SOFTWARE DESIGN DESCRIPTION

WinterGreen

Version 1.0

Team CloudSmiths -

Jaimin Savaliya, Jaynesh Bhandari, Krunal Savaj, Nithin Murthy, Rajath Gokhale

2025-03-29

Team Member Document Review

Member Name	Signature or Initial	Reviewed on (Date)
Jaimin Savaliya	JS	28th March 2025
Jaynesh Bhandari	JB	28th March 2025
Krunal Savaj	KS	28th March 2025
Nithin Murthy	NM	28th March 2025
Rajath Gokhale	RG	28th March 2025

Table 1: Member Signature and Review Table

Document Revision History

Version	Changes	Date
1.0	First Draft	28th Mar 2025

Table 2: Document Revision History

Contents

1 Introduction

1.1 Purpose

This document describes the software design for the WinterGreen Healthcare Provider On-boarding Application. It outlines the architecture, design decisions, data design, interfaces, and component interactions that we will be implementing in the project. It reiterates the scope and requirements of the project taken into account to implement the project. The document also uses UML diagrams to illustrate different aspects of the project. This document is intended for developers, testers, project managers, and system administrators.

1.2 Scope

The scope of our project is as follows:

- Creating a Web-App front-end to automate and simplify the existing manual work of on-boarding a HealthCare provider utilizing already defined AWS Lambda functions and step functions as the back-end that uses the healthcare provider data to fetch data from their EHR and processes the data through AWS healthlake and stores the data in AWS S3.
- Improving existing Lambda functions and Step functions, making them more modular, and adding functionality to access MySQL database to store and read data.
- Adding support in the system to handle more EHRs, namely eCW and Athena health.
- Adding functionality in the system to auto-refresh healthcare provider data after every 30 days.
- Creation of an interface to add support for new EHR in system without having to edit the code.
- Documenting the complete project to provide comprehensive details about the system to allow future upgrades changes and de-bugging.

1.3 Definitions, Acronyms and Abbreviations

- **HealthCare Provider:** Clients of WinterGreen (Example - Clinics or Hospitals).
- **AWS:** Amazon Web Services, a suite of cloud computing services.
- **EHR:** Electronic Health Record, a digital version of a patient's medical history.
- **FHIR:** Fast Healthcare Interoperability Resources, a standard for electronic healthcare data exchange.
- **HIPAA:** Health Insurance Portability and Accountability Act, U.S. national standards for protecting the privacy and security of patient health information.
- **Cerner:** Cerner, an Electronic Health Record platform.
- **Lambda:** AWS Lambda is a compute service that runs code in response to events and automatically manages the compute resources.
- **RDS:** Relational Database Service, a web service that makes it easier to set up, operate, and scale a relational database in the AWS Cloud.
- **eCW:** eClinicalWorks, an Electronic Health Record platform.
- **AWS S3 Bucket:** Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance.
- **Healthlake:** AWS HealthLake is a HIPAA-eligible service enabling healthcare and life sciences (HCLS) companies to securely store and transform their data into a consistent and query-able fashion.
- **IAM:** Identity and Access Management, a web service that helps you securely control access to AWS resources.
- **AWS Amplify:** AWS Amplify is a collection of cloud services and libraries for fullstack application development.

- **AWS API gateway:** Amazon API Gateway is an AWS service for creating, publishing, maintaining, monitoring, and securing REST, HTTP, and WebSocket APIs at any scale.
- **AWS Cognito:** AWS Cognito provides secure and scalable customer identity and access management that is enterprise-grade, cost-effective, and customizable.
- **AWS DynamoDB:** Amazon DynamoDB is a fast, highly scalable non-relational database service.
- **AWS Step-function:** Step Functions is a visual workflow service that helps developers use AWS services to build distributed applications, automate processes, orchestrate microservices, and create data pipeline.

