EduSphere — Document 4: Implementation Roadmap

Phased Build Plan for Claude Code

> **Purpose**: This document is the **execution blueprint** for Claude Code. Each phase builds on the previous one. **Never skip ahead.** Every phase has hard acceptance criteria — the phase is not "done" until ALL criteria pass. Claude Code must run the test/validation commands listed and show green output before proceeding.

> **Scale Target**: 100,000+ concurrent users. Every architectural decision must consider horizontal scaling, connection pooling, caching, and fault isolation from Phase 0.

> **Reference Documents** (must be loaded before any phase begins):
> - DATABASE-SCHEMA.md — single source of truth for all tables, RLS, graph, embeddings
> - API-CONTRACTS-GRAPHQL-FEDERATION.md — single source of truth for all GraphQL types, queries, mutations, subscriptions
> - EduSphere_Claude.pdf — architecture guide and technology decisions
> - EduSphere_DB.pdf — database design deep-dive

Technology Stack — Validated & Locked

Before any code is written, Claude Code must acknowledge these **locked** technology choices. These have been validated for production viability, MIT/Apache licensing, and ecosystem maturity.

Layer	Technology	Version	License	Validation Status
Gateway	Hive Gateway v2	latest (v2.x)	MIT	✅ 100% Federation v2.7 compliance (189/189 tests). Event-driven distributed subscriptions. Runtime log-level switching.
Gateway (Prod Option)	Hive Router (Rust)	latest	MIT	✅ ~1830 RPS, p95 ~48ms. Consider for production upgrade path.
Subgraph Runtime	GraphQL Yoga + NestJS	Yoga 5.x + `@graphql-yoga/nestjs-federation` 3.x	MIT	✅ `YogaFederationDriver` actively maintained (last publish: weeks ago).
Framework	NestJS	11.x	MIT	✅ Enterprise-grade DI, module system, guards, interceptors.
ORM	Drizzle ORM	1.x (beta → stable)	Apache 2.0	✅ Native RLS support via `pgTable.withRLS()`. `defineRelations()` API. Native pgvector support.
Database	PostgreSQL 16+	16.x	PostgreSQL	✅ With Apache AGE 1.5+ and pgvector 0.8+
Graph DB	Apache AGE	1.5.0 (PG16)	Apache 2.0	✅ Cypher queries within PostgreSQL
Vector Search	pgvector	0.8.0	PostgreSQL	✅ HNSW indexes, 768-dim nomic-embed-text
Auth	Keycloak	v26.x	Apache 2.0	✅ OIDC/JWT, multi-tenant realms, JWKS
Messaging	NATS JetStream	latest	Apache 2.0	✅ Event-driven subscriptions, at-least-once delivery
Object Storage	MinIO	latest	AGPLv3	✅ S3-compatible presigned URLs
CRDT/Collab	Yjs + Hocuspocus	latest	MIT	✅ Real-time collaborative editing
AI Layer 1	Vercel AI SDK	v6.x	Apache 2.0	✅ Unified LLM abstraction (Ollama ↔ OpenAI/Anthropic)
AI Layer 2	LangGraph.js	latest	MIT	✅ State-machine agent workflows
AI Layer 3	LlamaIndex.TS	latest	MIT	✅ RAG pipeline, knowledge graph indexing

Layer	Technology	Version	License	Validation Status
Transcription	faster-whisper	latest	MIT	✅ GPU-accelerated speech-to-text
Frontend	React + Vite	React 19 + Vite 6	MIT	✅ TanStack Query v5 for data layer
Mobile	Expo SDK 54	54.x	MIT	✅ Offline-first patterns
Reverse Proxy	Traefik	v3.6	MIT	✅ Auto-discovery, Let's Encrypt, rate limiting
Schema Registry	GraphQL Hive	latest	MIT	✅ Breaking change detection, composition
Telemetry	OpenTelemetry → Jaeger	latest	Apache 2.0	✅ Distributed tracing, Hive Gateway v2 native integration
Monorepo	pnpm workspaces + Turborepo	latest	MIT	✅ Efficient dependency hoisting, parallel builds

Monorepo Structure (Target)

```
edusphere/
├── apps/
│   ├── gateway/              # Hive Gateway v2 configuration
│   ├── subgraph-core/        # Port 4001 — Tenants & Users
│   ├── subgraph-content/     # Port 4002 — Courses, Media, Transcripts
│   ├── subgraph-annotation/  # Port 4003 — Annotation layers
│   ├── subgraph-collaboration/ # Port 4004 — CRDT, real-time
│   ├── subgraph-agent/       # Port 4005 — AI agents
│   ├── subgraph-knowledge/   # Port 4006 — Graph, embeddings, search
│   ├── web/                  # React + Vite SPA
│   ├── mobile/               # Expo SDK 54
│   └── transcription-worker/ # faster-whisper NATS consumer
├── packages/
│   ├── db/                   # Drizzle schema, migrations, seed, graph helpers
│   ├── graphql-shared/       # Shared SDL (scalars, enums, directives, pagination)
│   ├── graphql-types/        # Generated TypeScript types (codegen output)
│   ├── graphql-codegen/      # GraphQL Code Generator configuration
│   ├── eslint-config/        # Shared ESLint rules
│   ├── tsconfig/             # Shared TypeScript configs
│   ├── nats-client/          # Shared NATS JetStream client wrapper
│   └── auth/                 # JWT validation, context extraction, guards
├── infrastructure/
│   ├── docker/
```

```
│   │   ├── postgres-age/       # Custom PG16 + AGE + pgvector Dockerfile
│   │   └── keycloak/           # Keycloak realm import
│   ├── docker-compose.yml      # Full local development stack
│   ├── docker-compose.test.yml # CI/CD test environment
│   └── k8s/                    # Kubernetes manifests (Phase 7)
├── scripts/
│   ├── health-check.sh         # Validates all services are up
│   ├── smoke-test.sh           # E2E smoke tests
│   └── seed.ts                 # Database seeding
├── turbo.json
├── pnpm-workspace.yaml
└── .env.example
```

Agent Orchestration Protocol for Claude Code

Claude Code should operate with the following multi-agent patterns when executing phases:

