HIPAA-Compliant Encrypted Medical Chatbot

Detailed Day-by-Day Sprint Plan & Task Breakdown

Executive Summary

This sprint plan provides a comprehensive, day-by-day breakdown of all development, integration, testing, and delivery tasks required to build a functional, demo-ready HIPAA-compliant encrypted medical chatbot for the CyborgDB Healthcare Hackathon. The plan is structured around core technical workflows, quality assurance checkpoints, and clear deliverables aligned with hackathon judging criteria (Technology Application, Business Value, Presentation, and Originality).

Sprint Structure Overview

The sprint is organized into **5 core phases** with explicit task sequencing, interdependencies, and quality gates:

1. **Phase 1: Foundation & Data Infrastructure** (Days 1-2)
2. **Phase 2: De-Identification & Embedding Pipeline** (Days 3-4)
3. **Phase 3: Encrypted Vector Storage & Retrieval** (Days 5-6)
4. **Phase 4: Backend API & LLM Integration** (Days 7-8)
5. **Phase 5: Frontend, Testing, & Demonstration** (Days 9-10)

Each phase includes:

- Detailed task list with acceptance criteria
- Integration checkpoints to validate work
- Clear success metrics and completion criteria
- Risk mitigation strategies
- Quality assurance validation steps

Phase 1: Foundation & Data Infrastructure

Objective

Establish the development environment, data sources, and foundational architecture to support all downstream work.

Task 1.1: Development Environment Setup

Description:
Set up local and cloud-based development environments with all required tools, libraries, and services containerized for reproducibility.

Specific Instructions:

- Install Docker and Docker Compose on development machine
- Create a `.gitignore` file excluding sensitive files (credentials, keys, `.env` files)
- Initialize Git repository with README template
- Set up AWS/Azure/GCP account for cloud services (KMS, storage, compute)
- Configure AWS KMS or HashiCorp Vault for key management (test mode, no production keys)
- Create environment configuration template (`.env.example`) with all required variables:
    - Database credentials
    - Auth0/Cognito tenant details (test credentials)
    - CyborgDB connection string
    - LLM model paths
    - Encryption key paths
- Document all setup steps in `SETUP.md`

Deliverable:

- Fully functional local development environment (Docker Compose)
- Reproducible setup instructions
- Environment configuration template
- Git repository initialized with initial commit

Completion Criteria:

- Docker Compose starts all services without errors
- All containers communicate via internal network
- Environment variables correctly injected
- Setup documented and tested on clean machine (second team member runs setup successfully)

Success Metrics:

- Zero dependency conflicts
- All services health-checked and running
- Documentation clarity score (new developer can set up in <30 minutes)

Task 1.2: Synthetic FHIR Dataset Generation and Validation

Description:
Generate or obtain a realistic synthetic FHIR dataset containing 100-500 patient records with diverse clinical documents (discharge summaries, clinical notes, lab reports, medications, problem lists).

Specific Instructions:

- Download Synthea or use SMART on FHIR test data to generate synthetic patient records
- Alternative: Use publicly available de-identified FHIR datasets (NIH, CDC resources)
- Validate generated FHIR JSON against official HL7 FHIR schema (using FHIR Validator tool)
- Create diverse clinical document scenarios:
  - At least 10 variations of clinical note templates
  - Lab result variations (normal, abnormal, critical ranges)
  - Medication lists with dosages and frequencies
  - Problem lists with ICD-10 codes
  - Discharge summaries with clinical narratives
- Store dataset in version-controlled location with clear data dictionary
- Document data schema, field mappings, and clinical context for embeddings team

Deliverable:

- Synthetic FHIR dataset (JSON format, 100-500 records)
- Data dictionary documenting all fields and clinical meanings
- FHIR schema validation report
- Sample records (3-5 patients with multiple documents each)

Completion Criteria:

- All records pass FHIR schema validation
- Dataset contains at least 500+ total clinical documents
- Clinical notes contain realistic medical terminology and concepts
- No real patient data (all synthetic/de-identified)
- Data dictionary complete with field descriptions

Success Metrics:

- FHIR validation pass rate: 100%
- Dataset size: ≥500 clinical documents
- Diversity score: ≥8 different document types
- Schema documentation completeness: 100%

Task 1.3: Architecture Design and Documentation

Description:
Document the complete system architecture, including data flow diagrams, security boundaries, component interactions, and deployment topology.

Specific Instructions:

- Create architecture diagram showing:
  - Data ingestion layer (Prefect/Airflow)
  - De-identification layer (Presidio)
  - Embedding generation layer (BioBERT/ClinicalBERT)
  - Encryption layer (AES-256-GCM)
  - CyborgDB storage layer
  - Backend API (FastAPI)
  - LLM inference service

    - Frontend UI
    - Auth0/Cognito integration
    - Audit logging layer
- Define security boundaries (trust boundaries, encryption domains)
- Document data flow for each workflow (query, ingestion, embedding, retrieval, LLM call)
- Create component interaction matrix (which components talk to each other)
- Document all APIs/interfaces between components
- Create deployment topology diagram (local dev vs. production)
- Document assumptions, constraints, and risk factors

Deliverable:

- System architecture diagram (Lucidchart, draw.io, or ASCII art)
- Data flow diagrams (DFDs) for 3+ core workflows
- Component interaction matrix
- API interface specifications (input/output, errors, formats)
- Deployment topology documentation
- Risk assessment and mitigation strategies

Completion Criteria:

- Architecture diagram includes all 10+ major components
- All data flows documented with encryption/access control points marked
- Security boundaries clearly defined
- At least 3 critical workflows documented
- Documentation accessible and understandable to non-technical stakeholders

Success Metrics:

- Diagram completeness: 100% coverage of components
- Technical accuracy: Reviewed and approved by lead architect
- Clarity score: Non-technical team member understands high-level flow
- Risk identification: At least 5 risks documented with mitigations

---

Task 1.4: Repository and Documentation Structure

Description:
Create a well-organized repository structure that supports team collaboration, code sharing, and documentation accessibility.

Specific Instructions:

- Create directory structure:
    - `/backend` - FastAPI application code
    - `/data-pipeline` - Prefect/Airflow orchestration
    - `/embeddings` - Embedding generation scripts
    - `/encryption` - Encryption utilities
    - `/frontend` - React/HTML UI
    - `/tests` - Test suites (unit, integration, load)
    - `/docs` - Documentation (architecture, APIs, setup)
    - `/benchmarks` - Performance testing and results

    - /config - Configuration templates
    - /scripts - Utility and deployment scripts
- Create README.md with project overview, quick start, and key links
- Create CONTRIBUTING.md with code style guidelines and PR process
- Create ARCHITECTURE.md with detailed system design
- Create API_SPEC.md with endpoint documentation
- Create SETUP.md with environment setup instructions
- Create TESTING.md with testing approach and how to run tests
- Create DEPLOYMENT.md with deployment instructions
- Initialize project board (GitHub Projects or equivalent) with all sprint tasks
- Set up continuous integration (GitHub Actions, GitLab CI, or equivalent):
    - Linting and type checking on PR
    - Unit tests on every commit
    - Build verification

Deliverable:

- Complete repository structure with all directories
- Comprehensive documentation files
- Project board with all tasks linked to phases
- CI/CD pipeline configured and tested
- Code style guide and contribution guidelines

Completion Criteria:

- All directories created and documented
- README is clear and actionable
- Documentation files complete (at least 80% content)
- CI/CD pipeline runs successfully on test PR
- Project board synchronized with sprint plan

Success Metrics:

- Repository usability: New team member can clone and understand structure in <20 minutes
- Documentation completeness: All critical information documented
- CI/CD coverage: ≥80% of code paths included in automated checks
- Collaboration efficiency: Pull request review process established and documented

---

Phase 2: De-Identification & Embedding Pipeline

Objective

Build and validate the data cleaning and de-identification pipeline that prepares clinical data for embedding generation while ensuring no PHI leakage.

Task 2.1: PHI Detection and Masking Implementation

Description:
Implement a comprehensive PHI detection and masking pipeline using Presidio and spaCy that identifies and anonymizes all Protected Health Information in clinical documents.