1.4 References

1. "IEEE Standard for Information Technology–Systems Design–Software Design Descriptions". In: *IEEE STD 1016-2009* (2009), pp. 1–35. DOI: 10.1109/IEEESTD.2009.5167255
2. Amazon Web Services documentation - https://docs.aws.amazon.com

2 Requirements

This section outlines the system requirements as derived from the SRS document. Key requirements include the following functional and non-functional requirements:

2.1 Functional Requirements

2.1.1 Input and Storage of Healthcare Provider and EHR Details

Description : The system should allow authorized users to input healthcare provider location details along with their respective EHR system information (Cerner, eCW or Athena health). This data will be validated and stored in AWS RDS database for further processing.

Priority: High

Stimulus: An authorized user enters healthcare provider details along with the respective EHR information via the system's user interface.

Response: The system validates the input for completeness and correctness before storing it in the RDS database. If validation fails, an error message is returned with the appropriate details asking user to verify and re-enter the details.

2.1.2 Fetching Data from EHR Systems

Description : The system should fetch healthcare data from the specified EHR systems associated with each healthcare provider. This ensures accurate and up-to-date information is retrieved for processing.

Priority: High

Stimulus:

Case 1: The system initiates a request to fetch data from an EHR when a new entry is added in the System.

Case 2: When a scheduled refresh occurs.

Response: The system establishes a connection with the respective EHR system, retrieves the relevant data, stores it in AWS S3 bucket. If the EHR system is not supported, an error is logged, and no data is fetched.

2.1.3 Processing and Storing Fetched Data

Description : Once data is retrieved from an EHR system, it should be processed to ensure consistency. Processing includes structuring the data for further analysis.

Priority: High

Stimulus: New data is fetched from an EHR system.

Response: The system processes the data using AWS Healthlake, and then stores it back in AWS S3 bucket. If processing fails, an error is logged, and the issue is reported.

2.1.4 Scheduled Data Refresh Every 30 Days

Description : To ensure stored data remains up-to-date, the system shall automatically refresh data from the respective EHR system every 30 days. This refresh will overwrite outdated information with the latest available data.

Priority: Low

Stimulus: A scheduled task triggers a data refresh process every 30 days.

Response: The system fetches latest data from the EHR systems for all stored healthcare providers, processes it, and updates the S3 bucket. If a provider's EHR data is unavailable, an error is logged for further review.

2.1.5 Logging and Error Handling

Description: The system should maintain logs for all data retrieval, processing, and storage operations. Any failures, including validation errors, fetch failures, or processing issues, shall be logged and reported.

Priority: Medium

Stimulus: Any operation within the system (example - data submission, fetch request, processing task) encounters an error or completes successfully.

Response: The system logs details of the operation, including timestamps, status, and error details.

2.2 Non-functional Requirements

2.2.1 Performance and Scalability

- The system should be able to at least process 100 concurrent requests for data input, fetching and processing without affecting the performance.
- Data fetching from EHR should complete within 45 seconds.
- Provide clear and concise code documentation, making the code more maintainable.

2.2.2 Security and Compliance

- Authentication and authorization should be managed using AWS IAM roles and policies, ensuring that only authorized users and services can access sensitive data.
- Encrypt all data to maintain data access and control.

2.2.3 Data Integrity

- The system must validate all verifiable input data before storing it.
- Duplicate entries should be automatically detected and flagged, preventing data duplication.

3 Data Design

Data in the application is stored in following ways:

- AWS RDS MySQL inStance
- AWS S3 Bucket
- AWS Secret Manager

The following diagram helps understand the flow and storage of data in the system.

