AetherLearn LMS — Technical Foundation Document

Doc 1 of 6 · Architecture, Schema, Security, Multi-Tenancy, Roles, Standards, and ADRs

AetherLearn is a production-deployed modular monolith LMS at **lms.gagneet.com** built on Next.js 16, React 19, Prisma ORM, and PostgreSQL, with a **7-tier RBAC system spanning 72+ permissions and 30+ data models across four domains**. This document captures the system's current architectural state, validates existing technical decisions, and provides an implementable evolution roadmap — from the flat directory structure already in production to a properly bounded modular monolith with cross-module event-driven communication. Every recommendation references the actual patterns established in `lib/auth.ts`, `lib/rbac.ts`, and `lib/tenancy.ts`, and is designed for incremental adoption without disrupting production.

Section 1: Architecture overview and modular monolith blueprint

Current state — a working flat monolith

AetherLearn runs as a **Next.js 16 App Router application** with API routes at `app/api/`, role-based dashboard routing at `app/dashboard/student/`, `app/dashboard/tutor/`, and `app/dashboard/supervisor/`, and shared utilities in `lib/`. Business logic currently lives inside API route handlers and page components. This structure works for the current scale but creates three escalating risks: **cross-domain coupling** (finance logic importing academic helpers directly), **testing difficulty** (business logic entangled with HTTP handling), and **onboarding friction** (new developers cannot discern domain boundaries from the file tree).

The existing security helpers — `lib/audit.ts`, `lib/rbac.ts`, `lib/tenancy.ts` — already demonstrate the right instinct: cross-cutting concerns extracted into shared utilities. The modular evolution extends this pattern to all business logic.

Target module structure

The recommended architecture introduces a `modules/` directory where each domain encapsulates its service logic, types, validation schemas, events, and repository layer behind a barrel export. The `app/` directory retains only thin route handlers and UI components that delegate to modules. `thetshaped`

```
src/
├── app/                          # Routes only — thin delegation layer
│   ├── (auth)/login/page.tsx
│   ├── (dashboard)/
│   │   ├── layout.tsx            # Auth-protected, role-routing layout
│   │   ├── student/
│   │   │   ├── page.tsx
│   │   │   └── _components/      # Route-colocated UI (excluded from routing)
│   │   ├── tutor/
│   │   └── supervisor/
│   └── api/                      # Thin handlers → delegate to module services
│       ├── courses/route.ts
│       ├── attendance/route.ts
│       └── finance/invoices/route.ts
│
├── modules/                      # Domain-bounded business logic
│   ├── auth/                     # Existing lib/auth.ts, lib/rbac.ts extracted
│   │   ├── index.ts              # Barrel export: public API only
│   │   ├── auth.service.ts
│   │   ├── rbac.service.ts       # From current lib/rbac.ts
│   │   ├── auth.types.ts
│   │   └── auth.schema.ts        # Zod schemas for auth inputs
│   ├── core/                     # Centre, User management
│   │   ├── index.ts
│   │   ├── centre.service.ts
│   │   ├── user.service.ts
│   │   └── core.types.ts
│   ├── academic/                 # Courses, Lessons, Sessions, Attendance
│   │   ├── index.ts
│   │   ├── course.service.ts
│   │   ├── attendance.service.ts
│   │   ├── catchup.service.ts
│   │   ├── academic.repository.ts  # Prisma queries isolated here
│   │   ├── academic.events.ts      # Domain events: ATTENDANCE_RECORDED, etc.
│   │   ├── academic.types.ts
│   │   └── academic.schema.ts
│   ├── gamification/             # XP, Badges, Streaks, Levels
│   │   ├── index.ts
│   │   ├── xp.service.ts
│   │   ├── badge.service.ts
│   │   ├── leaderboard.service.ts
│   │   └── gamification.events.ts
│   ├── finance/                  # FeePlans, Invoices, Payments, Refunds
│   │   ├── index.ts
│   │   ├── invoice.service.ts
│   │   ├── payment.service.ts
```

```
│   │   └── finance.events.ts
│   ├── operations/                # Tickets, SLA, Assets (future)
│   │   ├── index.ts
│   │   ├── ticket.service.ts
│   │   └── sla.service.ts
│   ├── governance/                # AuditEvent, ApprovalRequest
│   │   ├── index.ts
│   │   ├── audit.service.ts       # From current lib/audit.ts
│   │   └── approval.service.ts
│   ├── documents/                 # FUTURE: QR codes, scanned docs
│   ├── notifications/             # FUTURE: templates, queues, delivery
│   ├── integrations/              # FUTURE: video providers, payment gateways
│   └── shared/                    # Cross-cutting: event bus, base types
│       ├── index.ts
│       ├── event-bus.ts           # BullMQ-backed async event bus
│       ├── sync-bus.ts            # In-process EventEmitter for same-request
│       └── base.types.ts
│
├── lib/                           # Infrastructure utilities (non-domain)
│   ├── prisma.ts                  # Prisma client singleton
│   ├── redis.ts                   # Redis client singleton
│   ├── tenancy.ts                 # Existing multi-tenancy enforcement
│   ├── logger.ts                  # Pino structured logging
│   ├── cache.ts                   # Redis cache wrapper
│   └── s3.ts                      # MinIO/S3 client
│
└── components/                    # Shared UI components
    ├── ui/                        # Button, Input, Card, etc.
    └── layout/                    # Header, Sidebar, Footer
```

The **barrel export pattern** ensures each module exposes only its public API. Internal implementation details — repositories, helper functions, internal types — remain encapsulated:

typescript

```typescript
// modules/academic/index.ts — the module's public contract
export { CourseService } from './course.service';
export { AttendanceService } from './attendance.service';
export { CatchUpService } from './catchup.service';
export { AcademicEvents } from './academic.events';
export type { CourseWithModules, AttendanceRecord, CatchUpPackage } from './academic.types';
export { createCourseSchema, recordAttendanceSchema } from './academic.schema';
// Internal: academic.repository.ts is NOT exported
```

API route handlers become thin delegation layers: makerkit

```typescript
// app/api/courses/route.ts
import { auth } from '@/modules/auth';
import { CourseService, createCourseSchema } from '@/modules/academic';
import { NextRequest, NextResponse } from 'next/server';

export async function POST(req: NextRequest) {
  const session = await auth();
  if (!session) return NextResponse.json({ error: 'Unauthorized' }, { status: 401 });

  const body = await req.json();
  const result = createCourseSchema.safeParse(body);
  if (!result.success) return NextResponse.json({ errors: result.error.flatten() }, { status: 422 });

  const course = await new CourseService().create({
    ...result.data,
    centreId: session.user.centreId,  // centreId from session ONLY
  });
  return NextResponse.json(course, { status: 201 });
}
```

Module dependency matrix

Not all modules should import from all others. The matrix below defines allowed direct dependencies (via barrel imports). All other cross-module communication flows through the **event bus**.

