Case Study: Applying Domain-Driven Design in a Healthcare Management System

Overview

Project: MedFlow – A comprehensive healthcare platform for managing patient records, appointments, prescriptions, and insurance claims.
Domain Complexity: High — strict regulations (HIPAA), complex workflows, and evolving medical logic.
Tech Stack: C# (.NET 6), Angular (Frontend), PostgreSQL, Event Store DB, Docker/Kubernetes
Team Size: 15 (8 backend, 4 frontend, 3 domain experts/analysts)
Duration: 12 months (from discovery to production rollout)

Challenge

MedFlow aimed to replace legacy hospital systems with a modern, compliant, and scalable solution. Initial attempts failed due to:

- Miscommunication between developers and clinical staff.
- Business logic scattered across services and stored procedures.
- Inconsistent terminology (e.g., "visit" vs. "appointment" vs. "consultation").
- Difficulty adapting to regulatory changes.

The team adopted **Domain-Driven Design (DDD)** to align software with the complexity of healthcare operations.

Implementation of Domain-Driven Design

1. Ubiquitous Language

- Developers and domain experts (doctors, nurses, administrators) held weekly modeling workshops.
- Created a shared glossary: e.g., Patient, Encounter, CarePlan, ClaimSubmission.
- All code, documentation, and APIs used this consistent language.
- Example: The term "Encounter" replaced ambiguous terms like "visit" or "session" across the system.

2. Bounded Contexts

The system was decomposed into clear, autonomous Bounded Contexts:

Bounded Context	Responsibility
Patient Management	Patient registration, demographics, consent
Scheduling	Appointment booking, calendar management
Clinical Records	Medical history, diagnoses, treatments
Prescriptions	E-prescribing, drug interactions
Billing & Claims	Insurance validation, claim submission
Notifications	Alerts, reminders, audit logs

Each context had its own model, database, and API — reducing ambiguity and coupling.

3. Context Mapping

Defined relationships between contexts:

- **Scheduling** → **Clinical Records**: *Upstream/Downstream* via events (AppointmentScheduled)
- **Prescriptions** → **Billing**: *Conformist* — Prescriptions uses Billing's medication codes
- **Patient Management** ↔ **All**: *Shared Kernel* for core patient identity
- **Claims** → **External Payers**: *Anti-Corruption Layer (ACL)* to translate internal models to HL7/FHIR standards

4. Strategic Design in Action

- Used **Event Storming** sessions to map patient journey from check-in to discharge.
- Identified **Aggregates** like Patient (root), Encounter, and Prescription with clear transactional boundaries.
- Designed **Repositories** and **Factories** aligned with aggregate consistency rules.

5. Tactical Patterns Applied

- **Entities**: Patient (ID-based lifecycle)
- **Value Objects**: Address, DiagnosisCode, Dosage
- **Aggregates**: Encounter (guards state during treatment updates)

- **Domain Events**: PatientRegistered, PrescriptionIssued, ClaimRejected
- **Services**: MedicationInteractionChecker (stateless business logic)
- **Modules**: Code organized by context (e.g., /ClinicalRecords/...) not technical layers

6. Event Sourcing & CQRS (Advanced DDD)

- Critical contexts (Clinical Records, Claims) used **Event Sourcing** to store all state changes as events.
- Enabled full audit trail and temporal queries (e.g., "Show patient record as of 3 months ago").
- **CQRS** separated read models (optimized dashboards) from write models (transaction-safe commands).

Results

- ✓ **Improved Communication**:
  Developers and clinicians spoke the same language — fewer misunderstandings and rework.
- ✓ **Faster Feature Delivery**:
  New features like telehealth visits were added in 3 weeks within the Scheduling context, isolated from others.
- ✓ **Regulatory Compliance**:
  Audit trails and data integrity improved; passed HIPAA audit with zero critical findings.
- ✓ **Scalable Microservices**:
  Bounded Contexts evolved into independently deployable microservices with minimal coupling.
- ✓ **Resilient to Change**:
  When insurance rules changed, only the **Billing & Claims** context needed updates — no ripple effects.
- ✓ **High Maintainability**:
  Codebase remained understandable despite growing complexity; onboarding time reduced to 10 days.

Lessons Learned

- **Invest in Discovery**: Weekly domain modeling sessions were essential — treated as non-negotiable.
- **Start Small**: Began with one context (Patient Management) before expanding.

- **Hire Domain Experts**: Two clinical informaticists embedded in the team made a huge difference.
- **Avoid Over-Engineering**: Not all contexts needed CQRS/event sourcing — applied only where justified.
- **Enforce Boundaries**: Used separate databases and namespaces to prevent context bleed.