3.1 Data Flow Diagram

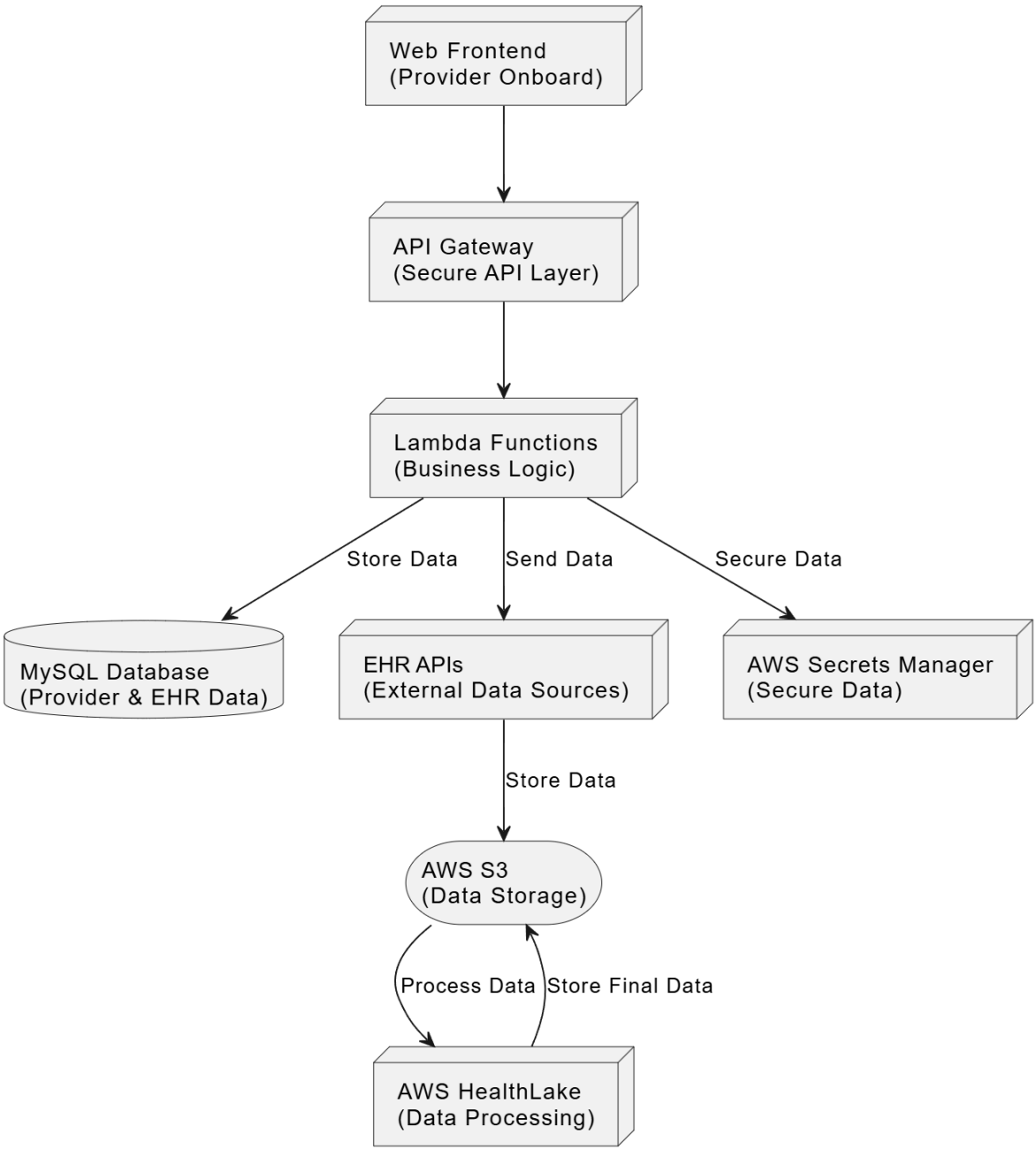

Figure 1: Data Flow Diagram

3.2 Database Schema Design

3.2.1 Healthcare provider Table

Table 3: healthcare_providers Table Schema

Column Name	Data Type	Description
provider_id	VARCHAR(36)	Primary Key, Unique identifier for the healthcare provider
provider_name	VARCHAR(255)	Name of the healthcare provider organization
provider_type	ENUM	Type of healthcare facility (Hospital, Clinic, Private Practice, Specialist Center, Other)
contact_email	VARCHAR(255)	Primary contact email address
contact_phone	VARCHAR(20)	Contact phone number
address	TEXT	Physical address of the provider
ehr_id	VARCHAR(36)	Foreign key to ehr_systems table
ehr_tenant_id	VARCHAR(255)	Tenant ID for multi-tenant EHR systems
ehr_group_id	VARCHAR(255)	Group ID of the group data to be fetched from EHR systems
secrets_manager_arn	VARCHAR(255)	AWS Secrets Manager ARN storing EHR credentials
onboarded_date	TIMESTAMP	Date when provider was onboarded into the system
last_data_fetch	TIMESTAMP	When provider data was last fetched from EHR
status	ENUM	Current status (Active, Inactive, Pending, Error)
notes	TEXT	Additional information about the provider

3.2.2 EHR Systems Table

Table 4: ehr_systems Table Schema

Column Name	Data Type	Description
ehr_id	VARCHAR(36)	Primary Key, Unique identifier for the EHR system
ehr_name	VARCHAR(255)	Name of the EHR system (must be unique)
api_base_endpoint	VARCHAR(255)	Base URL for the EHR API
description	TEXT	Description of the EHR system
is_supported	BOOLEAN	Whether the EHR is actively supported
added_on	TIMESTAMP	When the EHR was added to the system
last_updated	TIMESTAMP	When the EHR config was last updated

3.2.3 Data Fetch History Table

Table 5: data_fetch_history Table Schema

Column Name	Data Type	Description
fetch_id	VARCHAR(36)	Primary Key, Unique identifier for the fetch operation
provider_id	VARCHAR(36)	Foreign key to healthcare_providers table
fetch_time	TIMESTAMP	When the fetch operation completed
status	ENUM	Operation status (Success, Partial, Failed)
s3_location	VARCHAR(255)	S3 path where fetched data is stored
error_details	TEXT	Details of any errors encountered

3.3 Database Relationships

Table 6: Database Table Relationships

Primary Table	Related Table	Relationship