Specific Instructions:

- Set up Presidio:
  - Install Presidio package
  - Configure analyzers for entity detection (NAME, PHONE_NUMBER, EMAIL_ADDRESS, MEDICAL_RECORD_NUMBER, DATE_TIME, SSN, CREDIT_CARD, IP_ADDRESS, etc.)
  - Tune Presidio confidence thresholds (recommended: 0.8+ for high confidence)
  - Create custom regex patterns for institution-specific identifiers (internal patient IDs, medical record numbers, specimen IDs)
- Implement additional NER using spaCy:
  - Load spaCy medical NER models (e.g., clinical NER trained on clinical notes)
  - Detect clinical entities (medication names, dosages, procedures, diagnoses)
  - Create custom spaCy components for domain-specific masking rules
- Build masking logic:
  - Replace names with [PATIENT_NAME_XXX] tokens
  - Replace dates with relative indicators (e.g., "3 days post-op" instead of specific date)
  - Replace phone numbers with [PHONE_XXX]
  - Replace addresses with [ADDRESS_XXX]
  - Preserve clinically relevant information (diagnoses, medications, lab values)
- Create de-identification token mapping (separate secure storage):
  - Map each token back to original value (stored securely, not accessible to chatbot)
  - Include timestamp, document ID, and masking rule applied
- Implement validation:
  - Verify no original names appear in de-identified text
  - Verify no full dates appear (only relative dates)
  - Verify clinically relevant data preserved
  - Manual review of sample outputs
- Create configuration file for masking rules (Entity type → Mask pattern)
- Implement error handling and logging:
  - Log failed detection attempts
  - Flag documents with potential PHI leakage
  - Alert on unusual patterns

Deliverable:

- Presidio + spaCy integrated PHI detection module
- Masking configuration file
- De-identification token mapping system
- Validation and verification scripts
- Sample de-identified documents (before/after comparison)
- Comprehensive logging and audit trail

Completion Criteria:

- ≥95% PHI detection accuracy on test dataset
- Zero detected PHI in de-identified output (manual spot-check on 50+ samples)
- All clinical information preserved (diagnoses, medications, lab values)
- Masking rules consistently applied
- Token mapping traceable and secure
- Documentation of masking rules and exceptions

Success Metrics:

- PHI detection recall: ≥95%
- PHI detection precision: ≥90%
- De-identification completeness: 100% (no missed PHI)
- Processing speed: ≥1000 documents/hour
- Error rate: <1%

Task 2.2: Data Pipeline Orchestration (Prefect/Airflow)

Description:
Implement a scalable, monitored data pipeline that orchestrates ingestion, validation, de-identification, and storage of clinical data.

Specific Instructions:

- Choose orchestration platform:
  - Recommend Prefect for simpler orchestration with modern Python API
  - Alternative: Apache Airflow for more complex DAGs
- Define pipeline flow:
  - **Ingest task**: Read FHIR JSON from source (synthetic dataset)
  - **Validate task**: Validate FHIR schema, check data quality
  - **De-identify task**: Apply Presidio + spaCy masking
  - **Verify task**: Validate de-identification completeness
  - **Store task**: Write de-identified data to secure storage
  - **Index task**: Create searchable indexes for later retrieval
- Implement error handling:
  - Retry logic for transient failures (max 3 retries)
  - Alert on pipeline failures
  - Fallback to previous successful state
  - Detailed error logging with context
- Implement monitoring:
  - Log all task start/end times
  - Track data volume at each stage
  - Monitor for data quality regressions
  - Alert on SLO breaches (e.g., >5min per batch)
- Create pipeline configuration:
  - Batch size, retry counts, timeouts
  - Storage locations (staging, production)
  - Notification endpoints (Slack, email)
- Implement data lineage tracking:
  - Record which source documents contributed to each de-identified record
  - Track transformations applied
  - Enable audit trail for compliance

- Create dashboard or status page:
    - Pipeline execution history
    - Success/failure rates
    - Data volume metrics
    - Performance trends

Deliverable:

- Prefect/Airflow pipeline definition
- Error handling and retry logic
- Monitoring and alerting setup
- Pipeline configuration files
- Data lineage tracking system
- Status dashboard or monitoring interface
- Pipeline execution logs

Completion Criteria:

- Pipeline successfully processes entire synthetic dataset without errors
- Data quality validated at each stage
- De-identification verified end-to-end
- Monitoring active and alerting works
- Error handling tested with simulated failures
- Performance meets SLOs (complete batch in <10 minutes)

Success Metrics:

- Pipeline success rate: 100%
- Data quality validation: All records pass
- Processing latency: <10 min for 500 documents
- Error recovery: Automatic recovery 100% of time
- Monitoring coverage: All critical steps monitored

Task 2.3: De-Identification Validation and Compliance Testing

Description:
Establish comprehensive validation and testing procedures to verify that de-identification meets privacy standards and no PHI is inadvertently exposed.

Specific Instructions:

- Develop validation test suite:
    - Automated script that searches de-identified text for known PHI patterns (regex for phone, SSN, email, dates)
    - Check for statistical reuse risks (unusual combinations of age, gender, rare conditions that could re-identify)
    - Verify clinical data integrity (diagnoses preserved, medications present, labs intact)
    - Cross-reference with original documents (spot-check sample pairs)
- Create manual review process:
    - Randomly select 5% of de-identified documents for manual review
    - Clinical reviewer checks for clinical accuracy and PHI presence

    - Document any exceptions or edge cases
- Implement re-identification risk assessment:
    - Analyze quasi-identifiers (age, gender, medical conditions)
    - Estimate k-anonymity (how many people could theoretically re-identify a record)
    - Target k-anonymity ≥5 (at least 5 people with same characteristics)
- Create compliance checklist:
    - HIPAA Safe Harbor compliance: All 18 identifiers removed/masked
    - De-identification Standard compliance: Expert determination risk model
    - Clinical utility validation: Medical information preserved for decision support
    - Data integrity verification: No data corruption or loss
- Document all validation results:
    - Validation report with metrics and findings
    - Exception log with explanations
    - Remediation steps for any issues found
- Establish ongoing monitoring:
    - Regular validation runs (e.g., weekly)
    - Alerts if compliance metrics degrade
    - Feedback loop for continuous improvement

Deliverable:

- Automated validation test suite
- Manual review protocol and checklist
- Re-identification risk assessment report
- HIPAA compliance verification document
- Validation results and metrics
- Exception handling and remediation log
- Ongoing monitoring setup

Completion Criteria:

- 100% of documents pass automated PHI detection
- 0 instances of detected PHI in de-identified text
- ≥95% of manually reviewed samples approved
- K-anonymity ≥5 verified for demographic groups
- Clinical data integrity: ≥99% preservation
- All HIPAA Safe Harbor requirements met

Success Metrics:

- De-identification completeness: 100%
- PHI evasion: 0 instances
- Clinical utility: ≥99% information retention
- K-anonymity compliance: ≥5 threshold met
- Manual review approval rate: ≥95%
- Validation coverage: All 500 documents validated

Phase 3: Encrypted Vector Storage & Retrieval

Objective

Implement embedding generation, encryption, and storage in CyborgDB with validated encrypted retrieval capabilities.

Task 3.1: Embedding Generation and Optimization

Description:
Generate high-quality dense embeddings from de-identified clinical documents using pre-trained medical language models (BioBERT, ClinicalBERT, PubMedBERT).

Specific Instructions:

- Select embedding model:
    - Evaluate BioBERT, ClinicalBERT, PubMedBERT on domain relevance
    - Create comparison table: model size, embedding dimension, inference speed, VRAM requirements
    - Recommendation: Start with ClinicalBERT (MIMIC-trained, clinical-optimized)
- Set up embedding infrastructure:
    - Download model from Hugging Face
    - Configure GPU allocation (if available) or CPU inference
    - Test inference speed on sample documents (target: <1 second per document)
    - Implement batch processing to maximize throughput
- Implement embedding generation pipeline:
    - Tokenize clinical documents (handle long documents >512 tokens):
        - Strategy 1: Truncate to first 512 tokens
        - Strategy 2: Split into overlapping chunks, aggregate embeddings (e.g., mean pooling)
        - Strategy 3: Use hierarchical attention (summarize sections first)
    - Generate embeddings (output: 768-dim for ClinicalBERT)
    - Store embeddings with metadata (document type, patient ID, date range, section)
- Optimize for performance:
    - Batch size tuning (start with 32, adjust based on VRAM)
    - Parallel processing (if multi-GPU available)
    - Model quantization (FP32 → FP16) to reduce inference latency
    - Caching mechanism for repeated embeddings
- Validate embedding quality:
    - Spot-check: Retrieve semantically similar documents for sample queries
    - Verify embedding dimension consistency
    - Check for numerical stability (no NaN/Inf values)
    - Analyze embedding distribution (should be roughly unit normal)
- Create embedding index configuration:
    - Metadata schema (patient_id, document_type, date_range, etc.)
    - Embedding normalization (L2 norm recommended for cosine similarity)
    - Index versioning (model name, date, parameters)

Deliverable:

- Embedding generation module (Python script or service)

- Model comparison and selection report
- Performance optimization documentation
- Embedding quality validation report
- Metadata schema definition
- Configuration files for embedding generation

Completion Criteria:

- All 500+ documents successfully embedded
- Embedding dimension: 768-1024 (model-dependent)
- Zero NaN/Inf values in embeddings
- Processing speed: ≥100 documents/hour
- Embedding consistency verified (same document → identical embedding)
- Quality validation passed (spot-checked semantic relevance)

Success Metrics:

- Embedding coverage: 100% of documents
- Processing throughput: ≥100 docs/hour
- Quality score (semantic similarity spot-check): ≥80% relevant matches
- Numerical stability: 0 NaN/Inf values
- Model consistency: Deterministic output verified

Task 3.2: Encryption Implementation and Key Management

Description:
Implement client-side encryption of embeddings using strong cryptographic algorithms and establish secure key management procedures.