Progress Reporting (Every 3 minutes)

```
═══════════════════════════════════════════════
📊 PROGRESS REPORT — Phase X.Y — [timestamp]
═══════════════════════════════════════════════
🔵 Active Agents:
  Agent-1 [Architecture]: Building docker-compose.yml — 80% complete
  Agent-2 [Schema]:       Generating Drizzle migrations — running
  Agent-3 [Testing]:      Writing health-check tests — queued

✅ Completed this cycle:
  - Custom PostgreSQL Dockerfile built and tested
  - Keycloak realm configuration imported

⏳ Next actions:
  - Validate all containers start cleanly
  - Run health-check.sh

📈 Phase progress: 65% → estimated 8 min remaining
═══════════════════════════════════════════════
```

Agent Roles

Agent Role	Responsibility	Active Phases
Architect	File structure, configs, docker-compose, monorepo setup	0–1
Schema	Drizzle schemas, migrations, RLS policies, graph setup	1–2

Agent Role	Responsibility	Active Phases
API	GraphQL SDL, resolvers, guards, context	2–4
Test	Unit tests, integration tests, E2E, load tests	ALL
Security	RLS validation, JWT flow, scope enforcement, penetration	ALL
Frontend	React components, hooks, routing, state management	4–6
AI/ML	Embeddings, RAG pipeline, agent workflows	5–6
DevOps	CI/CD, monitoring, scaling, K8s manifests	6–7

Quality Gates (Enforced at every phase boundary)

```bash
# 1. TypeScript compilation (zero errors)
pnpm turbo build --filter='./apps/*' --filter='./packages/*'

# 2. Linting (zero warnings in CI mode)
pnpm turbo lint

# 3. Unit tests (100% pass, coverage thresholds met)
pnpm turbo test -- --coverage

# 4. Schema validation (supergraph composes without errors)
pnpm --filter @edusphere/gateway exec hive-gateway compose

# 5. Docker health (all containers healthy)
./scripts/health-check.sh

# 6. Security scan
pnpm audit --audit-level=high
```

Phase 0: Foundation — "Hello World"

Goal: `docker-compose up` brings up the entire infrastructure stack, and a simple GraphQL query returns a health-check response.

Duration estimate: 1–2 days

Phase 0.1: Monorepo Scaffolding

Tasks:

1. Initialize pnpm workspace with `pnpm-workspace.yaml`
2. Create `turbo.json` with build/lint/test/dev pipelines
3. Set up shared TypeScript config (`packages/tsconfig/`)
4. Set up shared ESLint config (`packages/eslint-config/`) with:
   - `@typescript-eslint/strict-type-checked`
   - `eslint-plugin-drizzle` (prevent N+1 footguns)
   - `eslint-plugin-import` (enforce module boundaries)
5. Create `.env.example` with all environment variables documented
6. Create `packages/graphql-shared/` with:
   - `src/scalars.graphql` — all custom scalars (DateTime, UUID, JSON, Cursor, etc.)
   - `src/enums.graphql` — all shared enums (UserRole, MediaType, etc.)
   - `src/directives.graphql` — @authenticated, @requiresScopes, @policy, @public, @requiresRole, @ownerOnly, @rateLimit
   - `src/pagination.graphql` — PageInfo, ConnectionArgs

Acceptance Criteria:

```bash
# All workspace packages resolve
pnpm install --frozen-lockfile  # exits 0

# TypeScript configs chain correctly
pnpm turbo build --dry-run  # shows correct topology

# Shared GraphQL package builds
pnpm --filter @edusphere/graphql-shared build  # exits 0
```

Phase 0.2: Infrastructure Docker Stack

Tasks:

1. Build custom PostgreSQL image (`infrastructure/docker/postgres-age/Dockerfile`):

```dockerfile
FROM postgres:16-bookworm
# Install Apache AGE 1.5.0 for PG16
# Install pgvector 0.8.0
# Configure shared_preload_libraries = 'age'
```

2. Create `docker-compose.yml` with all services:

- `postgres` — Custom PG16 + AGE + pgvector (port 5432)
- `keycloak` — v26 with dev realm import (port 8080)
- `nats` — JetStream enabled (port 4222)
- `minio` — S3-compatible storage (port 9000, console 9001)
- `jaeger` — Distributed tracing UI (port 16686)
- Networks: `edusphere-net` (bridge)
- Volumes: Named volumes for postgres data, minio data, keycloak data

3. Create Keycloak realm import JSON:
   - Realm: `edusphere`
   - Client: `edusphere-app` (public OIDC client)
   - Roles: `super_admin`, `org_admin`, `instructor`, `student`, `researcher`
   - Test users: admin@edusphere.dev / student@edusphere.dev
   - Client scopes mapping to API scopes: `org:manage`, `org:users`, `course:write`, `media:upload`, `annotation:write`, `agent:write`, `agent:execute`, `knowledge:write`
4. Create `scripts/health-check.sh`:

```
bash

  # Checks: PG accepts connections, AGE extension loads, pgvector loads,
  # edusphere_graph exists, Keycloak realm accessible, NATS healthy,
  # MinIO reachable, Jaeger UI responds
```

5. Create SQL init script (`infrastructure/docker/postgres-age/init.sql`):
   - Per DATABASE-SCHEMA.md §2: Create extensions (uuid-ossp, pgcrypto, age, vector)
   - Create `edusphere_graph` via `SELECT create_graph('edusphere_graph')`
   - Create `edusphere_app` role
   - Grant AGE permissions

Acceptance Criteria:

```
bash
```

```
# Full stack starts
docker-compose up -d  # all containers reach "healthy" state within 60s

# PostgreSQL + extensions verified
docker exec edusphere-postgres psql -U postgres -c "
  SELECT extname FROM pg_extension
  WHERE extname IN ('age', 'vector', 'uuid-ossp', 'pgcrypto')
" | grep -c 'age\|vector\|uuid-ossp\|pgcrypto'  # outputs 4

# AGE graph exists
docker exec edusphere-postgres psql -U postgres -c "
  LOAD 'age';
  SET search_path = ag_catalog;
  SELECT * FROM ag_graph WHERE name = 'edusphere_graph';
" | grep edusphere_graph  # found

# Keycloak realm accessible
curl -sf http://localhost:8080/realms/edusphere/.well-known/openid-configuration | jq .issuer
# → "http://localhost:8080/realms/edusphere"

# NATS healthy
curl -sf http://localhost:8222/healthz  # exits 0

# MinIO reachable
curl -sf http://localhost:9000/minio/health/live  # exits 0

# Health check script passes all checks
./scripts/health-check.sh  # exits 0, all green
```

Phase 0.3: First Subgraph — Core "Hello World"

Tasks:

1. Scaffold `apps/subgraph-core/` as NestJS application:
   - Install: `@nestjs/graphql`, `graphql-yoga`, `@graphql-yoga/nestjs-federation`, `graphql`
   - Configure `YogaFederationDriver` with schema-first approach
   - Minimal schema: `type Query { _health: String! }`
2. Scaffold `apps/gateway/` as Hive Gateway v2 config:
   - `gateway.config.ts` pointing to subgraph-core at `http://localhost:4001/graphql`
   - Supergraph composed from local SDL
3. Add both to `docker-compose.yml` or use `turbo dev`
4. Verify the full path: Client → Gateway (port 4000) → Core subgraph (port 4001)

Acceptance Criteria:

```bash
# Gateway responds to health query
curl -sf http://localhost:4000/graphql \
  -H "Content-Type: application/json" \
  -d '{"query":"{ _health }"}' | jq .data._health
# → "ok"

# Gateway introspection works
curl -sf http://localhost:4000/graphql \
  -H "Content-Type: application/json" \
  -d '{"query":"{ __schema { queryType { name } } }"}' | jq .data.__schema.queryType.name
# → "Query"
```

Phase 1: Data Layer — Schema, Migrations, RLS

Goal: Complete database layer with all 16 tables, RLS policies, Drizzle relations, Apache AGE graph ontology, and pgvector embedding tables. Seeded with development data.

Duration estimate: 2–3 days

Phase 1.1: Drizzle Schema Package

Tasks:

1. Create `packages/db/` with Drizzle ORM configuration
2. Implement **every table** from DATABASE-SCHEMA.md §4–§8:
   - `packages/db/src/schema/_shared.ts` — pk(), tenantId(), timestamps(), softDelete(), enums
   - `packages/db/src/schema/core.ts` — tenants, users
   - `packages/db/src/schema/content.ts` — courses, modules, media_assets, transcripts, transcript_segments
   - `packages/db/src/schema/annotation.ts` — annotations
   - `packages/db/src/schema/collaboration.ts` — collab_documents, crdt_updates, collab_sessions
   - `packages/db/src/schema/agent.ts` — agent_definitions, agent_executions
   - `packages/db/src/schema/embeddings.ts` — content_embeddings, annotation_embeddings, concept_embeddings
3. Enable RLS on all user-facing tables using `pgTable.withRLS()`
4. Define all indexes per DATABASE-SCHEMA.md:
   - B-tree indexes on `tenant_id` + commonly filtered columns
   - GIN indexes on `tags` (jsonb), `metadata` (jsonb)

- HNSW indexes on embedding vectors (cosine distance, m=16, ef_construction=64)
- Partial index on `deleted_at IS NULL` for all soft-delete tables

Reference: DATABASE-SCHEMA.md §3–§8 (exact column names, types, constraints)

Acceptance Criteria:

```bash
# Schema compiles without errors
pnpm --filter @edusphere/db build  # exits 0

# Migration generates successfully
pnpm --filter @edusphere/db exec drizzle-kit generate  # produces migration files

# Migration applies to database
pnpm --filter @edusphere/db exec drizzle-kit migrate  # exits 0

# All 16 tables exist
docker exec edusphere-postgres psql -U postgres -d edusphere -c "
  SELECT COUNT(*) FROM information_schema.tables
  WHERE table_schema = 'public' AND table_type = 'BASE TABLE'
" | grep 16
```

Phase 1.2: Row-Level Security Policies

Tasks:

1. Implement RLS policies per DATABASE-SCHEMA.md §9:
   - Every tenant-isolated table: `USING (tenant_id = current_setting('app.current_tenant')::uuid)`
   - Users table additional policy: super_admin sees all tenants
   - Annotations: owner-only for PERSONAL layer, shared for SHARED/INSTRUCTOR/AI_GENERATED
2. Create `packages/db/src/rls/withTenantContext.ts`:

```typescript
  // Wraps every resolver DB call:
  // SET LOCAL app.current_tenant = '<uuid>';
  // SET LOCAL app.current_user_id = '<uuid>';
  // SET LOCAL app.current_user_role = '<role>';
```

3. Write RLS integration tests that verify:
   - Tenant A cannot read Tenant B's data
   - Student cannot access instructor-only annotations

- Soft-deleted records are invisible

Acceptance Criteria:

```bash
# RLS enabled on all 16 tables
docker exec edusphere-postgres psql -U postgres -d edusphere -c "
  SELECT relname, relrowsecurity FROM pg_class
  JOIN pg_namespace ON pg_namespace.oid = relnamespace
  WHERE nspname = 'public' AND relkind = 'r'
  ORDER BY relname
" # ALL show relrowsecurity = true

# RLS tests pass
pnpm --filter @edusphere/db test -- --testPathPattern=rls  # all green

# Cross-tenant isolation verified
pnpm --filter @edusphere/db test -- --testPathPattern=tenant-isolation  # all green
```

Phase 1.3: Apache AGE Graph Ontology

Tasks:

1. Create `packages/db/src/graph/` with helpers from DATABASE-SCHEMA.md §15:
   - `client.ts` — `executeCypher()`, `addVertex()`, `addEdge()`
   - `ontology.ts` — graph creation, vertex label setup
2. Initialize vertex labels: Concept, Person, Term, Source, TopicCluster
3. Initialize edge labels: RELATED_TO, CONTRADICTS, PREREQUISITE_OF, MENTIONS, CITES, AUTHORED_BY, INFERRED_RELATED, REFERS_TO, DERIVED_FROM, BELONGS_TO
4. All vertices carry: `id` (UUID), `tenant_id`, `created_at`, `updated_at`
5. Edge-specific properties per DATABASE-SCHEMA.md §10

Acceptance Criteria:

```bash
```

```
# Graph ontology loads without errors
pnpm --filter @edusphere/db exec tsx src/graph/ontology.ts  # exits 0

# Vertex labels exist
docker exec edusphere-postgres psql -U postgres -d edusphere -c "
  LOAD 'age'; SET search_path = ag_catalog;
  SELECT * FROM ag_label WHERE graph = (SELECT graphid FROM ag_graph WHERE name = 'edusphere_graph')
" | grep -c 'Concept\|Person\|Term\|Source\|TopicCluster'  # outputs 5

# Graph CRUD operations work
pnpm --filter @edusphere/db test -- --testPathPattern=graph  # all green
```

Phase 1.4: Seed Data

Tasks:

1. Implement seed script per DATABASE-SCHEMA.md §14:
   - Default tenant with known UUID
   - Admin user + student user
   - Sample course with modules
   - Sample media assets
   - Sample knowledge graph vertices and edges
2. Make seed idempotent (uses `onConflictDoNothing`)

Acceptance Criteria:

```
bash

# Seed runs without errors
pnpm --filter @edusphere/db exec tsx src/seed.ts  # exits 0

# Seed is idempotent
pnpm --filter @edusphere/db exec tsx src/seed.ts  # exits 0 (second run)

# Data exists
docker exec edusphere-postgres psql -U postgres -d edusphere -c "
  SELECT COUNT(*) FROM tenants
" | grep 1
```

Phase 2: Core Subgraphs — Auth + Content

Goal: Core and Content subgraphs fully operational with JWT authentication, all queries/mutations per API-CONTRACTS, federation entity resolution working through the gateway.