healthcare_providers.provider_id	data_fetch_history.provider_id	One-to-Many
healthcare_providers.ehr_id	ehr_systems.ehr_id	Many-to-One

4 Architecture

The architecture is a serverless AWS solution for a secure healthcare provider on-boarding platform that allows authenticated users to onboard clients, store their details in a database, and fetch data from their external EHR on demand or automatically every 30 days after the last fetch. The application back-end is completely based on Lambda functions, which allows the serverless architecture. The Lambda functions also enables the system to scale up and down based on the usage.

5 Component and Object Oriented Design

This section details the modular components of the system.

5.1 Web Frontend Components

5.1.1 Authentication Module

- **Purpose:** Manages user authorization
- **Key Components:**
  - Login Component
  - Session Manager
- **Technologies:** React, AWS Amplify SDK
- **Interfaces:**
  - Integrates with AWS Cognito for authentication flows

5.1.2 Healthcare Provider Management Module

- **Purpose:** Handles healthcare provider onboarding and management
- **Key Components:**
  - Healthcare Provider Onboarding Form
  - Healthcare Provider EHR Info Form
  - Healthcare Provider List Component
  - Healthcare Provider Search Component
- **Technologies:** React, React-Hook-Form
- **Interfaces:**
  - Consumes Provider API endpoints
  - Form validation with schema-based approach

5.1.3 Common UI Components

- **Purpose:** Reusing shared UI elements for consistent user experience
- **Key Components:**
  - Navigation Bar
  - Form Controls
- **Technologies:** React, CSS-in-JS
- **Interfaces:**
  - Consistent interfaces
  - Theme provider for styling

5.2 Backend Lambda Functions

5.2.1 Healthcare Provider Management Functions

- **Purpose:** CRUD operations for healthcare provider records
- **Functions:**
  - `createProvider`: Creates new provider record

  - `getProvider`: Retrieves provider details
  - `updateProvider`: Updates provider information
  - `listProviders`: Retrieves filtered provider list
  - `addProviderEHR`: Updates provider EHR information
- **Technologies:** Node.js, Python, AWS SDK
- **Interfaces:**
  - REST API endpoints via API Gateway
  - RDS connections for database operations

