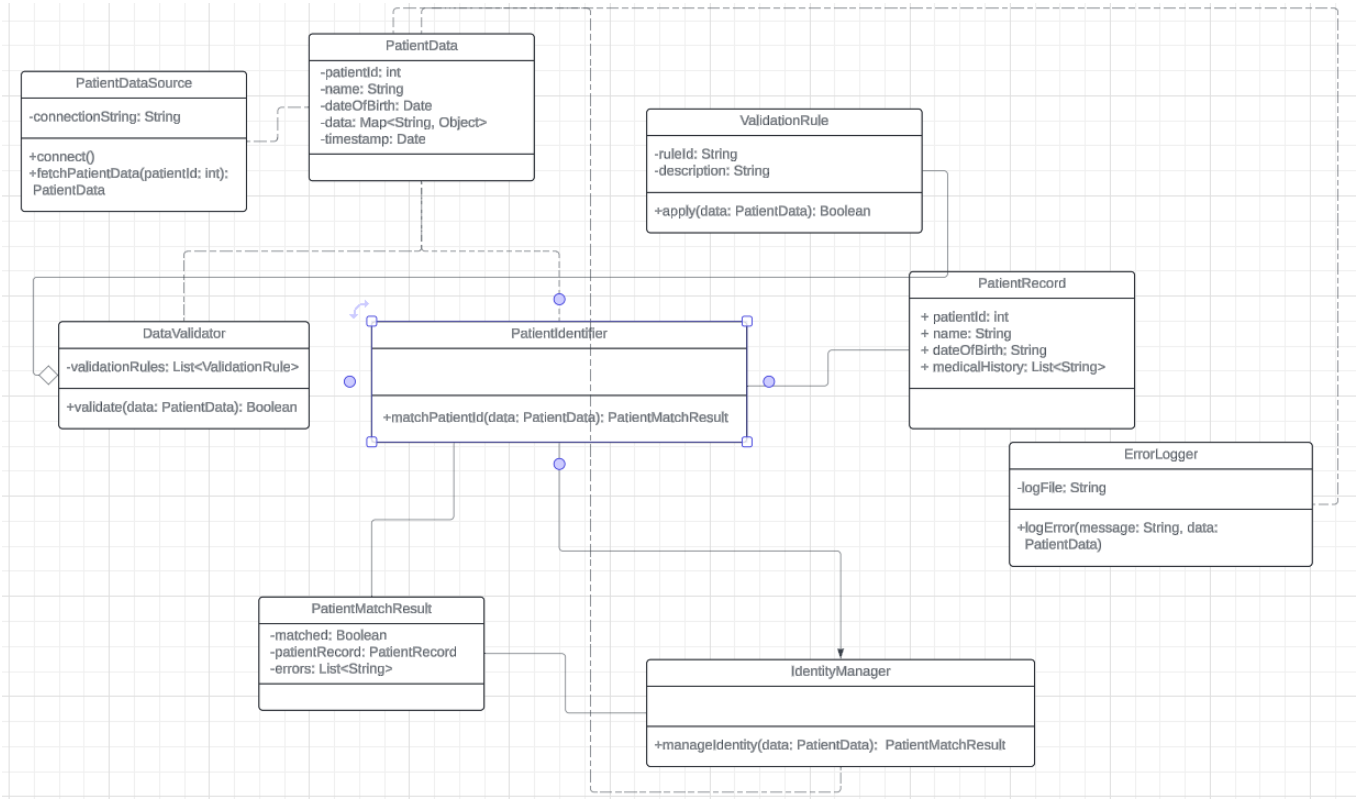
Data Storage System



The UML diagram represents a system for managing and monitoring patient data with a focus on security and alert generation. The Permission class defines specific actions that can be performed within a certain time frame, encapsulated by attributes such as action, allowedStartTime, and allowedEndTime. The Role class aggregates multiple Permission objects, specifying a roleName and a list of permissions, thereby defining what actions a user with a certain role can perform. Each User in the system has a unique userId, a password, and a list of roles, with methods for password validation (checkPassword). The AuthorizationService class is crucial for security, managing user roles and providing methods like isAuthorized and authenticate to check if a user is permitted to perform a specific action at a given time and to verify user credentials. The DataStorage class is responsible for handling patient data, offering methods to store, retrieve, and delete patient records through the storeData, retrieveData, and deleteData methods. Each patient's vital information and thresholds are encapsulated in the PatientData class, which includes attributes for heart rate, blood pressure, oxygen saturation thresholds, and dangerous

combinations. This class provides getter and setter methods to manipulate these thresholds. The AlertGenerator class plays a pivotal role in monitoring patient data. It has attributes for alert states and patient thresholds, and methods to evaluate data (evaluateData), assess dangerous combinations (evaluateDangerousCombination), and trigger alerts (triggerAlert). The Encrypter class ensures data security by encrypting and decrypting patient data using specified algorithms and keys. Finally, the DataRetriever class provides an interface to retrieve patient data. This comprehensive system ensures secure management, accurate monitoring, and timely alerting for patient health data.