Duration estimate: 3–5 days

Phase 2.1: Auth Infrastructure

Tasks:

1. Create `packages/auth/`:
   - `jwt-validator.ts` — JWKS fetching from Keycloak, JWT verification
   - `context-extractor.ts` — Extract `tenantId`, `userId`, `role`, `scopes` from JWT claims
   - `graphql-context.ts` — Type-safe GraphQL context interface:

```typescript
interface GraphQLContext {
  tenantId: string;
  userId: string;
  userRole: UserRole;
  scopes: string[];
  isAuthenticated: boolean;
}
```

   - `guards/` — NestJS guards for `@requiresRole`, `@ownerOnly`, `@requiresScopes`

2. Configure Hive Gateway JWT validation:
   - JWKS endpoint: `http://keycloak:8080/realms/edusphere/protocol/openid-connect/certs`
   - `@authenticated` enforcement at gateway level
   - `x-tenant-id` header propagation to subgraphs
   - `@requiresScopes` evaluation against JWT claims
3. Create dev token generation script for testing:

```bash
# scripts/get-dev-token.sh — gets JWT from Keycloak for dev users
```

Acceptance Criteria:

```bash
```

```
# Token generation works
TOKEN=$(./scripts/get-dev-token.sh admin@edusphere.dev)
echo $TOKEN | cut -d'.' -f2 | base64 -d | jq .sub  # shows user ID

# Authenticated query works
curl -sf http://localhost:4000/graphql \
  -H "Authorization: Bearer $TOKEN" \
  -H "Content-Type: application/json" \
  -d '{"query":"{ me { id email } }"}' | jq .data.me
# → returns user object

# Unauthenticated query fails
curl -sf http://localhost:4000/graphql \
  -H "Content-Type: application/json" \
  -d '{"query":"{ me { id email } }"}' | jq .errors[0].extensions.code
# → "UNAUTHENTICATED"
```

Phase 2.2: Core Subgraph — Full Implementation

Tasks:

1. Implement ALL Core subgraph types per API-CONTRACTS §7:
   - `Tenant` entity with `@key(fields: "id")`
   - `User` entity with `@key(fields: "id")`
   - All queries: `me`, `user(id)`, `users(filter, orderBy, pagination)`, `currentTenant`, `tenantBySlug(slug)`
   - All mutations: `updateMyProfile`, `updateUserRole`, `deactivateUser`, `reactivateUser`, `updateTenantSettings`
2. Implement Relay cursor pagination for `users` query
3. Implement all input types, filters, ordering per API-CONTRACTS
4. Wire resolvers to Drizzle via `withTenantContext()`
5. Unit test every resolver

Acceptance Criteria:

```
bash
```

```
# All Core queries respond correctly
pnpm --filter @edusphere/subgraph-core test -- --coverage  # >90% line coverage

# Federation entity resolution works
curl -sf http://localhost:4000/graphql \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"query":"{ me { id displayName role tenant { id name plan } } }"}' | jq .data.me
# → returns complete user with tenant

# Pagination works
curl -sf http://localhost:4000/graphql \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"query":"{ users(first: 2) { edges { node { id } cursor } pageInfo { hasNextPage endCursor } } }"}'
# → returns paginated results
```

Phase 2.3: Content Subgraph — Full Implementation

Tasks:

1. Implement ALL Content subgraph types per API-CONTRACTS §8:
   - `Course`, `Module`, `MediaAsset`, `Transcript`, `TranscriptSegment` entities
   - Entity extensions: `User.createdCourses`, `Tenant.courses`
2. All queries: `course(id)`, `courses(filter)`, `module(id)`, `mediaAsset(id)`, `mediaAssets(filter)`, `segmentsForTimeRange`, `searchTranscripts`
3. All mutations: CRUD for Course, Module, MediaAsset + `forkCourse`, `toggleCoursePublished`, `retriggerTranscription`
4. File upload via multipart (per API-CONTRACTS §14):
   - `initiateMediaUpload` → returns presigned MinIO URL
   - `completeMediaUpload` → confirms upload, triggers transcription pipeline
5. Subscription: `transcriptionStatusChanged(assetId)` via NATS
6. NATS integration:
   - Publish `edusphere.{tenant}.media.uploaded` on upload complete
   - Publish `edusphere.{tenant}.transcription.status` on status change

Acceptance Criteria:

```
bash
```

```
# All Content queries respond correctly
pnpm --filter @edusphere/subgraph-content test -- --coverage  # >90%

# Course CRUD works end-to-end
# Create → Read → Update → SoftDelete → Verify invisible

# Presigned URL generation works
curl -sf http://localhost:4000/graphql \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"query":"mutation { initiateMediaUpload(input: { courseId: \"...\", filename: \"lecture.mp4\", mimeType: \"video/mp4\",
# → returns uploadUrl (MinIO presigned) and assetId

# Federation cross-subgraph resolution works
# Query from gateway that touches both Core (User) and Content (Course)
curl -sf http://localhost:4000/graphql \
  -H "Authorization: Bearer $TOKEN" \
  -d '{"query":"{ courses(first: 5) { edges { node { id title creator { id displayName } } } } }"}' | jq .data
# → returns courses with resolved creator from Core subgraph

# Supergraph composes without errors
pnpm --filter @edusphere/gateway compose  # exits 0
```

Phase 3: Annotation + Collaboration Subgraphs

Goal: Complete annotation system with spatial comments, layers, and thread support. Real-time collaboration via CRDT with presence awareness.