5.2.2 EHR Integration Functions

- **Purpose:** Connects to external EHR systems and fetches data
- **Functions:**
  - `connectToEhr`: Establishes connection to EHR API
  - `fetchEhrData`: Extracts data from EHR
- **Technologies:** Node.js, Python, AWS SDK
- **Interfaces:**
  - Retrieves Secrets Manager for credentials
  - Standardized response structure with error handling

5.2.3 Data Processing Functions

- **Purpose:** Processes and transforms healthcare data
- **Functions:**
  - `processHealthData`: Handles data normalization on AWS HealthLake and then sends data to S3 storage
- **Technologies:** Node.js, Python, AWS SDK
- **Interfaces:**
  - S3 operations for data storage
  - HealthLake API integration

5.3 Data Storage Components

5.3.1 Relational Database (MySQL)

- **Purpose:** Stores structured application data
- **Tables:** As defined in the schema section
- **Key Features:**
  - Transactions for data integrity
  - Foreign key relationships
  - Query optimization
- **Interfaces:**
  - Connection pooling for Lambda functions
  - Structured query patterns

5.3.2 Secret Store

- **Purpose:** Securely manages sensitive credentials
- **Storage Structure:**
  - Provider-specific secrets
  - Service account credentials
- **Key Features:**
  - Encryption at rest
  - Automatic rotation
  - Access logging
- **Interfaces:**
  - SDK access from Lambda functions
  - JSON structure for credentials

5.3.3 Object Storage

- **Purpose:** Stores healthcare data files
- **Organization:**
  - Provider-specific buckets/folders
- **Key Features:**
  - Encryption at rest
  - Lifecycle policies
  - Access controls
- **Interfaces:**
  - S3 API for operations
  - Event notifications for processing

5.4 API Interface Design

5.4.1 Provider Management API

- **Endpoints:**
  - `POST /providers`: Create provider
  - `GET /providers/{id}`: Get provider details
  - `PUT /providers/{id}`: Update provider
  - `GET /providers`: List providers with filtering
- **Authentication:** JWT token from Cognito
- **Request/Response Format:** JSON
- **Error Handling:** Standard HTTP status codes with error messages

5.4.2 Data Refresh API

- **Endpoints:**
  - `POST /providers/{id}/refresh`: Trigger manual refresh
  - `GET /providers/{id}/refresh-history`: Get refresh history
- **Authentication:** JWT token from Cognito

- **Request/Response Format:** JSON
- **Error Handling:** Standard HTTP status codes with error messages

5.5 Integration Points

5.5.1 EHR API Integration

- **Systems Supported:**
  - Cerner API
  - eClinicalWorks API
  - Athena Health API
- **Integration Method:** REST API calls
- **Authentication Methods:**
  - Auth for Cerner
  - Auth for eCW
  - Auth for Athena Health
- **Data Exchange Format:** JSON

6 Interface Design

6.1 User Interface Design

6.1.1 Healthcare Provider Onboarding Interface

- **Healthcare Provider Details Form**: Captures provider information with validation.
- **EHR Selection & Configuration**: Dynamic form for connecting to EHR systems.

6.1.2 Data Refresh Interfaces

- **Refresh Status Tracker**: Monitors refresh operations with logs.
- **Refresh Schedule Manager**: List view for managing auto-refresh schedules.

6.2 API Interface Design

- **Healthcare Provider Management API**: CRUD endpoints for providers
- **Data Refresh API**: Trigger and monitor refresh operations
- **Authentication**: JWT-based with role permissions
- **Error Handling**: Standardized responses with codes and details

6.3 AWS Service Interfaces

- **HealthLake Integration**: FHIR resource import/query
- **S3 Storage Structure**: Organized by Healthcare provider/date
- **DynamoDB Refresh Schedule**: TTL-based triggering

7 Pattern Used

7.1 Creational Patterns

- **Builder Pattern**: Implements stepwise construction of complex provider onboarding requests.