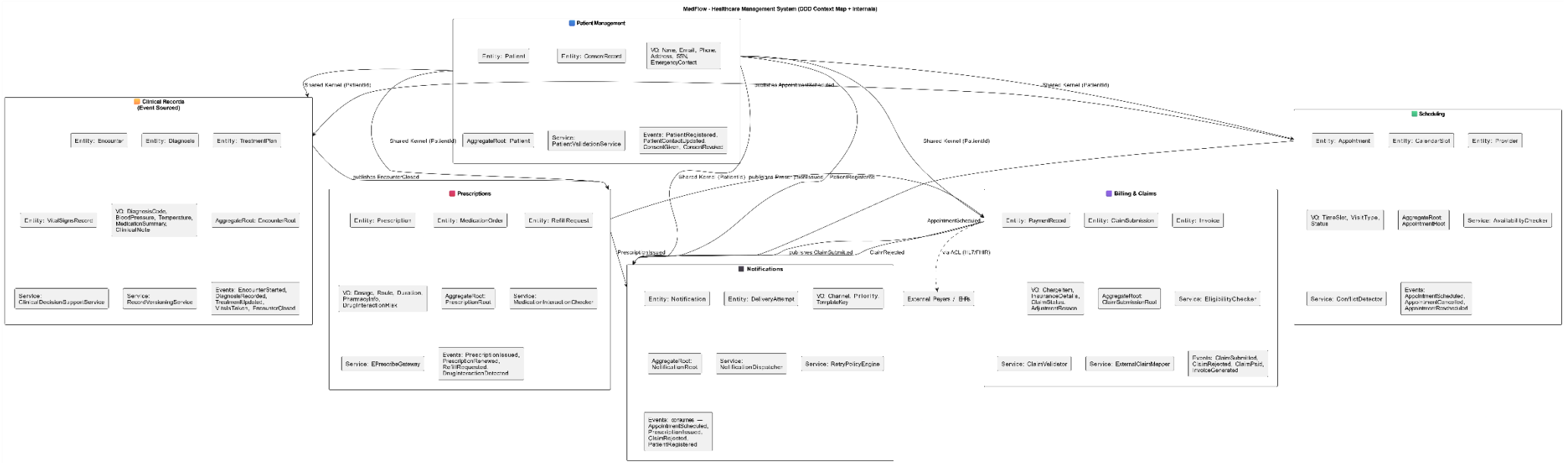
Conclusion

By applying **Domain-Driven Design**, MedFlow transformed a high-risk project into a maintainable, scalable, and clinically accurate system. The focus on **ubiquitous language**, **bounded contexts**, and **strategic modeling** allowed the team to tame complexity and deliver value faster. DDD proved indispensable in a domain where precision, compliance, and collaboration are paramount.

MedFlow – Bounded Contexts and Domain Model (DDD)

Bounded Context	Entities	Value Objects	Aggregate Roots	Domain Events	Services	Key Relationships
Patient Management ***(Core)***	Patient ConsentRecord	Name Email Phone Address SSN EmergencyContact	Patient	PatientRegistered PatientContactUpdated ConsentGiven ConsentRevoked	PatientValidationService	**Shared Kernel** with all contexts via PatientId Publishes: PatientRegistered → Scheduling, Clinical Records
Scheduling	Appointment CalendarSlot Provider	TimeSlot VisitType Status	Appointment	AppointmentScheduled AppointmentCancelled AppointmentRescheduled	AvailabilityChecker ConflictDetector	Depends on: Patient Management Publishes: AppointmentScheduled → Clinical Records, Notifications
Clinical Records ***(Event-Sourced)***	Encounter Diagnosis TreatmentPlan VitalSignsRecord	DiagnosisCode (ICD-10) BloodPressure Temperature MedicationSummary ClinicalNote	Encounter	EncounterStarted DiagnosisRecorded TreatmentUpdated VitalsTaken EncounterClosed	ClinicalDecisionSupportService RecordVersioningService	Consumes: AppointmentScheduled Publishes: EncounterClosed → Prescriptions, Billing
Prescriptions	Prescription MedicationOrder RefillRequest	Dosage Route Duration PharmacyInfo DrugInteractionRisk	Prescription	PrescriptionIssued PrescriptionRenewed RefillRequested DrugInteractionDetected	MedicationInteractionChecker EPrescribeGateway	Depends on: Clinical Records, Patient Management Publishes: PrescriptionIssued → Billing, Notifications
Billing & Claims	ClaimSubmission Invoice PaymentRecord	ChargeItem (CPT) InsuranceDetails ClaimStatus AdjustmentReason	ClaimSubmission	ClaimSubmitted ClaimRejected ClaimPaid InvoiceGenerated	EligibilityChecker ClaimValidator ExternalClaimMapper	Uses **Anti-Corruption Layer (ACL)** to external payers (HL7/FHIR) Consumes: PrescriptionIssued, EncounterClosed
Notifications	Notification DeliveryAttempt	Channel (SMS/Email)	Notification	*(Reactive consumer of events from all contexts)*	NotificationDispatcher RetryPolicyEngine	Subscribes to: • AppointmentScheduled • PrescriptionIssued

		Priority TemplateKey				• ClaimRejected • PatientRegistered Sends alerts via SMS, Email, In-app



Context Mapping Summary

Relationship Type	From → To	Description
Shared Kernel	Patient Management ↔ All	Shared model for PatientId and core identity; strict change control
Upstream → Downstream	Scheduling → Clinical Records → Prescriptions → Billing	Event-driven flow of clinical workflow

Anti-Corruption Layer (ACL)	Billing & Claims → External Payers	Translates internal ClaimSubmission into HL7/FHIR for insurance systems
Event-Driven Integration	All → Notifications	Asynchronous communication via message broker (e.g., Kafka)

Notes

- **Ubiquitous Language**: All terms (e.g., Encounter, Prescription) are defined in domain glossary and used consistently.
- **Event Sourcing**: Applied in *Clinical Records* and *Billing & Claims* for auditability and temporal queries.
- **Bounded Context = Microservice Boundary**: Each context can be implemented as an independent service with its own database.
- **Integration**: Domain events are published via a message bus; consumers react asynchronously.