Duration estimate: 3–4 days

Phase 3.1: Annotation Subgraph

Tasks:

1. Implement ALL Annotation types per API-CONTRACTS §9:
   - `Annotation` entity with all types: text, sketch, link, bookmark, spatial_comment
   - `AnnotationLayer` filtering (PERSONAL, SHARED, INSTRUCTOR, AI_GENERATED)
   - Thread support: `parentId`, `annotationThread(rootId)` query
2. All queries: `annotation(id)`, `annotations(filter)`, `annotationThread(rootId)`
3. All mutations: `createAnnotation`, `updateAnnotation`, `deleteAnnotation`, `toggleAnnotationPin`, `resolveAnnotation`, `moveAnnotationsToLayer`
4. Subscription: `annotationChanged(assetId, layers)` via NATS
5. Entity extensions: `User.annotations`, `MediaAsset.annotations`

6. Owner-only enforcement for PERSONAL layer mutations

Acceptance Criteria:

```bash
# Annotation CRUD with layer filtering works
pnpm --filter @edusphere/subgraph-annotation test -- --coverage  # >90%

# Layer-based access control verified
# Student A cannot see Student B's PERSONAL annotations
# Student A CAN see SHARED and INSTRUCTOR annotations

# Thread queries return correct nesting
# Subscription fires on annotation changes
```

Phase 3.2: Collaboration Subgraph

Tasks:

1. Implement ALL Collaboration types per API-CONTRACTS §10:
   - `CollabDocument` entity
   - `CollabSession` entity (presence tracking)
2. All queries: `collabDocument(id)`, `collabDocumentByName`, `collabDocumentsForEntity`, `collabConnectionInfo`
3. Mutations: `createCollabDocument`, `compactCollabDocument`
4. Subscription: `collaboratorPresenceChanged(documentId)`
5. Integrate Hocuspocus server:
   - WebSocket endpoint for Yjs CRDT sync
   - Authentication via JWT
   - Persistence to `crdt_updates` table
   - `collabConnectionInfo` returns WebSocket URL + auth token
6. CRDT compaction logic for `compactCollabDocument`

Acceptance Criteria:

```bash
```

```
# Collaboration CRUD works
pnpm --filter @edusphere/subgraph-collaboration test -- --coverage  # >90%

# Hocuspocus WebSocket server accepts connections
# Two clients can connect and see real-time sync
# Presence subscription shows connected users
# Document compaction reduces CRDT update count

# Full supergraph composes (4 subgraphs)
pnpm --filter @edusphere/gateway compose  # exits 0
```

Phase 4: Knowledge Subgraph — Graph + Embeddings + Search

Goal: Full knowledge graph with semantic search, hybrid search (vector + graph), concept extraction, and contradiction detection.

Duration estimate: 4–5 days

Phase 4.1: Knowledge Graph Resolvers

Tasks:

1. Implement ALL Knowledge subgraph types per API-CONTRACTS §12:
   - `Concept`, `Person`, `Term`, `Source`, `TopicCluster` entities
   - `KnowledgeRelation`, `ConceptMention`, `Contradiction`, `PrerequisiteLink`
2. All CRUD mutations for Concept, Person, Term, Source
3. Relation mutations: `createRelation`, `deleteRelation`, `createContradiction`
4. Graph traversal queries:
   - `relatedConcepts(conceptId, maxDepth)` — Apache AGE Cypher query with configurable depth
   - `contradictions(conceptId)` — find CONTRADICTS edges
   - `learningPath(conceptId, maxDepth)` — PREREQUISITE_OF chain traversal
5. `topicClusters(pagination)` — cluster browsing
6. `reviewInferredRelation` — accept/reject AI-inferred relations

Acceptance Criteria:

```
bash
```

```
# Graph queries return correct results
pnpm --filter @edusphere/subgraph-knowledge test -- --testPathPattern=graph  # all green

# Multi-hop traversal works (2-hop from concept)
# Contradiction detection returns correct edges
# Learning path follows prerequisite chains correctly
```

Phase 4.2: Embedding Pipeline + Semantic Search

Tasks:

1. Implement embedding generation service:
   - Uses Vercel AI SDK → `embed()` function
   - Dev: nomic-embed-text via Ollama (768 dimensions)
   - Prod: OpenAI text-embedding-3-small or equivalent
2. Content embedding pipeline:
   - On transcript segment creation → generate & store embedding in `content_embeddings`
   - On annotation creation → generate & store in `annotation_embeddings`
   - On concept creation → generate & store in `concept_embeddings`
3. Implement `semanticSearch(query, pagination)`:
   - Embed query text → HNSW index search on content_embeddings
   - Return ranked results with similarity scores
4. Implement `hybridSearch(query, graphDepth)`:
   - Parallel: pgvector semantic search + Apache AGE graph traversal (2-hop from vector hits)
   - Fusion: rank by combined vector similarity + graph centrality score
   - Return `HybridSearchResult` with `graphContext[]`
   - Per API-CONTRACTS §19 HybridRAG pattern
5. `reindexAssetEmbeddings` mutation for re-processing
6. Subscription: `conceptsExtracted(assetId)` via NATS

Acceptance Criteria:

```
bash
```

```
# Embedding generation works
pnpm --filter @edusphere/subgraph-knowledge test -- --testPathPattern=embedding  # all green

# Semantic search returns relevant results
# Query "Rambam's view on divine attributes" returns related segments

# Hybrid search fuses vector + graph results
# graphContext[] populated with related concepts and connections

# Full supergraph composes (5 subgraphs)
pnpm --filter @edusphere/gateway compose  # exits 0

# Search latency < 200ms for p95 (with HNSW index)
```

Phase 5: Agent Subgraph — AI Orchestration

Goal: Full AI agent system with LangGraph.js state machines, streaming responses, MCP tool integration, and sandboxed execution.

