Daily notes 16-20

Weekly Summary

- **DICOM Integration: Shipped.** Doctors/patients can now upload, download, and view images. Implemented a secure dual-storage model (SeaweedFS + Orthanc).
- **Epic Patient Import (FHIR): Shipped.** Added a two-step preview/confirm import process.
- **Accountant Search: Shipped.** Deployed a new unified search module for payments, deposits, and refunds with database-level filtering.
- **Patient Autocomplete Search: Enhanced.** Replaced old search with a high-performance, debounced autocomplete feature (case/accent-insensitive).
- **Stripe Refactor: Completed.** Centralized all payment logic into a single service, removing the Stripe SDK from the Domain layer.
- **Selective FHIR Fetching: Enhanced.** Doctors can now request specific FHIR resources on-demand.
- **Critical Cache Fix: Resolved.** Fixed a bug where payment updates (Cash/Stripe) failed to invalidate the medical record cache, ensuring data consistency.
- **Medical Record State Fix: Resolved.** Ensured medical records cannot be completed until all prescription items are finalized.
- **Pulumi (IaC):** Adopted a hybrid strategy, using high-level components for infrastructure (EKS) and low-level resources for app deployments.

Next Week Plan: Telemedicine

Description: Patients will be able to book appointments and initiate a direct chat or video call with a doctor through the platform. This feature is intended for consultations and diagnoses of conditions that do not require an in-person physical examination. (WebRTC, SignalR)

Day 11

New Feature: Patient MedicalRecord filter

Filtering Endpoint: `GET /api/patient-portal/medical-records` now supports server-side pagination and filtering via `specialty` , `doctorName` , `startDate` , and `endDate` parameters.

Architecture & Performance

- **Specification Pattern:** All filtering implemented using the Specification Pattern for efficient database-level queries, preventing in-memory processing overhead.
- **Async & Paginated:** Redis cache operations are fully asynchronous and paginated for scalability.

Critical Bug Fix

Cache Invalidation: Fixed bug where medical record cache wasn't invalidated after payment updates in `AccountantApplicationService` (cash payments/refunds) and `PaymentsController` (Stripe webhooks), ensuring data consistency.

New Feature: Accountant Search (Payments, Deposits, Refunds)

- **Unified Filtering:** New `AccountantFilterDto` provides standardized search criteria.
- **Updated Endpoints:**
  - `GET /api/accountant/eligible-for-deposit-appointments`
  - `GET /api/accountant/unpaid-medical-records`
  - `GET /api/accountant/refundable-medical-records`
- **Searchable Fields:** `PatientName` , `AppointmentId` , `MedicalRecordId` , `PaymentStatus` , date ranges.
- **Implementation:** Granular specifications (e.g., `UnpaidMedicalRecordSpecification`) ensure reusable, type-safe, database-level filtering with async pagination.

Bug Fixes

- Fixed incorrect `DateTimeKind` for PostgreSQL.

Enhancement: Doctor Appointment Filtering

Flexible Date Filtering: `GET /api/Appointments/my-appointments` endpoint modified to support opt-in date filtering via `startDate` and `endDate` parameters.

Functional Changes

- **Removed Default Constraints:** Eliminated enforced 7-day default date range. Endpoint now returns all doctor appointments when no date parameters provided.
- **Optional Date Range:** Date filtering is strictly opt-in, allowing doctors to view complete appointment history or filter by specific date ranges as needed.

Resolved

- **Bug:** Incorrect Medical Record State Validation.
- **Resolution:** A fix has been implemented. The system now correctly validates that all associated `PrescriptionItems` are in a final state (`Completed`, `Cancelled`, or `RequestCancelled`) before allowing a **Medical Record** to be marked as complete.

In Progress Working

- **Feature: DICOM (Orthanc) Integration**.
- **Objective:** Actively developing the feature to allow doctors and patients to upload, download, and view DICOM images directly within the medical record.

Day 12

Dicom Images

1. Data Flow

- **Storage Strategy:** A **Dual Storage** model is implemented.

- **Upload:** (`POST /api/dicom/upload`)
    1. The original file is always pushed to **SeaweedFS** (acting as the "source of truth").
    2. If the file is DICOM, it is *also* pushed to **Orthanc** for automatic metadata extraction and JPEG preview generation.
    3. `ImageInfo` is saved to PostgreSQL, linking the SeaweedFS key and the Orthanc ID.
- **Download/Preview:**
    - The original file (Download) is streamed directly from **SeaweedFS**.
    - The preview file (JPEG) is streamed from **Orthanc**.