Module → can import from ↓	shared	auth	core	academic	gamification	finance	operations	governance
auth	✅	—	✅	❌	❌	❌	❌	❌
core	✅	✅	—	❌	❌	❌	❌	❌
academic	✅	✅	✅	—	❌	❌	❌	❌
gamification	✅	✅	✅	❌	—	❌	❌	❌
finance	✅	✅	✅	❌	❌	—	❌	❌
operations	✅	✅	✅	❌	❌	❌	—	❌
governance	✅	✅	✅	❌	❌	❌	❌	—

The critical rule: **domain modules never import directly from peer domain modules**. When `academic` needs to trigger `gamification` (e.g., awarding XP after attendance), it publishes a domain event. This prevents circular dependencies and keeps modules extractable.

Enforce this with dependency-cruiser in CI: atomicobject

```javascript
// .dependency-cruiser.js
module.exports = {
  forbidden: [
    {
      name: 'no-cross-module-internals',
      severity: 'error',
      from: { path: '^src/modules/([^/]+)/' },
      to: {
        path: '^src/modules/([^/]+)/',
        pathNot: [
          '^src/modules/$1/',              // Same module OK
          '^src/modules/[^/]+/index\\.ts$',  // Barrel exports OK
          '^src/modules/shared/',           // Shared OK
          '^src/modules/auth/',             // Auth OK for all
          '^src/modules/core/',             // Core OK for all
        ],
      },
    },
    {
      name: 'modules-must-not-import-app',
      severity: 'error',
      from: { path: '^src/modules/' },
      to: { path: '^src/app/' },
    },
  ],
};
```

Event bus architecture for PM2 cluster mode

AetherLearn runs in **PM2 cluster mode**, meaning each worker is a separate Node.js process with isolated memory. A standard EventEmitter does not propagate events across workers. GitHub The solution is a **two-tier event system**:

Tier 1 — In-process synchronous bus for reactions that must happen within the same request (e.g., updating a response object):

```typescript
```

```typescript
// modules/shared/sync-bus.ts
import { EventEmitter } from 'events';

export interface DomainEvent {
  type: string;
  payload: Record<string, unknown>;
  timestamp: Date;
  correlationId: string;
  centreId: string;
}

class SyncEventBus extends EventEmitter {}
export const syncBus = new SyncEventBus();
```

Tier 2 — BullMQ async bus (Redis-backed) for cross-module, cross-worker events. This is the primary event bus:

typescript

```typescript
// modules/shared/event-bus.ts
import { Queue, Worker } from 'bullmq';
import redis from '@/lib/redis';

export const domainEventsQueue = new Queue('domain-events', {
  connection: redis,
  defaultJobOptions: {
    attempts: 3,
    backoff: { type: 'exponential', delay: 1000 },
    removeOnComplete: 1000,
    removeOnFail: 5000,
  },
});

export async function publishEvent(event: DomainEvent) {
  await domainEventsQueue.add(event.type, event);
}

// Event handler registry — each module registers its handlers at startup
const handlers = new Map<string, ((event: DomainEvent) => Promise<void>)[]>();

export function onEvent(type: string, handler: (event: DomainEvent) => Promise<void>) {
  const existing = handlers.get(type) || [];
  handlers.set(type, [...existing, handler]);
}

export function startEventWorker() {
  new Worker('domain-events', async (job) => {
    const event = job.data as DomainEvent;
    const eventHandlers = handlers.get(event.type) || [];
    await Promise.allSettled(eventHandlers.map(h => h(event)));
  }, { connection: redis, concurrency: 5 });
}
```

Example cross-module flow — attendance triggers gamification:

```typescript
```

```
// modules/academic/attendance.service.ts
import { publishEvent } from '@/modules/shared';

export class AttendanceService {
  async recordAttendance(studentId: string, sessionId: string, centreId: string) {
    const record = await this.repository.create({ studentId, sessionId, centreId });
    await publishEvent({
      type: 'attendance.recorded',
      payload: { studentId, sessionId, status: record.status },
      centreId,
      timestamp: new Date(),
      correlationId: crypto.randomUUID(),
    });
    return record;
  }
}

// modules/gamification/index.ts — registers handler at startup
import { onEvent } from '@/modules/shared';
import { XPService } from './xp.service';

onEvent('attendance.recorded', async (event) => {
  const xpService = new XPService();
  await xpService.awardXP(event.payload.studentId as string, 10, 'ATTENDANCE');
});
```

BullMQ workers run as separate PM2 processes in fork mode alongside the clustered API:

javascript

```
// pm2.config.cjs
module.exports = {
  apps: [
    { name: 'api', script: './server.js', instances: 'max', exec_mode: 'cluster' },
    { name: 'event-worker', script: './worker.js', instances: 2, exec_mode: 'fork' },
  ],
};
```

Migration strategy — flat to modular without breaking production

Phase 1 (Week 1–2): Scaffold modules, move types and schemas. Create all `modules/[domain]/` directories with `index.ts` barrel files. Move existing types and Zod schemas first — these have zero runtime impact. Update `tsconfig.json` path aliases. All existing code continues to work.

Phase 2 (Week 3–4): Extract services from API routes. Starting with the least-coupled domain (governance/audit), move business logic from API route handlers into `[domain].service.ts` files. API routes

become thin wrappers. Run Playwright E2E tests after each extraction to verify nothing breaks.

Phase 3 (Week 5–6): Extract repositories and enforce boundaries. Move Prisma queries into [domain].repository.ts. Add dependency-cruiser to CI. Fix any boundary violations. The lib/audit.ts → modules/governance/audit.service.ts migration is the template for all others.

Phase 4 (Week 7–8): Wire up event bus. Install Redis and BullMQ. Replace direct cross-domain function calls with event publishing. Add PM2 event worker configuration.

Section 2: Database schema — 30+ models and evolution plan

Current schema across four domains

AetherLearn's database contains **30+ Prisma models** organized into four domains plus core entities. All tenant-scoped models include centreId with composite unique constraints ensuring data isolation.