Specific Instructions:

- Select encryption algorithm:
  - Recommended: AES-256-GCM (authenticated encryption, prevents tampering)
  - Alternative: ChaCha20-Poly1305 (lighter weight)
  - Avoid: ECB mode, weak key sizes
- Implement encryption workflow:
  - Generate random 256-bit encryption key for embedding batch
  - Encrypt embedding vector + metadata together:
    - Format data as JSON (embedding, document_type, patient_id, date_range)
    - Serialize to bytes
    - Generate random 96-bit nonce (IV)
    - Apply AES-256-GCM: (plaintext, key, nonce) → ciphertext + authentication_tag
    - Store ciphertext, nonce, authentication tag, algorithm version
  - Implement envelope encryption:
    - Data key: Random 256-bit key used to encrypt embeddings (short-lived, batch-specific)
    - Master key: Stored in secure key management service (AWS KMS, HashiCorp Vault, Azure Key Vault)
    - Encrypt data key with master key, store alongside ciphertext

        - On retrieval, decrypt data key using master key, then decrypt embeddings
- Set up key management infrastructure:
    - AWS KMS setup:
        - Create KMS key for healthcare data (with key policy restricting access)
        - Enable automatic key rotation (annual)
        - Test key creation, encryption, decryption workflows
    - Alternatively, HashiCorp Vault:
        - Deploy Vault instance (or use managed service)
        - Configure transit engine for encryption
        - Set up authentication and authorization policies
- Implement key rotation:
    - Establish rotation schedule (quarterly recommended)
    - Plan for re-encryption of existing data (or keep multi-key support)
    - Create automated rotation task with testing
- Implement access controls:
    - Only authenticated backend service can access master keys
    - Log all key access requests
    - Restrict key operations (encrypt, decrypt) by role
    - Test failure scenarios (key unavailable, permissions denied)
- Create key material management procedures:
    - Secure random number generation (use `secrets` module or OS entropy)
    - No hardcoded keys in source code
    - Secure key storage (not in environment variables, but in secure vaults)
    - Document key lifecycle (creation, rotation, retirement)

Deliverable:

- Encryption module (Python crypto wrapper)
- Key management infrastructure setup
- Envelope encryption implementation
- Key rotation automation script
- Access control policies
- Key lifecycle documentation
- Test suite for encryption/decryption

Completion Criteria:

- All embeddings encrypted before storage
- Encryption verification: Attempt to read encrypted data without key fails
- Key management operational (keys stored in secure vault)
- Key rotation tested (old keys phased out, new keys active)
- Access logs show all key operations
- No plaintext key material in code/logs/configs

Success Metrics:

- Encryption coverage: 100% of embeddings encrypted
- Key security: Master keys in secure vault (not in source code)
- Access control: Only authorized service can decrypt
- Key rotation: Automated and tested
- Audit trail: All key operations logged

Task 3.3: CyborgDB Integration and Encrypted Storage

Description:
Integrate with CyborgDB for encrypted vector storage and establish encrypted similarity search capabilities.

Specific Instructions:

- Set up CyborgDB:
  - Download/deploy CyborgDB instance (local for dev, cloud-managed for production planning)
  - Configure encryption mode (homomorphic encryption or secure MPC per CyborgDB options)
  - Set up authentication (API keys or certificates)
  - Test basic operations (connect, create collection, insert, query)
- Design collection schema:
  - **Collection name**: `patient_embeddings`
  - **Fields**:
    - `id` (UUID): Unique embedding identifier
    - `patient_id` (encrypted string): Pseudonymized patient identifier
    - `document_type` (encrypted string): Type of document (note, lab, discharge summary)
    - `date_range` (encrypted string): Date range in relative format
    - `embedding` (encrypted vector): 768+ dimensional embedding
    - `metadata` (encrypted JSON): Additional context (encounter_id, visit_type, etc.)
    - `created_at` (timestamp): Creation time
    - `updated_at` (timestamp): Last update
    - `access_log` (encrypted array): List of access events
  - **Indexes**: (patient_id, document_type) for filtering before similarity search
- Implement insert operations:
  - Batch insert encrypted embeddings (recommended: batches of 100-1000)
  - Handle duplicates (update if exists, insert if new)
  - Validate schema compliance before insert
  - Test transaction rollback on failure
  - Monitor insert latency (target: <100ms per embedding)
- Implement encrypted search operations:
  - Query with plaintext query embedding (generated from clinician question)
  - CyborgDB performs similarity search on ciphertext:
    - Compute similarity (cosine or Euclidean) without decrypting
    - Return top-k results with highest similarity scores
  - Implement filtering:
    - Filter by patient_id (only matching patient's records)
    - Filter by document_type (prioritize recent clinical notes)
    - Filter by date range (recent documents first)
  - Test query latency (target: <500ms for k=10 from 500+ documents)
- Implement retrieval and decryption:
  - Retrieve encrypted embeddings from CyborgDB results
  - Decrypt in backend memory (never write plaintext to disk)
  - Decrypt metadata to reconstruct context
  - Delete decrypted data from memory after processing

- Implement error handling:
  - Connection failures: Retry with exponential backoff
  - Decryption failures: Log error, alert operator, graceful degradation
  - Data integrity: Validate authentication tags on all encrypted data
- Create monitoring and logging:
  - Log all search operations (query, results count, latency)
  - Monitor query patterns for suspicious activity (bulk extractions, unusual access)
  - Alert on anomalies
- Document CyborgDB integration:
  - API usage patterns
  - Encryption configuration details
  - Performance characteristics
  - Known limitations and workarounds
  - Integration insights for post-campaign feedback

Deliverable:

- CyborgDB SDK integration module
- Collection schema definition
- Insert/search/retrieval implementation
- Error handling and retry logic
- Monitoring and logging setup
- Integration documentation
- Performance benchmarks
- CyborgDB integration feedback report

Completion Criteria:

- CyborgDB instance operational and tested
- 500+ embeddings successfully stored and encrypted
- Encrypted search functional (queries return top-k relevant results)
- Query latency <500ms for k=10
- Data integrity verified (decryption successful)
- Zero plaintext embeddings on disk
- Monitoring active and alerting functional

Success Metrics:

- Storage coverage: 100% of embeddings in CyborgDB
- Search latency: <500ms per query
- Search accuracy: Spot-check relevance of top-k results (≥80% relevant)
- Error rate: <1%
- Monitoring coverage: All critical operations logged

Phase 4: Backend API & LLM Integration

Objective

Build a secure FastAPI backend that orchestrates authentication, encrypted retrieval, and private LLM-based answer generation.

Task 4.1: FastAPI Backend Scaffold and Security Infrastructure

Description:
Implement the core FastAPI application with security middleware, authentication, and foundational endpoint structure.