2. Security Analysis

- The system is protected by a **multi-layered** model:
    1. **Authentication:** All endpoints require a **JWT** (`[Authorize]`).
    2. **Authorization:** Uploads are restricted by the **"Doctor" Role**.
    3. **Data Access:** All retrieval actions perform an **ownership check** (`CanUserAccessMedicalRecord`), ensuring Doctors/Patients only see their own data.
- **Vulnerability Patched:** The `AuthenticatedImage` component has been implemented, forcing all images to stream via an authenticated API, patching a direct access vulnerability.

3. Performance

- **Trade-off:** The **Dual-upload** process (pushing to two systems) introduces a slight, but necessary and acceptable, latency.
- **Efficiency:**
    - Leveraging **Orthanc** for preview generation is highly efficient (offloads work from the .NET API).
    - Leveraging **SeaweedFS** for file streaming (high performance as a dedicated object store).

Refactor: Stripe Payment Logic

1. Created a `StripeSessionDto` in the Domain to **remove the Stripe SDK dependency** from the core Domain layer.
2. Created a centralized `StripePaymentService` in the Infrastructure layer to hold the single, authoritative logic for creating sessions.
3. Used **Dependency Injection** to inject this new service into the required controllers and services, replacing all duplicated code.

Day 13

Import Patient from Epic - Implementation Summary

Two-Step Import Process

- **GET /api/patients/import-preview**: Validates Epic Patient ID, fetches demographics from FHIR API, returns `PatientImportPreviewDto` for confirmation
- **POST /api/patients/import-confirm**: Creates patient profile after user confirmation, links to authenticated user

Domain Model Enhancement

- Replaced `int Age` with `DateTime? DateOfBirth` (nullable)
- `Age` now calculated property derived from `DateOfBirth`
- **Benefit**: Stores source of truth, prevents stale data, improves domain integrity

Security & Data Integrity

- `[Authorize]` on import-confirm endpoint
- Enforces one patient profile per user (returns 400 if `PatientId` exists)
- User ID retrieved from JWT claims server-side (`ClaimTypes.NameIdentifier`)

Performance

- Single Epic API call for preview
- Single atomic transaction for creation

Selective FHIR Resource Fetching (phân trang)

Functionality Introduced

Doctors can now request specific **FHIR resource types** (e.g., `Condition`, `Observation`) from the external Epic system on demand.

Implementation

- **New API Endpoint:** `GET /api/FhirEpic/patient/{patientId}/external-history-selective` was added. It accepts a `resourceTypes` query parameter array specifying the desired resources.
- **Application Service Logic:** The `EhrFhirApplicationService` now includes an overloaded `GetExternalPatientHistoryAsync` method that orchestrates calls to the underlying integration services based solely on the provided `resourceTypes`. If no types are specified, it defaults to fetching all resources for backward compatibility.

Architecture

- The selection logic is correctly placed in the **Application Service**, adhering to Clean Architecture.
- No modifications were needed in the core `EpicFhirIntegrationService` (Infrastructure layer).
- The feature is additive, ensuring **no breaking changes** to existing functionality.

Day 14

Appointment Search Enhancement

Overview

Enhanced patient search functionality in the doctor appointment management screen through two development phases:

1. **Fuzzy Search**: Initially implemented using FuzzySharp library to handle typos.

2. **Autocomplete Search**: Current implementation - system provides real-time name suggestions as doctors type, executing search only after suggestion selection.

Frontend Changes (`AppointmentsSection.tsx`)

UX Improvements: Dropdown suggestion list appears below search input; auto-fills selected name and triggers instant search without manual filter button click.

Performance Optimization: Debouncing with 300ms delay - API requests only fire after user stops typing, significantly reducing server load and resource consumption. Added state management (`suggestions` , `isSuggestionsLoading`) for smooth UI rendering.

Maintainability: Separated suggestion and main search logic; utilized standard React Hooks (`useState` , `useEffect` , `useRef`) for readable, maintainable code.

Backend Changes (C# / ASP.NET Core)

New Autocomplete Endpoint (`GET /api/Patients/search`): Dedicated endpoint optimized for fast response - returns string array of names only (not full Patient objects), uses `Distinct()` to remove duplicates, limits results to 10 via `Take(10)` . Protected with `[Authorize(Roles = "Doctor,Admin")]` .

Case & Accent-Insensitive Search: Case-insensitive via `ToLower()` in LINQ (translates to SQL `LOWER()` for database-level performance). Accent-insensitive requires PostgreSQL `unaccent` extension (run `CREATE EXTENSION IF NOT EXISTS unaccent;` once). Created `UnaccentExtension.cs` stub class and registered in `HospitalDbContext.OnModelCreating` to map to PostgreSQL's `unaccent()` function.