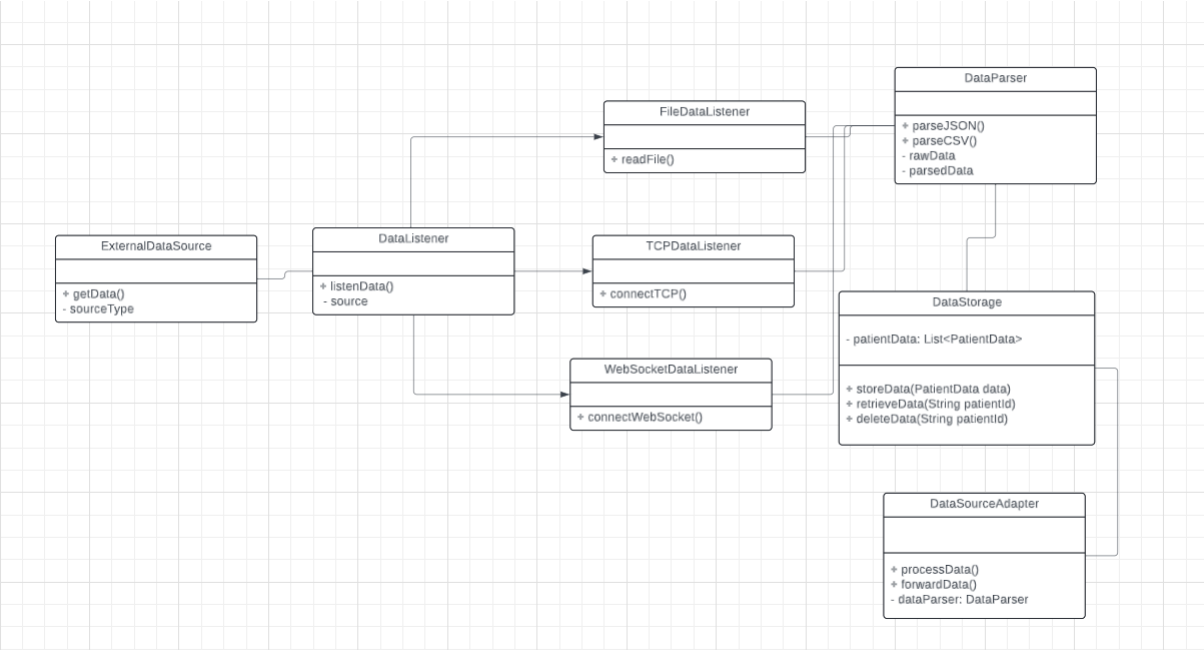
Patient Identification System



The Patient Identification System is designed to accurately link incoming patient data from various sources to the correct patient profiles within a hospital's main database, preventing misidentification and erroneous medical decisions. The system includes several interconnected classes, each with specific roles to ensure data integrity and correct patient identification. The PatientDataSource class connects to external data sources to fetch patient data using the fetchPatientData method. PatientData represents the incoming data structure with fields like patientId, name, dateOfBirth, data (a map of various data points), and timestamp. DataValidator ensures the accuracy and consistency of the incoming data by applying various validation rules through the validate method. ValidationRule defines individual checks, such as ensuring non-empty names or valid dates of birth. PatientIdentifier is the core

component that matches incoming PatientData with existing records using the matchPatientId method, returning a PatientMatchResult that indicates whether a match was found. PatientRecord represents existing patient records, including patientId, name, dateOfBirth, and medicalHistory, for comparison against incoming data. PatientMatchResult contains the matching process's result, indicating match status, the corresponding PatientRecord, and any errors. IdentityManager oversees the entire process, addressing discrepancies or anomalies using the manageIdentity method. ErrorLogger logs any errors or inconsistencies encountered during validation and matching processes with the logError method. Collectively, this system ensures that patient data streaming from various sources is accurately matched with the correct patient records, maintaining data integrity and supporting accurate medical decisions.