Specific Instructions:

- Initialize FastAPI application:
    - Create main app instance
    - Configure CORS policies (restrict to known frontend domains, not "*")
    - Enable HTTPS/TLS enforcement
    - Set up request/response logging middleware
    - Configure error handling and exception mappers
- Implement authentication and authorization:
    - JWT token validation:
        - Create dependency function to extract and validate JWT from Authorization header
        - Verify JWT signature using Auth0/Cognito public keys
        - Check token expiration
        - Extract user claims (user_id, roles, permissions)
    - Role-based access control (RBAC):
        - Define roles: Attending, Resident, Nurse, Pharmacist, Compliance Officer
        - Define role→permissions mapping:
            - Attending: patient-query, access-all-patients, audit-log-read
            - Resident: patient-query, access-assigned-patients-only
            - Nurse: limited-query, access-assigned-patients-only
            - Pharmacist: patient-query, medication-interaction-check
            - Compliance Officer: audit-log-read, admin access
        - Create permission check decorators
        - Implement patient relationship verification (clinician assigned to patient?)
    - Patient-level access control:
        - For each patient query, verify clinician is on care team or supervisor
        - Store access relationship in secure database
        - Audit every access attempt (success/failure)
- Implement Pydantic models for validation:
    - Request models: LoginRequest, QueryRequest, PatientSearchRequest
    - Response models: QueryResponse, PatientListResponse, ErrorResponse
    - Include field validation (required fields, string length limits, format checks)
- Set up structured logging:
    - Configure Python logging with JSON formatter
    - Log all requests: timestamp, user_id, endpoint, method, status
    - Log all errors with stack traces
    - Log all access attempts (auth success/failure, RBAC checks)
    - Implement log rotation and archival to secure storage
- Implement rate limiting:

    - Limit queries per user per minute (e.g., 60 requests/min)
    - Limit failed login attempts (e.g., lock after 5 failures)
    - Return 429 (Too Many Requests) when exceeded
- Configure HTTPS and TLS:
    - Generate self-signed certificate for local dev (openssl)
    - Configure production TLS via AWS Certificate Manager or similar
    - Force HTTPS redirect
    - Set security headers:
        - Content-Security-Policy
        - X-Frame-Options
        - X-Content-Type-Options
        - Strict-Transport-Security
- Implement health check endpoint:
    - /health: Returns 200 OK
    - /ready: Checks database, CyborgDB, key store connectivity
    - Useful for monitoring and load balancer health checks

Deliverable:

- FastAPI application scaffold
- JWT validation middleware
- RBAC implementation
- Pydantic validation models
- Structured logging setup
- Rate limiting configuration
- HTTPS/TLS setup
- Health check endpoints
- Comprehensive security documentation

Completion Criteria:

- FastAPI app starts without errors
- CORS policies correctly restricted
- JWT validation functional (valid tokens accepted, invalid rejected)
- RBAC enforced (unauthorized access blocked)
- All requests logged
- Rate limiting active and tested
- Health check endpoints responding correctly

Success Metrics:

- API startup: <5 seconds
- Authentication latency: <100ms per request
- RBAC check latency: <50ms per request
- Error handling: All exceptions caught and logged
- Security headers: All critical headers present

Task 4.2: Query Endpoint Implementation and Orchestration

Description:
Implement the core /ask-question endpoint that orchestrates the complete workflow: embedding generation, encrypted search, decryption, and LLM call.

Specific Instructions:

- Define endpoint signature:
  - **Route**: POST /api/v1/query
  - **Request body**:
    - patient_id: Pseudonymized patient ID
    - question: Clinical question from clinician
    - retrieve_k: Number of documents to retrieve (default 5)
    - temperature: LLM sampling temperature (default 0.7, range 0.0-1.0)
  - **Response body**:
    - query_id: Unique ID for this query (for audit trail)
    - answer: Generated clinical response
    - sources: List of retrieved document references (de-identified)
    - confidence: Confidence score (0-1, estimated from retrieval relevance)
    - disclaimer: Legal disclaimer about decision support
  - **Error responses**: 400 (validation error), 401 (unauthorized), 404 (patient not found), 500 (server error)
- Implement workflow steps:
  Step 1: Request validation and access control
  - Extract and validate JWT
  - Verify clinician authorized to query patient (RBAC + relationship check)
  - Validate question (not empty, <1000 characters)
  - Log query attempt with timestamp

  Step 2: Query embedding generation
  - Load embedding model (lazy-load, cache in memory)
  - Tokenize question (handle >512 tokens if needed)
  - Generate embedding (768-dim vector)
  - Validate embedding (no NaN/Inf)

  Step 3: CyborgDB encrypted search
  - Call CyborgDB API with query embedding
  - Pass filters: patient_id (must match), document_type (rank: note > discharge > lab)
  - Request top-k=10 (retrieve extra in case of filtering)
  - Receive encrypted results with similarity scores
  - Validate results received

  Step 4: Decryption in memory
  - For each retrieved result:
    - Retrieve encryption key from key store (with timeout)
    - Decrypt embedding metadata in memory
    - Reconstruct clinical snippet from metadata
    - Verify authentication tag (detect tampering)
    - Securely clear decrypted data from memory
  - Assemble context:
    - Patient pseudonym + age range (from patient_id)
    - Active medications (from CyborgDB metadata or separate query)

        - Active problems (from CyborgDB metadata)
        - Top-3 most relevant document snippets (by similarity score)

Step 5: Private LLM inference

  - Construct prompt template:
    You are a clinical decision support assistant. Your responses are for clinician review only and do NOT constitute medical orders or diagnoses.
    PATIENT CONTEXT:
      - Patient (pseudonym): [patient_pseudo_id]
      - Age range: [age_range]
      - Active conditions: [problem_list]
      - Current medications: [medication_list]

    RELEVANT MEDICAL RECORD EXCERPTS:
    [Document 1 - type: discharge summary, date: [relative_date]]
    [snippet_text]
    [Document 2 - type: clinical note, date: [relative_date]]
    [snippet_text]
    [Document 3 - type: lab report, date: [relative_date]]
    [snippet_text]
    CLINICAL QUESTION: [question]
    RESPONSE (max 500 words, include clinical reasoning):
  - Call LLM with prompt:
      - Model: GPT-NeoX or GPT-J
      - Temperature: [user-specified or default 0.7]
      - Max tokens: 500
      - Stop sequences: ["CLINICAL QUESTION:", "---"]
  - Implement timeout (e.g., 30 seconds)
  - Validate response (not empty, not refusing without reason)

Step 6: Safety guardrails and post-processing

  - Check response for forbidden patterns:
      - Specific drug dose recommendations without pharmacy approval
      - Diagnosis assertions (ensure framed as "differential diagnosis")
      - Urgent action statements (flag for immediate physician review if found)
  - Add mandatory disclaimer to all responses:
      - "This is clinical decision support, not a medical order"
      - "Consult with pharmacy and clinical team before implementing"
  - Truncate response if exceeds safe length (500 words)
  - Strip any PHI that might have leaked (post-processing check)

Step 7: Response logging and return

  - Log query completion:
      - query_id, patient_id, clinician_id, question_text
      - retrieved_documents (count), similarity_scores (top-k)
      - response_text, response_tokens, inference_latency
      - timestamp, outcome (success/failure)
  - Store in audit database
  - Return response with disclaimer and source references

- Implement error handling and fallback:
  - If CyborgDB search fails: Return error response, alert operator
  - If LLM timeout: Return partial response with warning
  - If key decryption fails: Log error, return error response (no plaintext fallback)
  - If RBAC check fails: Return 401 Unauthorized, log security event
- Implement timeout management:

    - CyborgDB search: 5 second timeout
    - LLM inference: 30 second timeout
    - Total endpoint response: 35 second timeout
    - Return graceful error if exceeded

Deliverable:

- Query endpoint implementation
- Workflow orchestration logic
- Safety guardrails and validation
- Error handling and fallback strategies
- Audit logging integration
- Comprehensive endpoint documentation

Completion Criteria:

- Endpoint functional and responding to valid requests
- RBAC enforcement verified (unauthorized access rejected)
- Encrypted search working (retrieved documents relevant)
- Decryption functional (plaintext retrieved in memory only)
- LLM generation working (responses clinically reasonable)
- All responses include mandatory disclaimer
- Audit logging capturing all query details
- Error handling tested with simulated failures

Success Metrics:

- Endpoint latency: <5 seconds for complete workflow
- Query success rate: ≥95%
- Retrieved document relevance: ≥80% (spot-check)
- LLM response generation: <30 seconds
- RBAC enforcement: 100%
- Error handling: All failures logged and alerted

Task 4.3: Authentication Integration (Auth0/Cognito)

Description:
Integrate Auth0 or AWS Cognito for secure clinician authentication, MFA, and token management.