Core Models (6): Centre (tenant root), User (with self-referential parentId/children for parent-student relationships), Course (with slug unique per centre), Module, Lesson, Content. The course hierarchy follows **Course → Module → Lesson → Content** with session support for live classes.

Academic Domain (8): ClassCohort (groups of students), ClassMembership, Session (live class instances with Teams/Zoom/Chime/Other provider integration), SessionAttendance, AttendanceRecord, CatchUpPackage (materials for absent students), AcademicProfile, GamificationProfile (XP, level, badges as COMPLETION/STREAK/MASTERY/PARTICIPATION/SPECIAL, achievements, activity streaks).

Finance Domain (7): FeePlan (with WEEKLY/MONTHLY/TERM/ANNUAL frequencies), StudentAccount, Invoice, InvoiceLine, Payment, Refund. Financial tracking supports the full invoice-to-payment lifecycle.

Operations Domain (4): Ticket (categories: IT, Inventory, Complaints, Maintenance, General), TicketComment, TicketAttachment, SLAConfig for response-time enforcement.

Governance Domain (2): AuditEvent (comprehensive audit logging), ApprovalRequest (workflow approvals).

RLS verdict — not worth adding now

PostgreSQL Row Level Security was thoroughly evaluated for layering atop the existing lib/tenancy.ts application-level filtering. **The recommendation is to not implement RLS at this stage.** Three factors drive this conclusion:

First, **Prisma has no native RLS support**. The official Prisma RLS example wraps every query in a $transaction to set app.current_centre_id via set_config(), and is explicitly marked "not intended for production." GitHub Third-party libraries like Yates and prisma-extension-rls exist but add fragile abstraction layers. GitHub

Second, **connection pooling creates data-leak risk**. Prisma's internal connection pool means session variables set via SET can persist across requests on the same connection. The mitigation — using

`set_config('app.current_centre_id', value, TRUE)` within transactions — adds overhead to every query and conflicts with Prisma's automatic batching.

Third, **performance is equivalent at best, worse at worst**. For simple equality policies (`centre_id = current_setting(...)`), PostgreSQL optimizes RLS to the same execution plan as a WHERE clause. Scottpierce But any policy complexity (joins, subqueries, non-`LEAKPROOF` functions) triggers per-row evaluation that can degrade queries from 1ms to 30 seconds. CYBERTEC PostgreSQL The existing application-level pattern provides identical security with better debuggability.

Revisit RLS if the database is ever exposed directly to other services (bypassing the application layer) or if regulatory requirements mandate database-level isolation.

Indexing strategy — centre_id as leading column

The **25+ existing indexes should be audited** using PostgreSQL's diagnostic views. The critical optimization: **`centreId` must be the leading column in all composite indexes** on tenant-scoped tables, because every query includes `WHERE centreId = ?` and B-tree indexes are scanned left-to-right. OneUptime

Recommended additions to `schema.prisma`:

```prisma
model AttendanceRecord {
  // ... fields
  @@index([centreId, date(sort: Desc)])
  @@index([centreId, studentId, date(sort: Desc)])
}

model Invoice {
  // ... fields
  @@index([centreId, status, createdAt(sort: Desc)])
}

model AuditEvent {
  // ... fields
  @@index([centreId, createdAt(sort: Desc)])
  @@index([centreId, entityType, entityId])
}

model GamificationProfile {
  // ... fields
  @@index([centreId, totalXp(sort: Desc)])  // Leaderboard queries
}
```

Partial indexes for hot query patterns (unpaid invoices, active tickets) must be added via custom migration SQL since Prisma doesn't support them natively: Atlas

```sql
```

```
CREATE INDEX idx_invoices_outstanding
  ON "Invoice" (centre_id, created_at DESC)
  WHERE status IN ('PENDING', 'OVERDUE');

CREATE INDEX idx_tickets_open
  ON "Ticket" (centre_id, priority, created_at DESC)
  WHERE status NOT IN ('CLOSED', 'RESOLVED');
```

To identify missing indexes in production, query `pg_stat_user_tables` for tables with high sequential scan ratios:

sql

```
SELECT relname, seq_scan, idx_scan,
  ROUND(100.0 * seq_scan / NULLIF(seq_scan + idx_scan, 0), 2) AS seq_pct
FROM pg_stat_user_tables WHERE n_live_tup > 1000
ORDER BY seq_tup_read DESC LIMIT 15;
```

Schema evolution — future models

Documents/QR Module (Phase 2):

prisma

```
model QRCode {
  id          String   @id @default(cuid())
  centreId    String
  entityType  String   // STUDENT, ASSET, DOCUMENT
  entityId    String
  payload     String   // Encoded data
  imageUrl    String?  // MinIO path
  scannedAt   DateTime?
  createdAt   DateTime @default(now())
  @@index([centreId, entityType, entityId])
}

model ScannedDocument {
  id          String   @id @default(cuid())
  centreId    String
  uploadedBy  String
  fileKey     String   // MinIO object key
  mimeType    String
  sizeBytes   Int
  annotations Json?    // Fabric.js JSON canvas state
  versions    DocumentVersion[]
  createdAt   DateTime @default(now())
  @@index([centreId, uploadedBy])
}

model DocumentVersion {
  id          String   @id @default(cuid())
  documentId String
  version     Int
  fileKey     String
  changedBy  String
  createdAt  DateTime @default(now())
}
```

Enhanced Gamification (Phase 2):

```
prisma
```

```
model XPLedger {
  id         String   @id @default(cuid())
  centreId   String
  studentId  String
  amount     Int      // Can be negative for deductions
  source     String   // ATTENDANCE, LESSON_COMPLETE, BADGE_EARNED, STREAK_BONUS
  sourceId   String?  // Reference to triggering entity
  balance    Int      // Running balance after this entry
  createdAt  DateTime @default(now())
  @@index([centreId, studentId, createdAt(sort: Desc)])
}

model BadgeRule {
  id         String   @id @default(cuid())
  centreId   String
  badgeType  BadgeType
  criteria   Json     // { type: "streak", days: 7 } or { type: "xp", threshold: 500 }
  isActive   Boolean  @default(true)
  @@unique([centreId, badgeType])
}

model LeaderboardCache {
  id         String   @id @default(cuid())
  centreId   String
  period     String   // WEEKLY, MONTHLY, ALL_TIME
  rankings   Json     // [{ studentId, rank, xp, name }]
  updatedAt  DateTime @updatedAt
  @@unique([centreId, period])
}
```