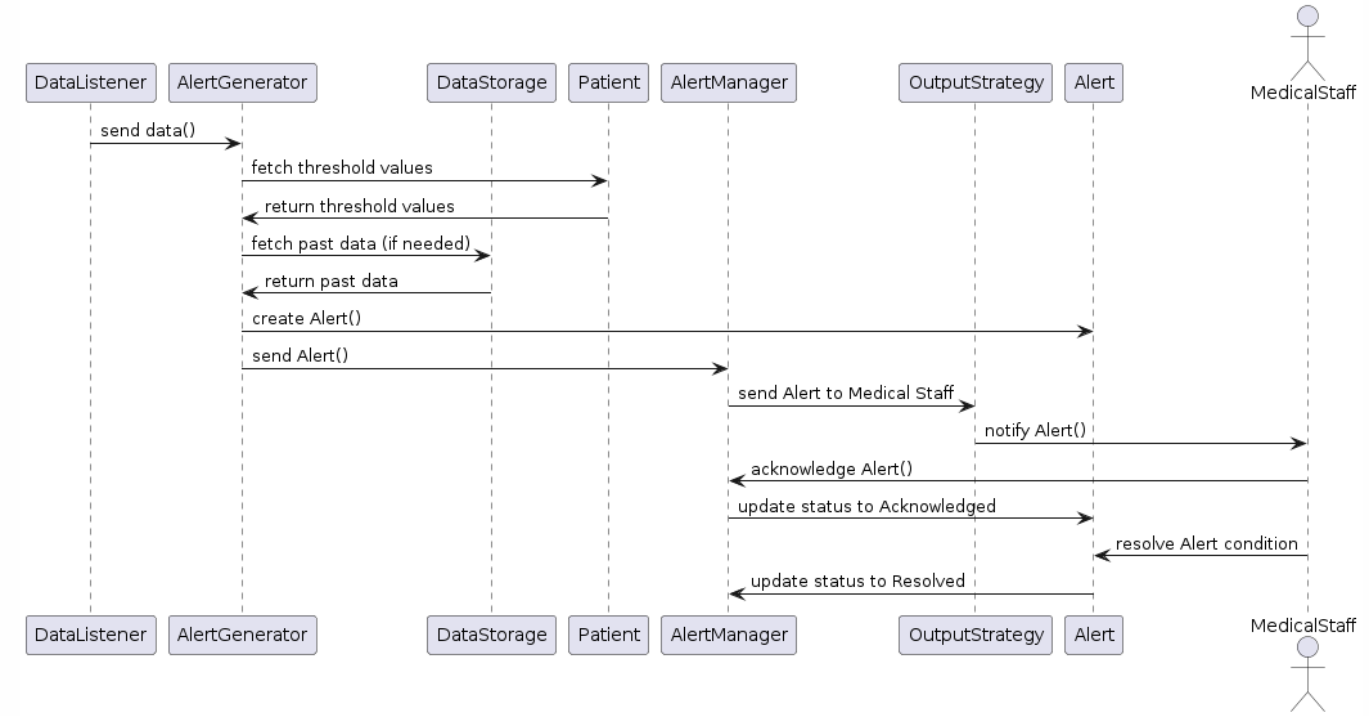
Data Access Layer



The system efficiently handles and processes data from various external sources, beginning with the ExternalDataSource class, which represents data from different origins. This data is passed to the DataListener class, a central component responsible for listening to incoming data, featuring three specific subclasses: FileDataListener, TCPDataListener, and WebSocketDataListener, each tailored to handle data from its respective source. For instance, FileDataListener reads data from files using its readFile() method, TCPDataListener manages TCP connections with the connectTCP() method, and WebSocketDataListener handles WebSocket connections via the connectWebSocket() method. Once the data is received by these specific listeners, it is forwarded to the DataParser class. The DataParser processes the raw data and converts it into a structured format using methods like

parseJSON() and parseCSV(), ensuring the data is in a consistent format suitable for storage. The processed data is then sent to the DataSourceAdapter, which further processes and prepares the data for storage. Finally, the DataSourceAdapter forwards the fully processed data to the DataStorage class, responsible for storing the data and providing methods for data management, including storeData(), retrieveData(), and deleteData(). This structured approach ensures seamless data handling from various sources, maintaining data integrity and accessibility from the initial source to final storage.