Specific Instructions:

- Set up Auth0 (SaaS option) or AWS Cognito (managed service):
    - **Auth0 setup**:
        - Create Auth0 tenant in test environment
        - Create Auth0 application (type: Regular Web Application)
        - Configure allowed callback URLs (localhost:3000/callback for dev)
        - Configure logout URL
        - Enable MFA (optional MFA for clinicians, mandatory for admins)
        - Configure rules for custom claims (roles, permissions)
    - **AWS Cognito setup**:
        - Create user pool

        - Configure sign-in options (email)
        - Enable MFA (TOTP recommended)
        - Create app client
        - Configure domain for hosted UI
        - Set up custom attributes (clinician_role, department)
- Implement OAuth 2.0 / OpenID Connect flow:
    - **Authorization Code Flow** (most secure for web apps):
        - User clicks "Login" on frontend
        - Redirect to Auth0/Cognito login page
        - User enters credentials and completes MFA
        - Redirect back to frontend with authorization code
        - Frontend exchanges code for tokens (backend-to-backend)
        - Issue access token and refresh token to frontend
        - Frontend stores tokens in secure httpOnly cookies (not localStorage)
- Implement token handling in backend:
    - Extract access token from Authorization header (Bearer token)
    - Validate JWT signature (verify against Auth0/Cognito public keys)
    - Check token expiry
    - Extract claims (user_id, roles, email)
    - Verify claims match user identity
- Implement refresh token flow:
    - Access tokens short-lived (1 hour)
    - Refresh tokens longer-lived (7 days)
    - Endpoint to refresh access token (POST `/auth/refresh`)
    - Handle refresh token rotation (issue new refresh token on each refresh)
    - Revoke old refresh tokens
- Implement login endpoint (backend):
    - **Route**: POST `/auth/login`
    - Accept username/password (or redirect to Auth0/Cognito login URL)
    - Validate credentials
    - Return tokens and user profile
- Implement logout endpoint:
    - **Route**: POST `/auth/logout`
    - Invalidate refresh tokens
    - Optional: Revoke tokens at Auth0/Cognito
- Implement MFA (optional but recommended):
    - Configure TOTP (Time-based One-Time Password) via authenticator app
    - Backup codes for recovery
    - Re-prompt MFA on sensitive operations (access patient data)
- Implement role and permission mapping:
    - Create custom claims in Auth0/Cognito:
        - `clinician_role`: Attending, Resident, Nurse, etc.
        - `department`: Cardiology, Neurology, etc.
        - `permissions`: Array of allowed actions
    - Map roles to API permissions in backend
    - Store role mappings in database for audit
- Test authentication flows:
    - Valid credentials → tokens issued
    - Invalid credentials → error
    - Expired tokens → refresh token used
    - Invalid refresh token → re-login required

    - MFA flow → challenges/responses
    - Logout → tokens invalidated

Deliverable:

- Auth0/Cognito configuration
- OAuth 2.0 / OIDC implementation
- Token validation middleware
- Refresh token management
- MFA implementation
- Login/logout endpoints
- Role and permission mapping
- Test scenarios and validation

Completion Criteria:

- Authentication functional (login works)
- Tokens issued and validated
- RBAC enforced based on roles
- MFA working (if configured)
- Refresh tokens rotating
- Logout invalidates tokens
- All auth flows tested

Success Metrics:

- Login success rate: ≥99%
- Authentication latency: <1 second
- Token validation latency: <100ms
- Refresh token success rate: ≥99%
- Security compliance: All auth flows follow OAuth 2.0 best practices

---

Task 4.4: Private LLM Deployment and Inference

Description:
Deploy a private, self-hosted language model (GPT-NeoX, GPT-J, or Llama) for clinical answer generation without external API calls.

Specific Instructions:

- Select and download LLM:
    - **GPT-NeoX-20B**: 20B parameters, good balance of quality and speed
    - **GPT-J-6B**: 6B parameters, faster inference, suitable for resource-constrained environments
    - **Llama-2-7B-Chat**: 7B parameters, optimized for conversational use
    - Download from Hugging Face Model Hub
    - Verify model integrity (hash check)
    - Document license terms (EleutherAI, Meta, etc.)
- Set up inference server:
    - **Option 1: vLLM** (recommended for simplicity and speed)
        - Install vLLM

        - Configure model path and quantization (FP16 recommended for VRAM efficiency)
        - Set inference parameters (batch size, GPU allocation, context length)
        - Start server (listens on port 8000)
    - **Option 2: Text Generation WebUI**
        - Install Text Generation WebUI
        - Configure model and settings
        - Expose REST API endpoint
    - **Option 3: TorchServe** (production-ready)
        - Package model in TorchServe format
        - Configure batch processing and auto-scaling
        - Deploy on GPU cluster
- Configure LLM inference parameters:
    - **Temperature**: 0.7 (default, controls randomness)
    - **Top-p (nucleus sampling)**: 0.9 (filters tokens by probability mass)
    - **Repetition penalty**: 1.1 (avoid repetitive outputs)
    - **Max tokens**: 500 (clinical response length limit)
    - **Stop sequences**: Custom sequences to stop generation (e.g., "---")
- Implement inference client in backend:
    - Create HTTP client to call LLM inference server
    - Construct prompt as documented in Task 4.2
    - Submit request with parameters
    - Receive response (generated text + metadata)
    - Implement timeout (30 seconds)
    - Handle errors (connection failed, timeout, invalid response)
- Optimize for latency:
    - Batch inference (queue multiple queries if backlog)
    - GPU optimization (FP16 precision, CUDA settings)
    - Model caching (keep model in GPU memory)
    - Warm-up requests (reduce cold-start latency)
    - Monitor inference latency (target: <5 seconds per 100-token response)
- Implement logging and monitoring:
    - Log all inference requests and responses
    - Monitor inference latency, VRAM usage, throughput
    - Alert on failures (server down, out of VRAM)
    - Track model performance over time
- Test inference quality:
    - Generate sample responses for various clinical questions
    - Evaluate responses for:
        - Clinical accuracy (consult with medical advisor if available)
        - Hallucination (false claims, made-up facts)
        - Bias (culturally appropriate, non-discriminatory)
        - Explainability (reasoning provided)
    - Document test results and edge cases
- Implement safety filters:
    - Post-process LLM responses for:
        - Specific drug dosages (flag for review)
        - Urgent language (flag for immediate review)
        - Harmful instructions
        - PHI leakage (if any patient names appear)
    - Log any flagged responses

        - Include disclaimers in response

Deliverable:

- LLM model downloaded and verified
- Inference server deployed (vLLM, Text Generation WebUI, or TorchServe)
- Inference client implementation
- Prompt templates and parameter configuration
- Safety filters and post-processing
- Performance optimization documentation
- Monitoring and logging setup
- Test results and quality assessment

Completion Criteria:

- LLM inference server running and responding to requests
- Inference latency <5 seconds for 100-token response
- Response quality acceptable (clinically reasonable, no hallucinations)
- Safety filters functional (flagged responses logged)
- Model loaded and ready for queries
- Monitoring active

Success Metrics:

- Inference success rate: ≥98%
- Inference latency: <5 seconds per response
- Response quality: ≥80% acceptable (manual evaluation)
- Hallucination rate: <5%
- Server availability: ≥99% uptime

Task 4.5: Audit Logging and Compliance Infrastructure

Description:
Implement comprehensive audit logging that captures all access, queries, and system events for HIPAA compliance and incident investigation.

Specific Instructions:

- Design audit log schema:
    - **Core fields**:
        - timestamp: ISO 8601 timestamp
        - user_id: Clinician ID
        - action: Type of action (login, query, retrieve, modify, delete)
        - resource: What was accessed (patient_id, document_id)
        - outcome: Success/Failure
        - details: JSON with additional context
        - ip_address: Source IP
        - user_agent: Browser/client info
    - **Sensitive fields**: Encrypted at rest
    - **Immutable**: Append-only, no modifications or deletions
- Choose audit log storage:
    - **PostgreSQL**: Reliable, supports encryption at rest, good for compliance

    - **CloudWatch Logs**: AWS-native, automated archival, good for analysis
    - **ELK Stack**: Elasticsearch for indexing, Logstash for processing, Kibana for visualization
- Implement audit logging points:
    - **Authentication events**:
        - Login attempt (success/failure)
        - Logout
        - Token refresh
        - MFA challenges
    - **Authorization events**:
        - Access granted
        - Access denied (RBAC violation)
        - Permission checks
    - **Query events**:
        - Query submitted
        - Patient accessed
        - Documents retrieved
        - LLM response generated
    - **System events**:
        - Configuration changes
        - Error conditions
        - Performance anomalies
    - **Compliance events**:
        - Audit log accessed (who reviewed logs)
        - Data export
        - Permission grants/revocations
- Implement logging in backend:
    - Use Python logging module with JSON formatter
    - Create audit logger instance
    - Call audit logger at each logging point with action, resource, outcome
    - Include try/except to ensure logging happens even on errors
- Implement audit log encryption:
    - Encrypt sensitive fields (user_id, patient_id, query text) before storage
    - Use column-level encryption (database-supported or application-level)
    - Store encryption keys in secure key management service
    - Log all access to audit logs (meta-logging)
- Implement audit log retention:
    - HIPAA requirement: ≥6 years
    - Archive old logs to cold storage (S3 Glacier, Azure Archive) after 90 days
    - Maintain indexes for recent logs (last 90 days) for fast access
    - Set up automated archival process
- Implement audit log search and analysis:
    - Create SQL queries or Kibana dashboards for common searches:
        - All accesses to patient X on date Y
        - All queries by clinician Z
        - All failed authentication attempts
        - All access by role (e.g., all resident accesses)
    - Implement full-text search over query text
    - Export audit logs for compliance audits (CSV, PDF)
- Implement anomaly detection and alerting:
    - Set up alerts for suspicious patterns:

        - Bulk patient access (>100 in short time)
        - After-hours access
        - Access from unusual locations
        - Failed login attempts (>5 in 10 minutes)
    - Alert security team on anomalies
    - Create incident response playbook
- Implement compliance reporting:
    - Generate monthly audit reports:
        - Access statistics (queries per user, per patient)
        - Failed attempts and anomalies
        - System uptime and performance
    - Generate compliance attestation (HIPAA audit trail confirmation)

Deliverable:

- Audit log schema definition
- Audit logging implementation (code points)
- Audit log storage infrastructure (database)
- Encryption for sensitive fields
- Audit log search and analysis capabilities
- Retention and archival policies
- Anomaly detection and alerting
- Compliance reporting templates
- Audit log access controls

Completion Criteria:

- All critical events logged
- Logs captured successfully over 24 hours
- Audit logs immutable (cannot be deleted/modified)
- Sensitive fields encrypted
- Search functionality working
- Retention policy configured
- Anomaly alerts working
- Compliance reports generated

Success Metrics:

- Logging coverage: ≥95% of events captured
- Logging latency: <100ms additional per request
- Audit log completeness: Zero missed events
- Encryption coverage: 100% of sensitive fields
- Retention compliance: All logs retained per policy
- Query performance: Searches complete in <5 seconds

Phase 5: Frontend, Testing, & Demonstration

Objective

Build the clinician UI, implement comprehensive testing, and prepare the system for demonstration and evaluation.

Task 5.1: Clinician Frontend UI Development

Description:
Build a user-friendly web interface for clinicians to authenticate, select patients, and interact with the chatbot.

Specific Instructions:

- Design UI/UX:
    - **Screens**:
        - Login page (email + password, MFA if enabled)
        - Patient selector (search by name, MRN, list view)
        - Chat interface (question input, response display, source documents)
        - Audit log viewer (for compliance officers)
    - **Design principles**:
        - Accessibility: WCAG 2.1 AA compliance (color contrast, keyboard navigation)
        - Responsiveness: Works on desktop and tablet
        - Clarity: Clinical context always visible (patient name, active problems, meds)
        - Disclaimers: Mandatory warnings clearly visible
- Implement authentication flow:
    - Login page with email/password fields
    - Redirect to Auth0/Cognito login URL
    - Handle callback and token storage (httpOnly cookie recommended)
    - Display user name and logout option after login
- Implement patient selection screen:
    - Search functionality (search by patient pseudo-ID or name)
    - List view of recent patients
    - Filter by department or provider
    - Display patient summary (age, active conditions, recent visits)
    - Confirm patient selection before proceeding
- Implement chat interface:
    - Display patient context (pseudo-name, age, medications, problems)
    - Question input box with character limit (1000 chars)
    - Submit button (with loading indicator during API call)
    - Response display area:
        - Generated answer in readable format
        - Mandatory disclaimer prominently displayed
        - Source documents listed (with icons for document type)
        - Option to expand source documents
    - Chat history (display previous queries in same session)
    - Feedback mechanism (was this response helpful? flag for review)
- Implement responsive design:
    - Desktop: Multi-column layout (patient info, chat, source docs)
    - Tablet: Collapsible sidebar for patient info
    - Mobile: Single column (patient info collapsible, chat primary)

- Implement security:
    - Secure token storage (httpOnly cookies, not localStorage)
    - CSRF protection (token in request headers)
    - Input validation and sanitization
    - Logout clears all stored data
    - No sensitive data logged to console
- Implement error handling:
    - Display user-friendly error messages
    - Retry mechanism for transient failures
    - Fallback UI if API unavailable
- Test for accessibility:
    - Color contrast checker (≥4.5:1 for normal text)
    - Keyboard navigation (tab through all elements)
    - Screen reader testing (labels, ARIA attributes)
    - Responsive design testing (mobile, tablet, desktop)

Deliverable:

- React or HTML/JavaScript UI implementation
- Login and authentication flow
- Patient selector
- Chat interface
- Responsive design
- Accessibility features
- Error handling
- Security implementation
- Comprehensive testing documentation

Completion Criteria:

- UI loads without errors
- Login flow works end-to-end
- Patient selection functional
- Chat interface responsive and intuitive
- Disclaimer clearly visible on all responses
- Accessibility requirements met (WCAG 2.1 AA)
- Works on desktop, tablet, mobile
- No security vulnerabilities (input validation, CSRF protection)

Success Metrics:

- UI load time: <3 seconds
- Responsiveness: Works on all target screen sizes
- Accessibility score: ≥90% (automated checker)
- User journey: ≤5 clicks to submit first query
- Error handling: All error cases handled gracefully

Task 5.2: Unit and Integration Testing

Description:
Implement comprehensive test suites covering all components and workflows.

Specific Instructions:

- **Unit tests** (test individual functions/methods):
  - PHI detection and masking:
    - Test each Presidio analyzer (PII detection rate ≥95%)
    - Test masking logic (original not in output)
    - Test edge cases (dates in different formats, abbreviations)
  - Embedding generation:
    - Test embedding output shape (768-dim)
    - Test consistency (same input → same output)
    - Test numerical stability (no NaN/Inf)
  - Encryption/decryption:
    - Test AES-256-GCM encryption/decryption
    - Test key generation and rotation
    - Test authentication tag verification
  - RBAC:
    - Test role-based permission checks
    - Test user-patient relationship verification
    - Test edge cases (no role, unknown role)
  - API endpoint validation:
    - Test request validation (missing fields, invalid types)
    - Test response format (all required fields present)
- **Integration tests** (test component interactions):
  - End-to-end query workflow:
    - Input: Patient ID, question
    - Verify: Embedding generated, CyborgDB search called, LLM response returned
    - Verify: Audit log entry created
  - Authentication + Authorization:
    - Login user → query patient (should succeed)
    - Login user → query unauthorized patient (should fail)
    - No auth → query (should fail with 401)
  - Data pipeline:
    - Input: Raw FHIR data
    - Verify: De-identification applied, no PHI in output
    - Verify: Embeddings generated and encrypted
    - Verify: Stored in CyborgDB successfully
- **Test data and fixtures**:
  - Create test FHIR records with known PHI and expected outputs
  - Create test users with different roles and permissions
  - Create test patient relationships
  - Mock external services (Auth0, CyborgDB, LLM) for isolated testing
- **Testing tools and framework**:
  - Framework: pytest (Python)
  - Mocking: pytest-mock, unittest.mock
  - Fixtures: pytest fixtures for reusable test data

    - Coverage: pytest-cov to measure code coverage (target: ≥80%)
    - CI integration: Run tests on every PR
- **Test organization**:
    - `/tests/unit` - Unit tests by component
    - `/tests/integration` - Integration tests by workflow
    - `/tests/fixtures` - Test data and mocks
- **Specific test cases**:
    - **PHI masking**: 20+ test cases covering all PHI types
    - **Embedding**: 10+ test cases covering model consistency, edge cases
    - **Encryption**: 15+ test cases covering encryption/decryption, key rotation
    - **Query workflow**: 10+ test cases covering success, error, edge cases
    - **RBAC**: 15+ test cases covering all role combinations
    - **API endpoints**: 5+ test cases per endpoint (valid, invalid, unauthorized)
- **Test execution and reporting**:
    - Run tests locally: `pytest tests/`
    - Generate coverage report: `pytest --cov=src tests/`
    - Display coverage in CI: Comment coverage % on PR
    - Fail CI if coverage <80%

Deliverable:

- Unit test suite (50+ test cases)
- Integration test suite (20+ test cases)
- Test fixtures and mocks
- pytest configuration
- Coverage reporting setup
- Test documentation and guidelines
- CI/CD integration for automated testing

Completion Criteria:

- All critical components have unit tests (PHI, embedding, encryption, RBAC, API)
- All critical workflows have integration tests
- Code coverage ≥80%
- All tests passing
- CI/CD running tests on every commit
- Test execution time <5 minutes

Success Metrics:

- Unit test count: ≥50 tests
- Integration test count: ≥20 tests
- Code coverage: ≥80%
- Test pass rate: 100%
- Test execution time: <5 minutes
- CI/CD success rate: 100%

Task 5.3: Load and Performance Benchmarking

Description:
Conduct comprehensive performance testing to validate latency, throughput, scalability, and resource utilization under realistic loads.