Enhanced Notifications (Phase 2):

```
prisma
```

```
model NotificationTemplate {
  id        String   @id @default(cuid())
  centreId  String?  // null = system-wide
  slug      String   // attendance_absent, invoice_due, xp_earned
  channel   String   // EMAIL, SMS, IN_APP, PUSH
  subject   String?
  body      String   // Handlebars template
  @@unique([centreId, slug, channel])
}

model NotificationQueue {
  id          String   @id @default(cuid())
  centreId    String
  recipientId String
  templateId  String
  channel     String
  payload     Json
  status      String   @default("PENDING") // PENDING, SENT, FAILED, CANCELLED
  scheduledAt DateTime @default(now())
  sentAt      DateTime?
  failReason  String?
  attempts    Int      @default(0)
  @@index([status, scheduledAt])
  @@index([centreId, recipientId])
}
```

Data archival with table partitioning

Growing tables — `AuditEvent`, `AttendanceRecord`, and the future `XPLedger` — should be partitioned by `created_at` using PostgreSQL native RANGE partitioning. The `pg_partman` extension automates partition creation and retention:

```sql
```

```
CREATE EXTENSION pg_partman;

SELECT partman.create_parent(
  p_parent_table := 'public."AuditEvent"',
  p_control := 'created_at',
  p_interval := '1 month',
  p_premake := 3
);

UPDATE partman.part_config
SET retention = '12 months',
    retention_keep_table = true,
    retention_schema = 'archive'
WHERE parent_table = 'public."AuditEvent"';
```

Critical Prisma caveat: Prisma has no native partitioning support. Both `db push` and `prisma migrate` will attempt to drop partition tables they don't recognize. `Lightrun` Manage partitioning DDL entirely outside Prisma via raw SQL scripts or pg_partman, and never run `db push` on partitioned tables.

Transitioning from db:push to Prisma Migrate

The system currently uses `prisma db push` for rapid prototyping. For production stability, transition to `prisma migrate`:

```bash
# Step 1: Generate baseline migration from current schema
npx prisma migrate diff --from-empty --to-schema-datamodel prisma/schema.prisma --script > prisma/migrations/0_init/migra

# Step 2: Mark as already applied (schema exists in DB)
npx prisma migrate resolve --applied 0_init

# Step 3: All future changes go through migrations
npx prisma migrate dev --name add_xp_ledger
```

Use `--create-only` for migrations requiring custom SQL (partial indexes, partitioning, RLS policies), edit the generated file, then apply. `Prisma +2`

Section 3: Authentication, authorization, and the security triad

The existing auth flow

AetherLearn uses **NextAuth.js v5 (Auth.js) with credentials provider and JWT strategy**. The session JWT is stored as an encrypted `HttpOnly` cookie `authjs` `Clerk` (`A256GCM` encryption using `AUTH_SECRET`).

Auth.js On each request, lib/auth.ts decodes the JWT to extract userId, role, and centreId. The lib/rbac.ts helper then checks whether the user's role has the required permission from the **72+ permission set**. Finally, lib/tenancy.ts injects centreId from the session into every database query.

This trio — **auth.ts → rbac.ts → tenancy.ts** — forms the **security triad** that protects every API endpoint. No route handler should execute business logic without passing through all three.

```typescript
// Pattern established in every API route:
const session = await auth();                    // 1. auth.ts: who is this?
if (!session) return unauthorized();
requirePermission(session.user.role, 'MANAGE_COURSES'); // 2. rbac.ts: can they do this?
const centreId = session.user.centreId;          // 3. tenancy.ts: scoped to their centre
const courses = await prisma.course.findMany({
  where: { centreId }                            // NEVER from request body
});
```

JWT session hardening

The current JWT approach has one significant gap: **tokens cannot be revoked before expiry** authjs +2 (e.g., for force-logout after password change or security incident). NextAuth.js The recommended solution is a **session version check** in the jwt callback:

```typescript
```

```
// auth.ts — add session version verification
callbacks: {
  jwt: async ({ token, user }) => {
    if (user) {
      token.sessionVersion = user.sessionVersion;
      token.role = user.role;
      token.centreId = user.centreId;
    }
    // Check revocation on every token refresh
    const dbUser = await prisma.user.findUnique({
      where: { id: token.sub },
      select: { sessionVersion: true, isBlocked: true },
    });
    if (!dbUser || dbUser.isBlocked || dbUser.sessionVersion !== token.sessionVersion) {
      throw new Error('Session revoked');
    }
    return token;
  },
},
session: {
  strategy: 'jwt',
  maxAge: 15 * 60,    // 15-minute token lifetime
  updateAge: 5 * 60,  // Refresh every 5 minutes of activity
},
```

To force-logout a user, increment their `sessionVersion` in the database. Their next token refresh (within 5 minutes) will fail, and they'll be redirected to login. For immediate revocation, add a **Redis blocklist** of revoked JTIs with TTL matching the token's remaining lifetime. Readthedocs

The 72+ permission system and 7-tier hierarchy

The role hierarchy follows a strict precedence: **SUPER_ADMIN > CENTER_ADMIN > CENTER_SUPERVISOR > FINANCE_ADMIN > TEACHER > PARENT > STUDENT**. Each role has an explicit permission set — roles do not automatically inherit all permissions from lower tiers. This is intentional: a FINANCE_ADMIN should not inherit TEACHER's `MANAGE_CURRICULUM` permission.

Additional roles assessment: The vision documents mention Receptionist, IT Manager, Assessor, and Vendor. Rather than adding new tiers (which increases hierarchy complexity), three of these map to existing tiers with **custom permission subsets**:

- **Receptionist** → CENTER_SUPERVISOR tier with permissions limited to `VIEW_STUDENTS`, `MANAGE_ATTENDANCE`, `VIEW_SCHEDULES`, `VIEW_ENROLLMENTS`. Explicitly excluded from `MANAGE_CURRICULUM`, `VIEW_FINANCES`, `MANAGE_GRADES`.
- **IT Manager** → CENTER_ADMIN tier with permissions focused on `MANAGE_USERS`, `MANAGE_SYSTEM_CONFIG`, `VIEW_AUDIT_LOGS`, `MANAGE_INTEGRATIONS`. Excluded from

`MANAGE_FINANCES`, `MANAGE_CURRICULUM`.