Maintainability: EF Core function registration is standard practice; separate `UnaccentExtension.cs` file keeps codebase clean and organized.

```
// In YourEksProject/Program.cs
using Pulumi;
using Pulumi.Aws.Eks;
using System.Collections.Generic;

return await Deployment.RunAsync(() =>
{
    var eksCluster = new Cluster("my-eks-cluster", new ClusterArgs
    {
        // ... các tham số cấu hình cluster EKS của bạn ...
    });

    // ───→ Bước quan trọng: Xuất kubeconfig ra làm stack output
    return new Dictionary<string, object?>
    {
        ["kubeconfig"] = eksCluster.KubeconfigJson
    };
});
```

Infrastructure as Code: High-Level vs Low-Level Resources in Pulumi

Core Concept Explanation

Low-Level Resources (`CustomResource`): These are cloud resources that map 1:1 directly to cloud provider APIs (AWS, Azure, GCP, Kubernetes). Every property available in the provider's official API documentation is exposed. They provide maximum control and granularity but require detailed configuration.

High-Level Resources (`ComponentResource`): These are logical abstractions created by Pulumi that encapsulate multiple low-level resources following industry best practices. They prioritize developer convenience by handling complex setup internally and exposing simplified APIs.

Technical Comparison

Low-Level Resources (e.g., `aws.eks.Cluster`)

Direct API Mapping: Returns raw outputs exactly as the cloud provider API returns them (endpoint, certificate, cluster name as separate fields).

Immediate Feature Access: New AWS features become available instantly without waiting for abstraction layer updates.

Full Control: All configuration options from the provider API are exposed, allowing precise infrastructure definition.

Manual Assembly Required: Users must manually combine outputs and write glue code to create higher-level artifacts (e.g., kubeconfig files).

High-Level Resources (e.g., `eks.Cluster`)

Multi-Resource Orchestration: Internally creates and wires together multiple low-level resources (cluster, node groups, IAM roles, networking).

Best Practices Baked In: Implements security and operational best practices by default, reducing boilerplate and configuration errors.

Convenient Outputs: Provides ready-to-use artifacts like `.Kubeconfig` by automatically assembling raw API outputs into usable formats.

Abstraction Trade-off: Less granular control over individual sub-resources, though most high-level components allow customization through configuration options.

Kubernetes Context: Deployment/Service Resources

Low-Level (`k8s.apps.v1.Deployment`, `k8s.core.v1.Service`): Maps 1:1 to Kubernetes API objects. Requires defining complete manifests programmatically, equivalent to writing YAML but in code.

High-Level (`kubernetes.helm.v3.Chart`): The primary high-level abstraction for Kubernetes. Deploys entire application stacks (Deployment + Service + Ingress + ConfigMaps) with minimal code by leveraging pre-packaged Helm charts.

Implementation Strategy

The optimal approach combines both paradigms:

Use High-Level for Infrastructure Foundation: Cluster creation, VPCs, and complex multi-resource setups benefit from components that handle intricate wiring.

Use Low-Level for Application-Specific Logic: Custom Deployments, Services, and resources requiring precise control are better defined with low-level APIs.

This hybrid strategy is standard practice in production Pulumi implementations, balancing rapid development with necessary customization.

Component Architecture

High-level components are themselves Pulumi resources that contain child resources internally. Teams can also author custom components to encapsulate organization-specific patterns, making them reusable across projects and languages through auto-generated SDKs.

Pulumi: High-Level vs Low-Level Resources

Khái Niệm Cốt Lõi

Low-Level Resources (Tài nguyên cấp thấp): Là các tài nguyên cloud ánh xạ 1:1 trực tiếp với API của nhà cung cấp (AWS, Azure, GCP, Kubernetes). Mọi thuộc tính trong tài liệu chính thức của provider đều được expose đầy đủ. Cho phép kiểm soát tối đa nhưng yêu cầu cấu hình chi tiết.

High-Level Resources (Tài nguyên cấp cao): Là các component do Pulumi tạo ra, đóng gói nhiều low-level resources theo các best practices. Ưu tiên sự tiện lợi bằng cách xử lý logic phức tạp bên trong và cung cấp API đơn giản hóa.

So Sánh Kỹ Thuật

Low-Level Resources (ví dụ: `aws.eks.Cluster`)

Ánh xạ API trực tiếp: Trả về output thô y như cloud provider API (endpoint, certificate, cluster name là các field riêng biệt).

Truy cập tính năng ngay lập tức: Các tính năng mới của AWS khả dụng ngay mà không cần chờ abstraction layer cập nhật.