Specific Instructions:

- **Define performance targets**:
  - Query latency: <5 seconds end-to-end (perception of "fast")
  - Throughput: ≥10 concurrent queries without degradation
  - Error rate: <1%
  - Resource utilization: <80% CPU, <80% GPU VRAM, <80% memory
  - Availability: ≥99% uptime
- **Load testing scenarios**:
  - **Scenario 1: Steady state**
    - 5 concurrent clinicians
    - Duration: 10 minutes
    - Expected: Consistent latency <5s, <1% errors
  - **Scenario 2: Peak load**
    - 50 concurrent clinicians
    - Duration: 5 minutes
    - Expected: Latency <7s, <2% errors (degradation acceptable)
  - **Scenario 3: Stress test**
    - Ramp up to 100 concurrent, hold for 2 minutes
    - Expected: System handles gracefully (queue requests, eventually timeout)
  - **Scenario 4: Sustained load**
    - 20 concurrent clinicians
    - Duration: 1 hour
    - Expected: No memory leaks, consistent performance
- **Benchmarking components** (measure each independently):
  - Query embedding generation: <200ms for single question
  - CyborgDB search: <500ms for 500+ embeddings
  - LLM inference: <5 seconds for 100-token response
  - API response time: <100ms (excluding LLM)
  - End-to-end query: <5 seconds
- **Tools**:
  - Load testing: Locust (Python-based, easy to script), Apache JMeter, or k6
  - Profiling: py-spy, cProfile (identify hotspots)
  - Monitoring: Prometheus + Grafana (metrics visualization)
  - Database monitoring: Query performance logs
- **Test script**:
  - Create realistic query workload:
    - Vary questions (10+ different clinical questions)
    - Vary patient IDs (ensure CyborgDB distributes across patients)
    - Simulate realistic think time between queries
  - Login before querying (test authentication scalability)
  - Track metrics: response time, error rate, resource usage
- **Execution**:
  - Run tests in isolated environment (not production)

    - Warm up system first (5 min) to stabilize
    - Record baseline metrics
    - Document hardware (CPU, GPU, RAM, network)
    - Capture logs during test (for troubleshooting)
- **Analysis and reporting**:
    - Generate performance report with:
        - Latency percentiles (p50, p95, p99)
        - Throughput (queries/sec)
        - Error rates and types
        - Resource utilization charts
        - Bottleneck identification (where is time spent?)
    - Create visual dashboard (latency over time, error rate spikes)
    - Document findings and recommendations:
        - Optimization opportunities
        - Scaling recommendations
        - Acceptable limits
- **Optimization based on results**:
    - If latency >5s: Identify bottleneck (LLM? CyborgDB? API?)
    - If errors >1%: Investigate failure modes
    - If resource usage >80%: Plan scaling strategy
    - Document optimizations and re-test

Deliverable:

- Load testing script (Locust/JMeter/k6)
- Performance test scenarios documented
- Monitoring and metrics setup
- Performance test results report
- Bottleneck analysis and optimization recommendations
- Scalability recommendations
- Benchmarking documentation for post-campaign

Completion Criteria:

- Load tests run successfully
- Performance metrics captured for all scenarios
- Latency <5s for steady state achieved (or documented as limitation)
- Throughput ≥10 concurrent users achieved
- Error rate <1% at steady state
- Resource utilization monitored
- Report generated with findings

Success Metrics:

- Latency p99: <5 seconds (steady state)
- Throughput: ≥10 concurrent queries
- Error rate: <1%
- Resource utilization: <80%
- Uptime during 1-hour test: ≥99%

Task 5.4: End-to-End System Testing and Integration Validation

Description:
Conduct comprehensive end-to-end testing covering all system components working together and all critical workflows.

Specific Instructions:

- **Test scenarios** (user journeys):
  Scenario 1: Happy path - clinician answers patient question
  - Clinician logs in
  - Selects patient
  - Asks clinical question
  - Receives answer with sources
  - Verifies audit log entry
  - Logout
  - Expected: All steps succeed, response clinically reasonable

  Scenario 2: Access control - unauthorized clinician cannot query patient
  - Clinician A logs in
  - Attempts to query patient assigned to Clinician B
  - Expected: 403 Forbidden, audit log shows denied access attempt

  Scenario 3: Data security - encrypted search functional
  - Verify embeddings encrypted in CyborgDB (cannot read plaintext)
  - Query system, verify correct results returned
  - Expected: Search works without decrypting on disk

  Scenario 4: Compliance - audit trail complete
  - Execute multiple queries
  - Review audit log
  - Verify all actions logged with timestamps, user IDs, outcomes
  - Expected: Complete trail of all activity

  Scenario 5: Error handling - graceful failure on component outage
  - Simulate CyborgDB outage
  - Attempt query
  - Expected: Graceful error response, audit log shows failure
  - Restore service, queries resume successfully

  Scenario 6: Safety guardrails - response filtering
  - Query LLM with prompts designed to elicit unsafe responses
  - Verify safety filters catch and flag responses
  - Expected: Unsafe responses flagged, disclaimer added
- **Test execution**:
  - Execute all scenarios manually first (verify expectations)
  - Document any deviations or unexpected behavior
  - Fix issues, re-test
  - Automate scenarios where possible (Selenium for UI, requests for API)
- **Validation checklist**:
  - [ ] Login/logout works
  - [ ] Patient selection works
  - [ ] Query submitted successfully
  - [ ] Response returned with answer + sources
  - [ ] Disclaimer included in response
  - [ ] RBAC enforced (unauthorized queries rejected)

    - [ ] Audit log captures all events
    - [ ] Encryption verified (no plaintext on disk)
    - [ ] Error handling graceful (no crashes)
    - [ ] Safety guardrails functional
    - [ ] Performance acceptable (<5s latency)
    - [ ] No memory leaks or resource issues
- **Test environment**:
    - Use identical setup to hackathon demo environment
    - Use realistic test data (synthetic FHIR)
    - Simulate network conditions (latency, packet loss)
- **Documentation**:
    - Document all test scenarios
    - Record test results (pass/fail)
    - Screenshot or video of successful workflows
    - Document any edge cases or known limitations
    - Create "Troubleshooting Guide" for demo day

Deliverable:

- End-to-end test scenarios documented
- Test execution results and sign-off
- Test videos/screenshots for demo preparation
- Troubleshooting guide
- Known limitations and workarounds documented
- Final system validation report

Completion Criteria:

- All critical workflows tested and passing
- RBAC and security controls verified
- Audit logging complete and accurate
- Error handling graceful
- Performance acceptable
- System ready for demonstration

Success Metrics:

- Test scenario success rate: 100%
- System stability: No crashes or unhandled errors
- User experience: Intuitive and responsive
- Security validation: All controls functioning

Task 5.5: Documentation and Demonstration Preparation

Description:
Create comprehensive documentation and prepare materials for hackathon demonstration and submission.

Specific Instructions:

- **Technical Documentation**:
    - Create `/docs` directory with:

        - ARCHITECTURE.md: System design, components, data flows
        - API_SPEC.md: Endpoint documentation (request/response format, error codes)
        - SETUP.md: Environment setup and deployment instructions
        - TESTING.md: How to run tests and expected results
        - DEPLOYMENT.md: Production deployment considerations
        - SECURITY.md: Security architecture, encryption, access control
        - PERFORMANCE.md: Benchmarking results, scalability analysis
    - Include diagrams where helpful (ASCII or embedded images)
    - Keep documentation up-to-date as system evolves
- **README.md for hackathon submission**:
    - **Project title**: HIPAA-Compliant Encrypted Medical Chatbot
    - **Overview**: 2-3 paragraph summary of what the system does
    - **Problem solved**: Privacy concerns with medical AI, HIPAA compliance
    - **Solution**: Encrypted vector search + private LLM + access controls
    - **Key features**:
        - End-to-end encryption (at rest, in transit, in-use)
        - Private LLM (no external API calls)
        - HIPAA compliance (audit logs, access control, de-identification)
        - Clinical decision support (embeddings from clinical docs)
    - **Technology stack**: CyborgDB, BioBERT, FastAPI, Auth0, etc.
    - **Quick start**: How to run locally (docker-compose up)
    - **Demo**: Link to demo video or live instance
    - **Benchmarks**: Performance metrics and results
    - **Team**: Team member names and roles
    - **Future work**: Post-campaign roadmap
    - **References**: Links to papers, CyborgDB docs, etc.
- **Demo materials**:
    - **Demo script**: Step-by-step walkthrough (5-10 minutes)
        - Show login
        - Select patient
        - Ask clinical question
        - Show response with sources
        - Show audit log
        - Explain security/encryption (architecture diagram)
    - **Live demo setup**:
        - Pre-load data in demo environment (500 patient records)
        - Pre-login as demo clinician
        - Pre-populate audit log with sample activities
        - Test all workflows beforehand
        - Have fallback: recorded demo video if live fails
    - **Presentation slides** (if hackathon requires):
        - Title slide (project name, team)
        - Problem statement (2 slides)
        - Solution architecture (2-3 slides with diagrams)
        - Technology stack (1 slide)
        - Demo (5-10 min live or video)
        - Results/Benchmarks (1-2 slides)
        - Impact/Future (1 slide)
        - Q&A
    - **Demo video** (backup):