- **Assessor** → TEACHER tier with permissions scoped to `CREATE_ASSESSMENTS`, `GRADE_ASSESSMENTS`, `VIEW_STUDENT_PERFORMANCE`. Excluded from `EDIT_CURRICULUM`, `MANAGE_ATTENDANCE`.
- **Vendor → New tier required.** Vendors are external entities needing fundamentally different access: `VIEW_OWN_CONTENT`, `UPLOAD_CONTENT`, `VIEW_CONTENT_ANALYTICS`. No student PII access. Place outside the main hierarchy at the bottom.

Implementation approach: add a `roleTemplate` field to the User model that selects a permission subset within the tier, keeping the 7-tier hierarchy clean while supporting role specialization.

CSRF protection

Server Actions in Next.js 16 have built-in CSRF protection: POST-only enforcement, `SameSite=Lax` cookies by default, and `Origin` header validation against `Host`/`X-Forwarded-Host`. Configure `allowedOrigins` for the CloudFlare tunnel domain:

```javascript
// next.config.js
module.exports = {
  experimental: {
    serverActions: {
      allowedOrigins: ['lms.gagneet.com'],
    },
  },
};
```

API Route Handlers (`route.ts`) do not get automatic CSRF protection. Add `@edge-csrf/nextjs` middleware for all custom API routes.

Rate limiting — defense in depth

Nginx currently handles IP-based rate limiting (**20/min login, 100/min API**). Application-level rate limiting should be added for **user-aware, endpoint-specific** control that Nginx cannot provide. Use `rate-limiter-flexible` with the self-hosted Redis instance (preferred over `@upstash/ratelimit` since AetherLearn is self-hosted, not on Vercel):

Endpoint	Limit	Identifier
Login/signup	5/min	IP address
Password reset	3/hour	IP + email
API reads	100/min	User ID
API writes	30/min	User ID
File uploads	10/hour	User ID
Admin APIs	200/min	User ID

COPPA compliance for under-13 students

AetherLearn serves students who may be under 13, making it subject to COPPA. The **FTC's 2025 COPPA Rule update** (effective June 23, 2025, compliance deadline **April 22, 2026**) tightens requirements significantly. Key technical implementation requirements:

- **Never allow self-registration for under-13 accounts.** Student accounts must be created by a PARENT or SCHOOL role. Add `dateOfBirth` and `isMinor` fields to the User model.
- **Parental consent logging.** Track who consented, when, for which child, and for what data uses. The existing PARENT role and parent-child relationship via `parentId` provides the foundation.
- **Data minimization.** Under-13 accounts should collect only what's educationally necessary. No behavioral analytics, no persistent tracking identifiers.
- **Parent data access.** PARENT role must have full read + delete access to their child's data. The existing parent-student relationship supports this; build a "View My Child's Data" dashboard.
- **Retention limits.** Delete student data when no longer needed for educational purpose (end of enrollment). The ARCHIVED soft-delete pattern needs a companion hard-delete workflow for COPPA compliance.

Input validation — Zod as single source of truth

Establish `modules/[domain]/[domain].schema.ts` files as the canonical validation layer shared between client forms and server routes:

```typescript
```

```typescript
// modules/academic/academic.schema.ts
import { z } from 'zod';

export const createCourseSchema = z.object({
  title: z.string().min(3).max(200),
  slug: z.string().regex(/^[a-z0-9-]+$/, 'Slug must be lowercase alphanumeric with hyphens'),
  description: z.string().max(5000).optional(),
  status: z.enum(['DRAFT', 'PUBLISHED', 'ARCHIVED']).default('DRAFT'),
});

export const recordAttendanceSchema = z.object({
  sessionId: z.string().cuid(),
  records: z.array(z.object({
    studentId: z.string().cuid(),
    status: z.enum(['PRESENT', 'ABSENT', 'LATE', 'EXCUSED']),
    notes: z.string().max(500).optional(),
  })).min(1, 'At least one attendance record required'),
});

export type CreateCourseInput = z.infer<typeof createCourseSchema>;
export type RecordAttendanceInput = z.infer<typeof recordAttendanceSchema>;
```

Always use `.safeParse()` (not `.parse()`) in API routes to return structured errors instead of throwing.

Section 4: Multi-tenancy — the centreId contract

The iron rule

`centreId` MUST come from the session only, never from the request body. This is the single most important security invariant in the codebase. If a route handler accepts `centreId` from user input, an attacker can access another centre's data by substituting the value. The `lib/tenancy.ts` helper enforces this by extracting `centreId` exclusively from the authenticated session.

```typescript
```

```
// lib/tenancy.ts — the enforcement pattern
export function getTenantScope(session: Session) {
  if (!session?.user?.centreId) {
    throw new Error('No centre context in session');
  }
  return { centreId: session.user.centreId };
}

// Usage in every data access layer:
const scope = getTenantScope(session);
const students = await prisma.user.findMany({
  where: { ...scope, role: 'STUDENT' },
});
```

SUPER_ADMIN exception: SUPER_ADMIN users need cross-centre access for analytics and platform management. Handle this explicitly:

typescript

```
export function getTenantScope(session: Session, allowCrosscentre = false) {
  if (session.user.role === 'SUPER_ADMIN' && allowCrosscentre) {
    return {}; // No centreId filter — sees all centres
  }
  if (!session.user.centreId) throw new Error('No centre context');
  return { centreId: session.user.centreId };
}
```

Centre-specific configuration — feature flags and branding

Each centre should support configurable settings without schema changes. Add a `CentreConfig` model using JSONB:

prisma

```
model CentreConfig {
  id        String   @id @default(cuid())
  centreId  String   @unique
  branding  Json     // { logo: "url", primaryColor: "#hex", name: "..." }
  features  Json     // { gamification: true, ticketing: true, documents: false }
  gamification Json  // { xpPerAttendance: 10, xpPerLesson: 25, streakBonusMultiplier: 1.5 }
  subscription String @default("BASIC") // BASIC, PROFESSIONAL, ENTERPRISE
  centre    Centre   @relation(fields: [centreId], references: [id])
}
```

Feature flags control which modules are active per centre — enabling a phased rollout of new capabilities (e.g., documents module available only to ENTERPRISE centres).

Performance implications of centre_id filtering

Every query includes `WHERE centreId = ?`. With **centreId as the leading column in composite indexes**, PostgreSQL resolves this as an equality predicate on the index prefix — the most efficient B-tree access pattern. For a typical centre with thousands of records in a table with millions total, the index reduces the scan to the centre's partition of the B-tree.

Performance degrades when: indexes don't lead with centreId (forces index skip scan or full scan), large centres dominate the dataset (index selectivity drops), or queries join across many tenant-scoped tables without proper indexes on foreign keys. Monitor with `EXPLAIN ANALYZE` on production queries.