Duration estimate: 4–5 days

Phase 5.1: Agent CRUD + Execution Engine

Tasks:

1. Implement ALL Agent subgraph types per API-CONTRACTS §11:
   - `AgentDefinition` entity with `AgentConfig` (systemPrompt, toolsEnabled, modelOverride)
   - `AgentExecution` entity with status lifecycle: QUEUED → RUNNING → COMPLETED/FAILED/CANCELLED
2. All queries: `agentDefinition(id)`, `agentDefinitions(filter)`, `agentTemplates`, `agentExecution(id)`, `agentExecutions(filter)`
3. All mutations: `createAgentDefinition`, `updateAgentDefinition`, `deleteAgentDefinition`, `executeAgent`, `cancelAgentExecution`
4. Execution engine:
   - `executeAgent` mutation creates execution record (QUEUED) → publishes to NATS
   - Worker picks up from NATS → runs LangGraph.js workflow → updates status
   - Token-by-token streaming via NATS subjects
5. Subscriptions:
   - `agentExecutionUpdated(executionId)` — status changes
   - `agentResponseStream(executionId)` — real-time token stream

Acceptance Criteria:

```bash
# Agent CRUD works
pnpm --filter @edusphere/subgraph-agent test -- --coverage  # >90%

# Execution lifecycle: QUEUED → RUNNING → COMPLETED
# Cancellation: RUNNING → CANCELLED

# Token streaming via subscription verified
# Status subscription fires on state changes
```

Phase 5.2: LangGraph.js Agent Workflows

Tasks:

1. Implement pre-built LangGraph.js state machines per API-CONTRACTS §19:
   - `chavruta-debate-graph` — Dialectical debate using CONTRADICTS edges
   - `summarize-graph` — Progressive summarization of transcript segments
   - `quiz-assess-graph` — Adaptive quizzing using PREREQUISITE_OF edges
   - `research-scout-graph` — Cross-reference finder with contradiction detection
   - `explain-graph` — Adaptive explanation with prerequisite chains
   - `custom` — User-defined via JSON config
2. Integrate Vercel AI SDK for LLM calls:
   - Dev: Ollama (phi4:14b / llama3.1:8b)
   - Prod: OpenAI / Anthropic (configurable per tenant)
3. MCP tool integration per API-CONTRACTS §19:
   - `knowledge_graph` — Query/create concepts, relations
   - `semantic_search` — Vector search
   - `transcript_reader` — Read segments by time range
   - `annotation_writer` — Create annotations
   - `web_search` — External search (sandboxed)
   - `calculator` / `citation_formatter`

Acceptance Criteria:

```bash
```

```
# Each agent template produces valid output
pnpm --filter @edusphere/subgraph-agent test -- --testPathPattern=workflows  # all green

# Chavruta agent finds contradictions and debates both sides
# Summarizer produces progressive summaries
# Quiz master adapts difficulty based on responses

# MCP tools are correctly sandboxed (no direct DB access)
# Agent execution respects tenant resource limits
```

Phase 5.3: Agent Sandboxing

Tasks:

1. Implement resource limits per tenant plan:
   - FREE: 10 executions/day, 30s timeout, 256MB memory
   - STARTER: 100 executions/day, 60s timeout, 512MB
   - PROFESSIONAL: 1000 executions/day, 120s timeout, 1GB
   - ENTERPRISE: unlimited, 300s timeout, 2GB
2. MCP proxy mediation — agents cannot directly access databases
3. Rate limiting at gateway level per `@rateLimit` directive

Acceptance Criteria:

```
bash

# Resource limits enforced
# Free tier user gets blocked after 10 executions
# Execution timeout kills runaway agent

# Full supergraph composes (6 subgraphs — ALL subgraphs now running)
pnpm --filter @edusphere/gateway compose  # exits 0

# Full API contract coverage:
# 44 queries implemented ✓
# 44 mutations implemented ✓
# 7 subscriptions implemented ✓
```

Phase 6: Frontend — Web Application

Goal: Complete React SPA consuming the federated supergraph with full authentication flow, course management, video player with annotations, semantic search, and AI agent chat.

Duration estimate: 5–7 days

Phase 6.1: React Application Shell

Tasks:

1. Scaffold `apps/web/` with Vite + React 19 + TypeScript
2. Set up TanStack Query v5 with GraphQL client
3. Run GraphQL Code Generator (per API-CONTRACTS §18):
   - Generate TypeScript types from supergraph SDL
   - Generate TanStack Query hooks for all operations
4. Implement authentication flow:
   - Keycloak OIDC redirect login
   - Token refresh cycle
   - Protected routes
5. Core layout: navigation, sidebar, tenant context

Phase 6.2: Course & Content Management

Tasks:

1. Course list with cursor pagination
2. Course detail view with modules
3. Media upload flow (presigned URL → direct MinIO upload → completeMediaUpload)
4. Video player (Video.js v8) with:
   - HLS streaming
   - Transcript display synced to playback
   - Time-range seeking from search results

Phase 6.3: Annotation Layer

Tasks:

1. Annotation sidebar with layer filtering
2. Sketch canvas overlay (Konva.js v10) on video player
3. Text annotation creation/editing
4. Spatial comment placement
5. Thread view for annotation replies
6. Real-time annotation updates via subscription

Phase 6.4: Knowledge & Search

Tasks:

1. Semantic search interface with hybrid search support
2. Knowledge graph visualization (D3.js or Cytoscape.js)
3. Concept detail view with relations, contradictions, prerequisites
4. Learning path visualization

Phase 6.5: AI Agent Interface

Tasks:

1. Agent chat panel with real-time token streaming
2. Agent template selector (Chavruta, Summarizer, Quiz Master, etc.)
3. Agent execution history
4. Side-panel / overlay rendering based on `outputFormat`

Phase 6 Acceptance Criteria:

```bash
# Frontend builds
pnpm --filter @edusphere/web build  # exits 0

# E2E tests pass (Playwright)
pnpm --filter @edusphere/web test:e2e  # all green

# Lighthouse scores
# Performance: >80
# Accessibility: >90
# Best Practices: >90

# Core user flows work:
# Login → Browse courses → Open course → Play video →
# Create annotation → Search content → Chat with AI agent → Logout
```

Phase 7: Production Hardening

Goal: The system is production-ready for 100,000+ concurrent users with monitoring, autoscaling, security hardening, and performance optimization.