7.2 Structural Patterns

- **Facade**: Simplifies interaction with the complex subsystems (HealthLake, EHR APIs).

7.3 Behavioral Patterns

- **State**: Manages Healthcare provider on-boarding workflow transitions.

8 Design Concept Review

8.1 Abstraction

The healthcare provider onboarding system uses abstraction by using independent components. This approach simplifies development and maintenance.

8.2 Architecture

The system follows a serverless AWS architecture with React frontend hosted on AWS Amplify, API Gateway for request routing, Lambda functions for business logic, and Step Functions for workflow orchestration. Additionally, this approach enables automatic 30-day data refresh cycles through DynamoDB TTL triggers.

8.3 Patterns

Key design patterns include Builder Pattern construction of complex provider onboarding requests, Facade for simplifying interaction between compex subsystems, and State for managing on-boarding workflow transitions.

8.4 Separation of Concerns

Clear boundaries exist between provider management, EHR integration, and data processing components. The frontend handles data collection and visualization, middleware manages authentication and request validation, while backend Lambda functions handle specific business operations without overlapping responsibilities.

8.5 Modularity

The system achieves modularity through self-contained AWS Lambda functions for specific tasks (provider creation, EHR connection, data fetching). React components are organized by functionality, and EHR configurations are stored as database records rather than hard-coded, allowing new EHRs to be added without code changes.

8.6 Information Hiding

Sensitive EHR API credentials are stored in AWS Secrets Manager, referenced only by ARN in the database. Internal data transformation logic is encapsulated within specialized Lambda functions, and the web application remains unaware of how data refreshes are scheduled and executed in the backend.

8.7 Functional Independence

Each Lambda function performs a focused task, such as creating a provider record or refreshing EHR data. Functions communicate via events and defined interfaces rather than shared state. Step Functions perform these independent components while maintaining loose coupling between them.

8.8 Refinement

The system refines the complex healthcare data integration process into manageable steps: provider onboarding, credential verification, initial data fetch, and scheduled refreshes. Each component is further refined into specific operations with well-defined inputs and outputs.

8.9 Aspects

Cross-cutting concerns including logging , authentication (Cognito), error handling, and monitoring are implemented consistently across the application. These aspects provide centralized functionality without cluttering the core business logic.

8.10 Refactoring

The database-driven approach to EHR configuration facilitates refactoring, as new fields or EHR systems can be added without code changes. The clean separation between UI, API, and backend services allows each component to evolve independently without affecting others.

8.11 Object-Oriented Design Concepts

The application leverages inheritance for specialized EHR connectors and polymorphism to handle different EHR systems through a common interface. Encapsulation help bundle together healthcare provider data and operations, while composition builds complex data processing pipelines from simpler components.

8.12 Design Classes

The system incorporates entity classes (Provider, EHR System) and infrastructure classes (logging, authentication) to organize code according to responsibilities.

8.13 Dependency Inversion

High-level business logic depends on EHR interface abstractions rather than concrete implementations.

8.14 Design for Test

The system facilitates testing through functions in Lambda handlers and mocking end points for external services. The separation of concerns enables component testing in isolation, while Step Functions support testing of workflow scenarios including error conditions.

9 Architectural Design Considerations

9.1 Economy

The healthcare provider onboarding system uses serverless AWS services to eliminate infrastructure management. Single-purpose Lambda functions and a configuration-driven approach for EHR systems reduce complexity while maintaining essential functionality. The system centralizes common operations and minimizes code duplication.

9.2 Visibility

Step Functions provide visual workflow representations, enhancing system transparency. API contracts are explicitly documented, and loggers will record all the logs and errors in the system. Each component has a well-defined purpose with clear inputs and outputs, making the system comprehensible for all stakeholders.

9.3 Spacing

The system cleanly separates the front end (React / amplify), API layer (API Gateway), and back end (Lambda / step functions). Lambda functions are organized by domain to avoid entanglement of features. Data storage is appropriately divided between RDS (structured provider data) and DynamoDB (refresh scheduling), eliminating hidden dependencies.