Section 5: Role interaction map and cross-domain workflows

Permission matrix — Role × Domain × Access Level

Domain / Action	SUPER_ADMIN	CENTER_ADMIN	CENTER_SUPERVISOR	FINANCE_ADMIN	TEACHE
Centre config	CRUD all	CRUD own	Read	—	—
User management	CRUD all	CRUD own centre	Read own centre	—	—
Courses / Modules	CRUD all	CRUD	Read + Assign	—	CRUD assigned
Lessons / Content	CRUD all	CRUD	Read	—	CRUD assigned
Class Cohorts	CRUD all	CRUD	CRUD	—	Read assigned
Sessions	CRUD all	CRUD	CRUD	—	CRUD assigned
Attendance	CRUD all	CRUD	CRUD	—	CRUD assigned
Catch-up Packages	CRUD all	CRUD	CRUD	—	Create + Read
Gamification (XP/Badges)	CRUD all	Read + Config	Read	—	Read assigned
Fee Plans	CRUD all	CRUD	Read	CRUD	—

Domain / Action	SUPER_ADMIN	CENTER_ADMIN	CENTER_SUPERVISOR	FINANCE_ADMIN	TEACHE
Invoices	CRUD all	CRUD	Read	CRUD	—
Payments / Refunds	CRUD all	Read	—	CRUD	—
Tickets	CRUD all	CRUD	CRUD	Read own	Create + Read
SLA Config	CRUD all	CRUD	Read	—	—
Audit Events	Read all	Read own centre	—	—	—
Approval Requests	CRUD all	CRUD	Create + Read	—	—

Cross-role workflow data flows

Workflow 1: Student completes a lesson

```
Student submits lesson → Progress.update(lessonId, COMPLETED)
  → Event: 'progress.lesson_completed' published
    → Gamification module: XPLedger.create(+25 XP, LESSON_COMPLETE)
    → Gamification module: GamificationProfile.update(totalXp, level check)
    → Gamification module: Badge check (COMPLETION badge criteria met?)
    → Notification module: queue in-app notification to PARENT
  → Teacher dashboard: real-time SSE update shows student progress
  → Supervisor analytics: aggregated completion rates recalculated
```

Tables written: Progress, XPLedger, GamificationProfile, NotificationQueue **Tables read by each role:** Student → own Progress, GamificationProfile; Parent → child's Progress, XPLedger; Teacher → class Progress aggregates; Supervisor → centre-wide analytics

Workflow 2: Teacher marks attendance → absent student triggers catch-up

```
Teacher records attendance → AttendanceRecord.createMany()
  → Event: 'attendance.recorded' published
    → For PRESENT students: Gamification awards +10 XP (ATTENDANCE)
    → For ABSENT students:
      → Academic module: CatchUpPackage.create(sessionId, studentId)
      → Notification module: queue EMAIL + IN_APP to PARENT
      → Academic module: AcademicProfile.update(at-risk flag if absences > threshold)
  → Supervisor dashboard: real-time attendance summary, at-risk alerts
```

Tables written: AttendanceRecord, XPLedger, CatchUpPackage, AcademicProfile, NotificationQueue **Tables read:** Teacher → session roster; Supervisor → centre attendance aggregates; Parent → child's attendance + catch-up packages

Workflow 3: Finance admin creates invoice → parent pays

```
Finance admin creates invoice → Invoice.create() + InvoiceLine.createMany()
  → Event: 'invoice.created' published
    → Notification module: queue EMAIL to PARENT with invoice PDF
  → Parent views invoice → initiates payment
    → Payment.create(invoiceId, amount, method)
    → Invoice.update(status: PAID)
    → StudentAccount.update(balance)
    → Event: 'payment.received' published
      → Governance module: AuditEvent.create(PAYMENT_RECEIVED)
      → Notification module: queue receipt EMAIL to PARENT
```

Tables written: Invoice, InvoiceLine, Payment, StudentAccount, AuditEvent, NotificationQueue

Workflow 4: Parent submits tutor change request

```
Parent creates ticket → Ticket.create(category: COMPLAINTS, type: TUTOR_CHANGE)
  → Event: 'ticket.created' published
    → Operations module: SLA timer starts (response within 24h)
    → Notification module: queue IN_APP to CENTER_SUPERVISOR
  → Supervisor reviews → ApprovalRequest.create(type: TUTOR_REASSIGNMENT)
  → CENTER_ADMIN approves → ClassMembership.update(teacherId)
    → Event: 'membership.updated' published
      → Notification module: notify PARENT, old TEACHER, new TEACHER
      → Ticket.update(status: RESOLVED)
```

Table-level read/write access per role

Table	SUPER_ADMIN	CENTER_ADMIN	SUPERVISOR	FINANCE	TEACHER	PARENT
Centre	R/W	R/W	R	—	R	R

Table	SUPER_ADMIN	CENTER_ADMIN	SUPERVISOR	FINANCE	TEACHER	PARENT
User	R/W	R/W	R	R	R (assigned)	R (own+chil
Course	R/W	R/W	R	—	R/W	—
ClassCohort	R/W	R/W	R/W	—	R	—
AttendanceRecord	R/W	R/W	R/W	—	R/W	R
GamificationProfile	R/W	R	R	—	R	R
Invoice	R/W	R/W	R	R/W	—	R
Payment	R/W	R	—	R/W	—	R
Ticket	R/W	R/W	R/W	R	R/W	R/W
AuditEvent	R	R	—	—	—	—

(R = Read, W = Write, R/W = Read and Write)

Section 6: Project structure, standards, and testing

TypeScript strict mode patterns

The codebase should enforce `strict: true` in `tsconfig.json` with these additional rules:

```json
{
  "compilerOptions": {
    "strict": true,
    "noUncheckedIndexedAccess": true,
    "forceConsistentCasingInFileNames": true,
    "noImplicitReturns": true,
    "exactOptionalPropertyTypes": true,
    "paths": {
      "@/*": ["./src/*"],
      "@/modules/*": ["./src/modules/*"],
      "@/lib/*": ["./src/lib/*"]
    }
  }
}
```