Duration estimate: 5–7 days

Phase 7.1: Performance Optimization

Tasks:

1. **Gateway caching**: Persisted queries, response caching at gateway level
2. **Database optimization**:
   - Connection pooling (PgBouncer or built-in pool sizing)
   - Query analysis and index optimization
   - Read replicas for search-heavy queries
3. **CDN/Caching**:
   - Static asset CDN configuration
   - GraphQL response caching headers
   - MinIO CDN integration for media delivery
4. **Subscription optimization**:
   - NATS JetStream consumer groups
   - Subscription deduplication (Hive Gateway v2 feature)

Phase 7.2: Observability & Monitoring

Tasks:

1. **OpenTelemetry instrumentation**:
   - Gateway → all subgraphs → database spans
   - Custom spans for embedding generation, graph queries
2. **Metrics**:
   - Prometheus endpoints on all services
   - Grafana dashboards: QPS, latency p50/p95/p99, error rates, active subscriptions
3. **Alerting**:
   - Error rate > 1% → PagerDuty
   - p95 latency > 500ms → warning
   - Database connection pool exhaustion → critical
4. **Logging**:
   - Structured JSON logging (Pino)
   - Log aggregation (ELK or Loki)
   - Runtime log-level switching (Hive Gateway v2 feature)

Phase 7.3: Security Hardening

Tasks:

1. **Rate limiting**: Per-tenant, per-IP, per-operation limits at gateway

2. **Query complexity analysis**: Block expensive queries (depth > 10, breadth > 100)
3. **Persisted queries only** in production (block arbitrary queries)
4. **CORS**: Strict origin whitelist
5. **CSP headers**: Content Security Policy for web app
6. **Dependency audit**: `pnpm audit --audit-level=high` in CI
7. **Secret management**: Vault or cloud-native secrets (no .env in production)
8. **gVisor sandboxing** for agent executions per API-CONTRACTS §19

Phase 7.4: Kubernetes Deployment

Tasks:

1. Kubernetes manifests (`infrastructure/k8s/`):
   - Deployments for each subgraph (min 2 replicas)
   - HPA: CPU 70% → scale up, min 2, max 20 replicas per subgraph
   - Gateway: min 3 replicas for high availability
   - PostgreSQL: StatefulSet with PVC or managed (RDS/CloudSQL)
   - NATS: JetStream cluster (3 nodes)
   - Keycloak: HA mode (2+ replicas)
2. Traefik v3.6 as ingress controller:
   - Auto TLS via Let's Encrypt
   - Rate limiting middleware
   - Circuit breaker for subgraph failures
3. Helm charts for parameterized deployment
4. CI/CD pipeline (GitHub Actions):
   - Build → Test → Lint → Security scan → Docker build → Deploy to staging → Integration tests → Deploy to production

Phase 7.5: Load Testing

Tasks:

1. k6 or Artillery load test scripts:
   - 10K concurrent users: login → browse → search → chat
   - 50K concurrent users: sustained read queries
   - 100K concurrent users: mixed read/write/subscription
2. Chaos engineering: Kill random pods, verify recovery
3. Database failover testing

Phase 7 Acceptance Criteria:

```bash
# Load test: 100K concurrent users
k6 run scripts/load-test-100k.js
# → p95 latency < 500ms
# → error rate < 0.1%
# → zero dropped subscriptions

# Security scan clean
pnpm audit --audit-level=high  # zero high/critical vulnerabilities

# All monitoring dashboards populated
# Grafana shows all metrics with 24h retention

# Zero downtime deployment verified
# Rolling update with zero failed requests

# Chaos test: Kill 1 of 3 gateway pods
# → zero dropped requests (load balancer reroutes)

# Full E2E test suite passes on staging
pnpm turbo test:e2e  # all green in staging environment
```

Phase 8: Mobile + Advanced Features

Goal: Expo mobile app with offline-first patterns, transcription worker pipeline, and Chavruta (partner learning) real-time features.

Duration estimate: 5–7 days

Phase 8.1: Expo Mobile Application

Tasks:

1. Scaffold `apps/mobile/` with Expo SDK 54
2. Offline-first architecture:
   - SQLite local cache for viewed courses/annotations
   - Queue mutations when offline → sync on reconnect
   - Optimistic UI updates
3. Core screens: Login, Course list, Video player, Annotations, Search, AI Chat
4. Push notifications for agent completion, annotation replies

Phase 8.2: Transcription Worker Pipeline

Tasks:

1. Implement `apps/transcription-worker/`:
   - NATS consumer listening to `edusphere.*.media.uploaded`
   - Downloads media from MinIO
   - Runs faster-whisper for speech-to-text
   - Writes `TranscriptSegment` records with timestamps
   - Triggers embedding generation for each segment
   - Publishes `transcription.status` updates
2. GPU support configuration for faster-whisper
3. Batch processing for multiple concurrent uploads

Phase 8.3: Chavruta (Partner Learning) Features

Tasks:

1. Real-time paired study session:
   - Match two users on the same content
   - Shared annotation canvas via Yjs CRDT
   - AI Chavruta agent mediating discussion
   - Debate mode: present thesis → AI finds contradictions → users argue → synthesize
2. Session recording and summary generation
3. Progress tracking and learning analytics

Phase 8 Acceptance Criteria:

```bash
# Mobile app builds for iOS and Android
pnpm --filter @edusphere/mobile expo build  # exits 0

# Offline mode: create annotation while disconnected
# → reconnect → annotation syncs to server

# Transcription pipeline: upload video → segments appear within 2 minutes
# → embeddings generated for all segments

# Chavruta session: two users see real-time shared canvas
# → AI agent responds to both users' annotations
# → Session summary generated on close
```

Cross-Cutting Concerns (Active in ALL Phases)

Testing Strategy

Test Type	Tool	Coverage Target	When Run
Unit Tests	Vitest	>90% line coverage per package	Every commit
Integration Tests	Vitest + Testcontainers	All DB operations, RLS, graph queries	Every PR
API Tests	SuperTest + GraphQL	All 44 queries, 44 mutations, 7 subscriptions	Every PR
E2E Tests	Playwright	Core user flows (10+ scenarios)	Pre-merge, nightly
Load Tests	k6	100K concurrent users	Weekly, pre-release
Security Tests	OWASP ZAP + custom	No critical/high vulnerabilities	Weekly
Schema Tests	Hive CLI	No breaking changes	Every schema change

Code Quality Standards

- **TypeScript**: `strict: true`, `noUncheckedIndexedAccess: true`, `exactOptionalPropertyTypes: true`
- **ESLint**: `@typescript-eslint/strict-type-checked` with zero warnings in CI
- **Prettier**: Consistent formatting enforced by pre-commit hook
- **Commit convention**: Conventional Commits (`feat:`, `fix:`, `chore:`, etc.)
- **Branch protection**: Require passing CI + code review before merge
- **Dependency updates**: Renovate bot with auto-merge for patch/minor

Security Checklist (Every Phase)