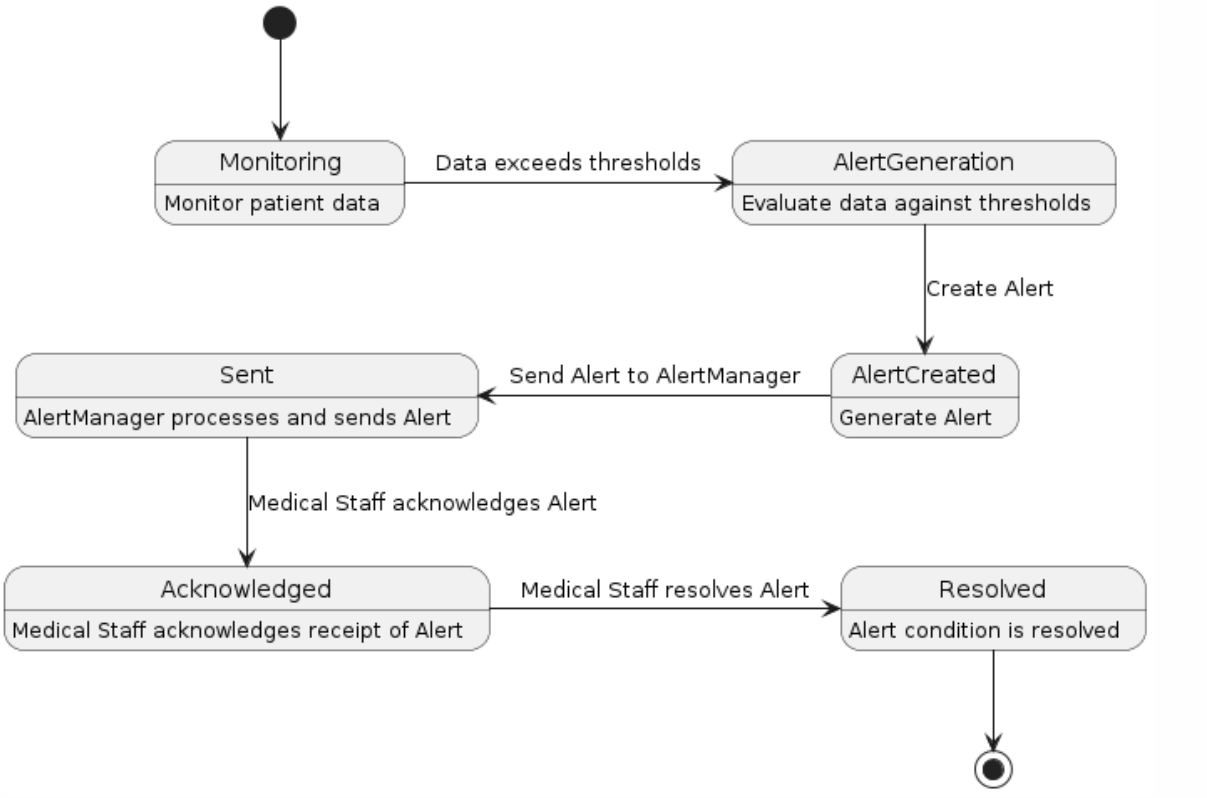
Sequence Diagram of AlertGenerator



The sequence diagram above depicts the interaction between various components involved in the alert generation and management system. It starts with the DataListener device sending patient data, such as heart rate, blood pressure, and oxygen saturation levels, to the AlertGenerator. The AlertGenerator then fetches the threshold values specific to the patient from the Patient class. If it is needed, it also retrieves past data from the DataStorage to perform a comprehensive evaluation. Upon receiving the necessary information, the AlertGenerator compares the incoming data against the patient's predefined thresholds. If any data points exceed these thresholds or meet dangerous combination criteria, the AlertGenerator creates a new Alert object, containing the patient ID, the specific condition, and a timestamp. This alert is then sent to the AlertManager. The AlertManager uses the OutputStrategy to ensure the alert is efficiently communicated to the medical staff.

The OutputStrategy notifies the medical staff, who then acknowledge the receipt of the alert by sending a response back to the AlertManager. The AlertManager updates the alert's status to "Acknowledged." Subsequently, the medical staff intervenes to resolve the patient's condition. Once resolved, the alert's status is updated to "Resolved," signifying the end of the alert lifecycle.

State Diagram of AlertGenerator



The state diagram illustrates the lifecycle of an alert in a medical monitoring system. The process starts in the Monitoring state, where patient data is continuously observed. When patient data exceeds predefined thresholds, the system transitions to the AlertGeneration state to evaluate the data against these thresholds. If the data meets the criteria for generating an alert, the system moves to the AlertCreated state, where an alert is created with the necessary details, including the patient ID, specific conditions, and a timestamp. The alert then transitions to the Sent state, where the AlertManager processes and sends it to the appropriate medical staff. Upon receiving the alert, the system enters the Acknowledged state as the medical staff acknowledges the alert. The final transition occurs when the medical staff resolves the alert condition, moving the system into the Resolved state. In this state, the alert condition is either automatically normalized or manually addressed by the medical staff, concluding the alert's lifecycle. This structured approach ensures continuous monitoring of patient health metrics and prompt response to critical conditions, thereby enhancing patient safety and care.