Prisma query patterns — the repository layer

Every module's data access should go through a repository that enforces the tenancy scope:

```typescript
// modules/academic/academic.repository.ts
import { prisma } from '@/lib/prisma';
import { Prisma } from '@prisma/client';

export class AcademicRepository {
  async findCourses(centreId: string, filters?: { status?: string }) {
    return prisma.course.findMany({
      where: {
        centreId,  // Always scoped
        ...(filters?.status && { status: filters.status }),
      },
      include: { modules: { include: { lessons: true } } },
      orderBy: { createdAt: 'desc' },
    });
  }

  async createAttendance(centreId: string, data: Prisma.AttendanceRecordCreateManyInput[]) {
    return prisma.attendanceRecord.createMany({
      data: data.map(record => ({ ...record, centreId })),  // Inject centreId
    });
  }
}
```

API route error handling pattern

Standardize error responses across all routes:

```typescript
```

```typescript
// lib/api-response.ts
import { NextResponse } from 'next/server';
import logger from '@/lib/logger';

export function apiSuccess<T>(data: T, status = 200) {
  return NextResponse.json({ success: true, data }, { status });
}

export function apiError(message: string, status: number, details?: unknown) {
  logger.error({ status, message, details }, 'API error');
  return NextResponse.json({ success: false, error: message, details }, { status });
}

// Usage in route handlers:
export async function GET(req: NextRequest) {
  try {
    const session = await auth();
    if (!session) return apiError('Unauthorized', 401);
    requirePermission(session.user.role, 'VIEW_COURSES');
    const courses = await courseService.list(session.user.centreId);
    return apiSuccess(courses);
  } catch (error) {
    if (error instanceof PermissionError) return apiError(error.message, 403);
    if (error instanceof ValidationError) return apiError(error.message, 422, error.details);
    return apiError('Internal server error', 500);
  }
}
```

Structured logging with Pino

Pino is recommended over Winston for three reasons: 5x throughput (**~50,000 vs ~10,000 logs/sec**), native JSON output for log aggregation, and compatibility with both client and server contexts in Next.js (Winston's `fs` dependency causes issues in client components).

```typescript
```

```
// lib/logger.ts
import pino from 'pino';

const logger = pino({
  level: process.env.LOG_LEVEL || 'info',
  timestamp: pino.stdTimeFunctions.isoTime,
  formatters: {
    level: (label) => ({ level: label }),
  },
  transport: process.env.NODE_ENV !== 'production'
    ? { target: 'pino-pretty', options: { colorize: true } }
    : undefined,
});

export default logger;

// Per-request child logger with correlation ID:
export function requestLogger(requestId: string, path: string) {
  return logger.child({ requestId, path });
}
```

Pino outputs JSON to stdout → PM2 captures it automatically. For human-readable dev logs, pipe through `pino-pretty`. For production log aggregation, use Filebeat or Vector to ship PM2 log files to Loki or ELK.

Testing strategy — three-tier pyramid

Unit tests (Vitest): Test service logic with mocked Prisma client. Fast, run on every commit.

typescript

```
// vitest.config.mts
import { defineConfig } from 'vitest/config';
import tsconfigPaths from 'vite-tsconfig-paths';

export default defineConfig({
  plugins: [tsconfigPaths()],
  test: {
    environment: 'jsdom',
    globals: true,
    include: ['src/**/*.test.ts'],
    coverage: { provider: 'v8', reporter: ['text', 'html'] },
  },
});
```

Integration tests (Vitest + test database): Test Prisma queries against a real PostgreSQL instance, each test wrapped in a rolled-back transaction using `vitest-environment-prisma-postgres`.

E2E tests (Playwright): Test full user workflows across all 7 roles. Use Playwright's **project-based auth** to maintain separate authenticated sessions per role:

```typescript
// playwright.config.ts
const ROLES = ['superadmin', 'admin', 'supervisor', 'finance', 'teacher', 'parent', 'student'];

export default defineConfig({
  projects: [
    ...ROLES.map(role => ({
      name: `setup-${role}`,
      testMatch: /auth\.setup\.ts/,
    })),
    ...ROLES.map(role => ({
      name: role,
      dependencies: [`setup-${role}`],
      use: { storageState: `playwright/.auth/${role}.json` },
      testMatch: new RegExp(`\\.${role}\\.spec\\.ts`),
    })),
    {
      name: 'multi-role',
      dependencies: ROLES.map(r => `setup-${r}`),
      testMatch: /\.multi-role\.spec\.ts/,
    },
  ],
});
```

Multi-role tests create separate browser contexts with different auth states to test cross-role workflows (teacher creates assignment → student sees it → parent reviews grades).

Section 7: Architectural decision records

ADR-001: Next.js 16 modular monolith, not microservices

Status: Accepted **Context:** AetherLearn serves individual learning centres with moderate traffic. The team is small. Microservices add deployment complexity, network latency, distributed transaction challenges, and operational overhead (service discovery, circuit breakers, distributed tracing). **Decision:** Single deployable Next.js application with domain modules. Module boundaries designed for future extraction if needed. **Consequence:** Simpler deployment (single PM2 process group), shared database transactions, faster development. Trade-off: must enforce module boundaries through tooling rather than network isolation. Real-world validation: Shopify's modular monolith and a streaming platform that consolidated Lambda functions into a monolith achieved **90% infrastructure cost reduction**.

ADR-002: Prisma ORM with db:push for rapid development (transitioning to migrations)

Status: Accepted, evolving **Context:** During initial build, `db push` enabled rapid schema iteration. Now with production data, schema changes need reproducibility and safety. **Decision:** Transition to `prisma migrate` with baseline migration. Use `--create-only` for custom SQL needs (partial indexes, partitioning). Maintain `db push` only for local development experimentation. **Consequence:** Migration files tracked in source control. CI/CD runs `prisma migrate deploy`. Zero-downtime changes follow the expand-and-contract pattern.

ADR-003: Shared-database multi-tenancy with application-level isolation

Status: Accepted **Context:** Evaluated three models: database-per-tenant (operational nightmare at scale), schema-per-tenant (complex migrations), shared-database with row-level filtering (simplest). **Decision:** Single database, `centreId` column on all tenant-scoped models, enforced by `lib/tenancy.ts` extracting centreId from session. RLS evaluated and rejected (see Section 2). **Consequence:** Simple schema management, easy cross-centre analytics for SUPER_ADMIN. Requires discipline: every query must include centreId, tested via integration tests and code review.

ADR-004: 7-tier RBAC with 72+ explicit permissions

Status: Accepted **Context:** Role-based access must support educational hierarchy (admin → supervisor → teacher → student) plus financial and parental roles. **Decision:** Seven discrete roles with explicit permission arrays. No automatic permission inheritance between tiers. New role specializations (Receptionist, IT Manager, Assessor) mapped to existing tiers with custom permission subsets. Vendor to be added as 8th tier for external entities. **Consequence:** Fine-grained control. Adding new permissions requires updating role configuration but not the hierarchy. The 72+ permissions cover all CRUD operations across all domains.