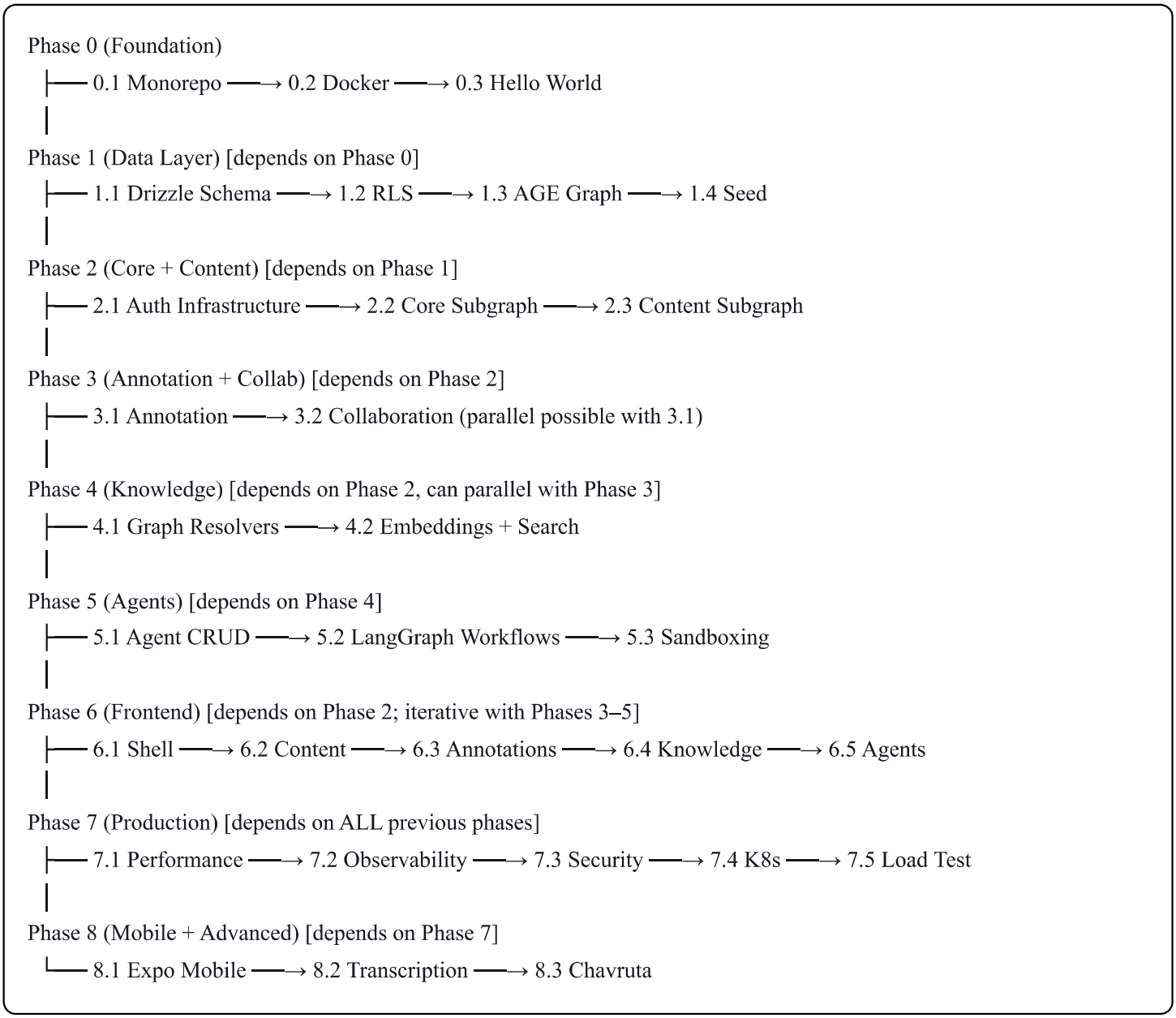
- [ ] No secrets in code or git history
- [ ] RLS policies verified for new tables
- [ ] JWT scopes checked for new mutations
- [ ] Input validation on all user-provided data (via Zod schemas)
- [ ] SQL injection prevented (parameterized queries via Drizzle)
- [ ] GraphQL query depth/complexity limits configured
- [ ] CORS properly configured
- [ ] Rate limiting applied to sensitive operations
- [ ] Audit logging for admin actions

Documentation Requirements (Every Phase)

- [ ] README.md updated for new packages/apps
- [ ] API-CONTRACTS.md updated for schema changes
- [ ] DATABASE-SCHEMA.md updated for table changes
- [ ] OpenAPI/GraphQL introspection endpoint accessible
- [ ] Architecture Decision Records (ADRs) for significant decisions

---

Phase Dependency Graph



Parallelization Opportunities

- **Phase 3 + Phase 4**: Annotation and Knowledge subgraphs can be developed in parallel (both depend on Phase 2 but not each other)
- **Phase 6.1–6.2 + Phase 3–5**: Frontend shell and content UI can start while remaining subgraphs are built
- **Phase 7.2 (Observability)**: Can start alongside Phase 5–6 for incremental monitoring setup

- **Phase 8.1 (Mobile) + Phase 8.2 (Transcription)**: Fully parallel development paths

---

Quick Start Command for Claude Code

When starting a new session, Claude Code should run:

```bash
# 1. Load all reference documents
cat DATABASE-SCHEMA.md API-CONTRACTS-GRAPHQL-FEDERATION.md

# 2. Check current progress
git log --oneline -20
ls -la apps/ packages/

# 3. Determine current phase by checking acceptance criteria
./scripts/health-check.sh 2>/dev/null
pnpm turbo build 2>/dev/null
pnpm turbo test 2>/dev/null

# 4. Resume from the first failing acceptance criterion
```

Estimated Total Duration

Phase	Duration	Cumulative
Phase 0: Foundation	1–2 days	1–2 days
Phase 1: Data Layer	2–3 days	3–5 days
Phase 2: Core + Content	3–5 days	6–10 days
Phase 3: Annotation + Collab	3–4 days	9–14 days
Phase 4: Knowledge	4–5 days	13–19 days
Phase 5: Agents	4–5 days	17–24 days
Phase 6: Frontend	5–7 days	22–31 days
Phase 7: Production	5–7 days	27–38 days
Phase 8: Mobile + Advanced	5–7 days	32–45 days

Total: 32–45 working days (with parallelization: ~25–35 days)

> **CRITICAL REMINDER FOR CLAUDE CODE**:
>
> - **Never skip phases.** Each phase builds on the previous one.
> - **Run acceptance criteria before proceeding.** Green output = permission to advance.
> - **Report progress every 3 minutes** using the format above.
> - **Reference API-CONTRACTS and DATABASE-SCHEMA** for every type, field, and resolver.
> - **No deviation** from the locked technology stack without updating this document.
> - **Test everything.** No untested code enters the repository.