9.4 Symmetry

All EHR integrations follow consistent interface patterns despite underlying API differences. Error handling, authentication, and data refresh processes implement uniform approaches throughout the system. This symmetry creates predictability for developers and simplifies maintenance through recognizable patterns.

9.5 Emergence

The database-driven EHR configuration supports adding new systems without code changes. Modular Lambda functions can be recombined for new workflows as requirements evolve. The event-driven architecture enables new components to subscribe to existing events without modifying publishers, allowing the system to adapt to changing healthcare integration needs.

10 Component Design Principles

10.1 Open-Closed Principle

The healthcare provider onboarding system implements the open-closed principle through its EHR integration layer. The core EHR connector interface is closed for modification but open for extension through database-driven configurations. New EHR systems can be added without changing existing code, as connection details and field mappings are stored as data rather than hardcoded logic.

10.2 Liskov Substitution Principle

Different EHR connector implementations (Cerner, eClinicalWorks, Athena Health) function interchangeably through a common interface. Each connector adheres to the same contract, ensuring that any EHR implementation can be substituted without affecting the data fetch operations or provider onboarding workflow. This allows transparent switching between EHR systems when needed.

10.3 Dependency Inversion Principle

Lambda functions work with abstract interfaces rather than specific implementations. For example, they access database operations through a generic data repository interface and interact with EHR systems through standardized connector interfaces. This approach allows the system to change database technologies or add new EHR systems without modifying the core business logic.

10.4 Interface Segregation Principle

The system defines focused, minimal interfaces rather than general-purpose ones. EHR connectors expose only the methods needed for specific operations (authentication, data fetching, validation), preventing clients from depending on methods they don't use. API endpoints are similarly segregated by function rather than creating monolithic interfaces.

10.5 Release Reuse Equivalency Principle

Components with related functionality are bundled together for deployment. Provider management functions form one deployment unit, while EHR integration functions form another. This organization ensures that highly interdependent components are released together, minimizing version compatibility issues while facilitating code reuse within logical boundaries.

10.6 Common Closure Principle

Components that change for similar reasons are grouped together. Field mapping logic is centralized since changes to data formats affect all mapping operations. Authentication handling is isolated in dedicated components since security requirements typically change together. This organization minimizes the impact of changes by containing them within well-defined boundaries.

10.7 Common Reuse Principle

The system packages components that tend to be reused together. The core EHR connection logic is bundled with authentication and error handling since these are typically needed together. Data transformation utilities are packaged as a reusable library used across multiple Lambda functions. This principle minimizes unnecessary dependencies between components.

11 Deployment Model

11.1 Deployment Architecture

The healthcare provider onboarding application will be deployed using a fully managed AWS serverless architecture. The React frontend will be hosted on AWS Amplify, which provides continuous deployment and hosting. The backend components will be deployed through AWS Lambda. This approach enables reliable, reproducible deployments with minimal manual intervention.

11.2 Deployment Process

1. Frontend deployment via AWS Amplify CI/CD pipeline connected to WinterGreen's Git repository.
2. Backend infrastructure provisioning using API Gateway, Lambda functions, and database resources.
3. Database initialization scripts for creating tables and loading initial EHR configurations.
4. Secrets Manager setup for securely storing EHR API credentials.
5. Configuration of loggers for monitoring.

11.3 Intellectual Property Considerations

- All custom code developed specifically for this project will be the property of the client.
- AWS account and resource ownership will be transferred to client upon project completion.
- Documentation will include clear attribution of all components and their ownership.

11.4 Maintenance Documentation

The following documentation will be provided to ensure the client can effectively maintain the system:

- System Design Document detailing all components and their interactions.
- Database schema documentation.
- Extensively documented code.
- Deployment guide with step-by-step instructions.
- Instructions for adding new EHR systems through database configuration.

11.5 Support Model

- Knowledge transfer sessions with client's technical team for complete code documentation and walk through.