ADR-005: Tailwind CSS v3, not v4

Status: Accepted **Context:** Tailwind CSS v4 introduced breaking changes incompatible with Next.js 16 at time of development. **Decision:** Pin to Tailwind CSS v3.x. Monitor Next.js + Tailwind v4 compatibility in future releases. **Consequence:** Stable styling system. May require migration effort when upgrading to Tailwind v4 in the future.

ADR-006: PM2 cluster mode for production

Status: Accepted **Context:** Single-process Node.js doesn't utilize multi-core servers. PM2 cluster mode spawns workers per CPU core with zero-downtime reloads. **Decision:** PM2 cluster mode with auto-restart and exponential backoff. Separate fork-mode processes for event workers (BullMQ). **Consequence:** CPU utilization scales with cores. **Critical constraint: in-memory state is not shared between workers** — all shared state must use Redis. This drives the Redis caching and BullMQ event bus decisions.

ADR-007: CloudFlare tunnel for SSL/CDN

Status: Accepted **Context:** Self-hosted on Ubuntu. Need SSL termination, DDoS protection, and CDN without managing certificates. **Decision:** CloudFlare tunnel from server to CloudFlare edge. SSL terminates at CloudFlare. Nginx reverse proxy on localhost. **Consequence:** Free SSL, DDoS protection, CDN caching for static assets. Slight added latency for non-cached requests routed through CloudFlare's network.

ADR-008 (Proposed): BullMQ event bus for cross-module communication

Status: Proposed **Context:** Modules need to communicate without direct imports (see dependency matrix). PM2 cluster mode requires cross-process messaging. **Decision:** BullMQ (Redis-backed) as the primary async event bus. In-process `EventEmitter` for synchronous same-request side effects. BullMQ workers run as separate PM2 fork-mode processes. **Alternatives rejected:** Custom EventEmitter (doesn't work across PM2 workers), Redis Pub/Sub alone (no persistence, no retry), Kafka/RabbitMQ (overengineered for current scale). **Consequence:** Reliable cross-module events with retry, backoff, and persistence. Requires Redis. Events become the seam for future microservice extraction.

ADR-009 (Proposed): Self-hosted Redis for caching and shared state

Status: Proposed **Context:** PM2 cluster mode invalidates in-memory caches. Leaderboards, dashboard aggregations, and session data need sharing across workers. **Decision:** Self-hosted Redis on the same Ubuntu server (`sudo apt install redis-server`). Use for: BullMQ backing store, leaderboard sorted sets, dashboard cache (60s TTL), rate limiting counters, SSE pub/sub for real-time notifications. Next.js custom `cacheHandler` to share ISR cache across workers. **Alternatives rejected:** Upstash Redis (HTTP-based, 5-50ms latency vs sub-ms for local), KeyDB (unnecessary complexity for current scale), in-memory only (incompatible with PM2 cluster).

ADR-010 (Proposed): MinIO for file storage

Status: Proposed **Context:** Scanned documents, course materials, profile photos, and QR code images need durable storage with secure access. Estimated **10–100GB initially, plan for 500GB**. **Decision:** Self-hosted MinIO on Ubuntu. S3-compatible API enables migration to AWS S3 without code changes. Pre-signed URLs for secure, time-limited access. Direct client-to-MinIO uploads for files >4MB (bypasses Next.js body limit). **Alternatives rejected:** Local filesystem (no pre-signed URLs, no replication, hard to migrate), AWS S3 (recurring cost, added latency from self-hosted server).

ADR-011 (Proposed): SSE + Redis Pub/Sub for real-time updates

Status: Proposed **Context:** Notifications, live attendance updates, and dashboard refreshes need server-to-client push. AetherLearn's real-time needs are exclusively server→client (no bidirectional messaging required). **Decision:** Server-Sent Events via Next.js Route Handlers + Redis Pub/Sub for cross-worker broadcasting. Redis Pub/Sub ensures events published by any PM2 worker reach all SSE-connected clients. **Alternatives rejected:** WebSocket/Socket.io (requires custom server outside Next.js, complex Nginx/CloudFlare config, overkill for unidirectional needs), long polling (wastes connections for frequent updates). **Critical Nginx config:** `proxy_buffering off` and `X-Accel-Buffering: no` headers required for SSE to stream through Nginx.

ADR-012 (Proposed): PostgreSQL full-text search, graduating to Meilisearch

Status: Proposed **Context:** Users need to search courses, students, tickets, and documents. Search volume is moderate. **Decision:** Phase 1: PostgreSQL full-text search with `tsvector` columns and GIN indexes. Handles course/user/ticket search with zero additional infrastructure. Phase 2: Meilisearch (self-hosted, ~128MB RAM) when UX demands typo tolerance, instant search, and faceted filtering — likely triggered by the documents module rollout. **Alternatives rejected:** Algolia (SaaS cost, data leaves self-hosted environment), Elasticsearch (heavy resource requirements for current scale).

Conclusion

AetherLearn's existing architecture — the security triad of `auth.ts/rbac.ts/tenancy.ts`, the 7-tier RBAC with 72+ permissions, the centreId multi-tenancy contract, and the 30+ model schema — forms a **production-validated foundation**. The key evolution is organizational, not fundamental: extracting business logic from route handlers into domain-bounded modules, wiring them together through BullMQ events, and adding Redis as the shared state layer that PM2 cluster mode demands.

Three decisions carry the highest architectural leverage. **First**, the transition from `db push` to `prisma migrate` should happen immediately — it's the prerequisite for safe schema evolution, partitioning, and partial indexes. **Second**, Redis installation unblocks four capabilities simultaneously (event bus, caching, rate limiting, real-time SSE). **Third**, the modular extraction should proceed domain by domain starting with governance (least coupled), validated by Playwright E2E tests after each extraction.

The modular monolith approach is not a stepping stone to microservices — it's the target architecture. The module boundaries and event bus pattern provide an extraction seam if individual modules need independent scaling, but the operational simplicity of a single deployable unit serving the self-hosted Ubuntu/PM2/Nginx stack is a feature, not a limitation. Every ADR above optimizes for this reality: self-hosted infrastructure, small team, moderate scale, and the discipline to enforce boundaries through tooling rather than network isolation.