Kiểm soát toàn diện: Mọi tùy chọn cấu hình từ provider API đều được expose, cho phép định nghĩa infrastructure chính xác.

Cần lắp ráp thủ công: Người dùng phải tự kết hợp các output và viết glue code để tạo các artifact cấp cao hơn (ví dụ: file kubeconfig).

High-Level Resources (ví dụ: `eks.Cluster`)

Điều phối đa tài nguyên: Bên trong tự động tạo và kết nối nhiều low-level resources (cluster, node groups, IAM roles, networking).

Best practices tích hợp sẵn: Triển khai các best practices về bảo mật và vận hành mặc định, giảm boilerplate và lỗi cấu hình.

Output tiện dụng: Cung cấp các artifact sẵn sàng sử dụng như `.Kubeconfig` bằng cách tự động lắp ráp các raw API output thành format khả dụng.

Đánh đổi abstraction: Ít kiểm soát chi tiết hơn đối với các sub-resource riêng lẻ, tuy nhiên hầu hết high-level components cho phép tùy chỉnh qua configuration options.

Trong Bối Cảnh Kubernetes: Deployment/Service Resources

Low-Level (`k8s.apps.v1.Deployment`, `k8s.core.v1.Service`): Ánh xạ 1:1 với Kubernetes API objects. Yêu cầu định nghĩa manifest đầy đủ bằng code, tương đương viết YAML nhưng bằng ngôn ngữ lập trình.

High-Level (`kubernetes.helm.v3.Chart`): High-level abstraction chính cho Kubernetes. Deploy toàn bộ application stack (Deployment + Service + Ingress + ConfigMaps) với ít code nhờ tận dụng Helm charts đã đóng gói sẵn.

Chiến Lược Triển Khai

Cách tiếp cận tối ưu là kết hợp cả hai:

Dùng High-Level cho Infrastructure Foundation: Tạo cluster, VPCs, và các setup đa tài nguyên phức tạp hưởng lợi từ components xử lý wiring phức tạp.

Dùng Low-Level cho Application-Specific Logic: Custom Deployments, Services và các resources cần kiểm soát chính xác được định nghĩa tốt hơn với low-level APIs.

Chiến lược hybrid này là chuẩn mực trong các triển khai Pulumi production, cân bằng giữa phát triển nhanh và khả năng tùy chỉnh cần thiết.

Kiến Trúc Component

High-level components chính là Pulumi resources chứa các child resources bên trong. Các team có thể tự tạo custom components để đóng gói các pattern đặc thù của tổ chức, làm chúng có thể tái sử dụng qua nhiều project và ngôn ngữ thông qua auto-generated SDKs.

efs csi, cluster auto scaler

CoreDNS

1. **Nó có thật sự cần thiết không?**
   Có, bắt buộc phải có. Cluster Kubernetes sẽ hoàn toàn không thể sử dụng được nếu thiếu CoreDNS. Nó chính là DNS của cluster, chịu trách nhiệm phân giải tên của các service.
   Không có nó, các pod sẽ không thể giao tiếp với nhau bằng tên, và toàn bộ kiến trúc microservice sẽ sụp đổ.

2. **Tại sao nó cài lâu?**
   Thời gian cài đặt lâu không phải chỉ là do việc "deploy" một vài pod. Khi được cài dưới dạng một EKS Addon, AWS đang thực hiện nhiều việc hơn ở phía sau:

- Đảm bảo CoreDNS được tích hợp sâu vào control plane của EKS.
- Cấu hình nó để có tính sẵn sàng cao (high availability).
- Quản lý vòng đời, phiên bản và tự động vá lỗi cho nó.

Bạn đang trả một khoản "phí thời gian" một lần để đổi lấy một thành phần cốt lõi được AWS quản lý, ổn định và tự động duy trì về sau.

1. **Có thể tắt riêng nó không?**
   Không thể. Với component high-level Pulumi.Eks.Cluster, bạn chỉ có thể dùng cờ skipDefaultAddons=true để tắt toàn bộ các addon mặc định (coredns, kube-proxy, vpc-cni) chứ
   không thể tắt riêng lẻ.

Việc tắt toàn bộ sẽ khiến bạn phải tự cài đặt thủ công lại cả 3, điều này còn phức tạp và tốn thời gian hơn nhiều.

Kết luận: Dù thời gian chờ có thể gây khó chịu, CoreDNS là một thành phần nền tảng không thể thiếu. Thời gian đó là để AWS đảm bảo nó được cài đặt một cách bền vững và được
quản lý hoàn toàn.