        - Record screen walkthrough of all features
        - Narrate key points
        - Show performance/benchmarks
        - Keep <10 minutes
- **Code quality and submission readiness**:
    - Run linter (pylint, flake8): Fix all warnings
    - Run type checker (mypy): Ensure type annotations
    - Run tests: All passing
    - Check code coverage: ≥80%
    - Review security:
        - No hardcoded secrets
        - No SQL injection vulnerabilities
        - No unvalidated user input
        - Encryption implemented correctly
    - Check for TODOs/FIXMEs: Resolve or document
    - Final code review: Someone other than author reviews
- **Submission package**:
    - GitHub repository public (MIT or Apache 2.0 license)
    - README complete and clear
    - All documentation in place
    - Code well-commented
    - `.env.example` provided (no secrets)
    - Docker Compose working
    - Tests passing
    - Benchmarks documented
    - Final tag: `v1.0-hackathon-submission` in Git
- **Hackathon criteria alignment**:
    - **Technology Application**: Document how CyborgDB, medical LLMs, HIPAA frameworks applied
    - **Business Value**: Explain clinical benefit, operational efficiency, compliance risk reduction
    - **Presentation**: Clear README, good diagrams, working demo
    - **Originality**: Explain novel approach (combining encrypted search + private LLM + de-identification)

Deliverable:

- Comprehensive technical documentation
- README for hackathon submission
- Demo script and materials
- Presentation slides (if required)
- Demo video (backup)
- Code quality review and fixes
- Security audit checklist
- Submission package (GitHub repository)

Completion Criteria:

- Documentation complete and clear
- README compelling and actionable
- Demo prepared and tested
- Code quality meets standards

- All requirements for submission met
- Submission ready for evaluation

Success Metrics:

- Documentation completeness: 100% of key topics covered
- Code quality: All linting/type checks passing
- Demo success: All workflows demonstrable
- Presentation clarity: Clear to non-technical judges
- Submission readiness: No missing files or documentation

Task 5.6: Security and Compliance Audit

Description:
Conduct a final security and compliance audit to ensure system meets HIPAA standards and has no critical vulnerabilities.

Specific Instructions:

- **Security audit checklist**:
    - **Authentication**:
        - [ ] JWT validation implemented
        - [ ] Token expiration enforced
        - [ ] MFA available (if configured)
        - [ ] Password requirements enforced (if using native auth)
        - [ ] Session timeout configured
    - **Authorization**:
        - [ ] RBAC implemented for all endpoints
        - [ ] Patient-level access control verified
        - [ ] Least privilege principle followed
        - [ ] Permission checks on all sensitive operations
    - **Encryption**:
        - [ ] All data at rest encrypted (embeddings, audit logs)
        - [ ] TLS 1.3 for all network communication
        - [ ] Encryption keys in secure key store (KMS/Vault)
        - [ ] No hardcoded keys in source code
        - [ ] Key rotation scheduled and tested
    - **Data protection**:
        - [ ] PHI de-identification verified
        - [ ] No PHI in logs or error messages
        - [ ] Audit logs immutable and encrypted
        - [ ] Data retention policies enforced
    - **Input validation**:
        - [ ] All user input validated
        - [ ] SQL injection prevention (parameterized queries)
        - [ ] XSS prevention (input sanitization)
        - [ ] CSRF prevention (token-based)
        - [ ] File upload validation (if applicable)
    - **Error handling**:
        - [ ] No sensitive information in error messages
        - [ ] Stack traces logged but not returned to users

        - [ ] Graceful error handling (no crashes)
    - **Logging and monitoring**:
        - [ ] All access logged
        - [ ] Security events logged and alerted
        - [ ] Logs cannot be deleted or modified
        - [ ] Monitoring active for anomalies
    - **Infrastructure**:
        - [ ] Network isolation (VPC/firewall rules)
        - [ ] Secrets management (no env vars for sensitive data)
        - [ ] Secure defaults (HTTPS only, strong TLS)
        - [ ] Regular backups (for recovery)
        - [ ] Patch management (dependencies up-to-date)
- **Compliance audit checklist** (HIPAA):
    - [ ] Administrative Safeguards:
        - Security management process defined
        - Authorization and supervision procedures
        - Security awareness training documented
        - Security incident procedures documented
    - [ ] Physical Safeguards:
        - Facility access controls
        - Workstation use policies
        - Workstation security procedures
    - [ ] Technical Safeguards:
        - Encryption and decryption (in-use, at-rest, in-transit)
        - Access controls (authentication, authorization)
        - Audit logs and accountability
        - Integrity controls (detect tampering)
    - [ ] Organizational Requirements:
        - Business Associate Agreements (BAAs) with vendors
        - Workforce security policies
        - Information access management
    - [ ] Documentation:
        - Security policies documented
        - Risk assessment completed
        - Compliance audit trail maintained
- **Vulnerability testing**:
    - Static analysis: Run SAST tool (Bandit for Python, npm audit for Node)
        - Bandit: `bandit -r src/`
        - npm audit: `npm audit` (for frontend)
        - Fix critical/high severity issues
    - Dependency scanning: Check for known vulnerabilities
        - pip: `pip install safety && safety check`
        - npm: `npm audit`
        - Update dependencies if vulnerabilities found
    - Manual code review:
        - Check for hardcoded secrets (grep for "password", "api_key", "secret")
        - Check for insecure randomness (should use `secrets` module)
        - Check for unsafe deserialization (pickle, yaml.load)
        - Check for SQL injection risks
    - Penetration testing (if time permits):
        - Attempt unauthorized access (invalid JWT, no token)

        - Attempt unauthorized patient access (cross-patient query)
        - Attempt SQL injection in search inputs
        - Test CSRF protection
        - Test rate limiting
- **Documentation of security measures**:
    - Create SECURITY.md documenting:
        - Encryption mechanisms (algorithms, key management)
        - Authentication and authorization implementation
        - Audit logging and monitoring
        - Incident response procedures
        - Known limitations and mitigation strategies
    - Document any deviations from best practices with justification
    - Create security runbook for operators (key rotation, incident response)
- **Sign-off**:
    - Complete security checklist
    - Document any findings and remediation
    - Compliance lead (or equivalent) reviews and approves
    - Get written sign-off (email/document) for submission

Deliverable:

- Security audit checklist (completed)
- Vulnerability scan results (SAST, dependency scan)
- Manual code review findings
- Penetration test results
- HIPAA compliance checklist (completed)
- Security documentation (SECURITY.md)
- Remediation plan for any findings
- Security sign-off (approval document)

Completion Criteria:

- All critical security issues addressed
- No critical or high-severity vulnerabilities remaining
- HIPAA compliance checklist ≥95% complete
- Encryption verified and working
- Access controls validated
- Audit logging functional

Success Metrics:

- SAST findings: 0 critical, 0 high
- Vulnerability scan findings: 0 critical, 0 high
- Compliance checklist: ≥95% items verified
- Security review: Approved by security lead

Summary of Deliverables by Phase

Phase 1 (Days 1-2): Foundation

- Development environment (Docker Compose)
- Synthetic FHIR dataset (500+ documents)
- Architecture documentation
- Repository structure and CI/CD

Phase 2 (Days 3-4): De-Identification

- PHI masking pipeline (Presidio + spaCy)
- Data orchestration (Prefect/Airflow)
- De-identification validation report
- Quality metrics (≥95% PHI detection)

Phase 3 (Days 5-6): Encrypted Storage

- Embedding generation module (BioBERT/ClinicalBERT)
- Encryption and key management
- CyborgDB integration
- Performance benchmarks

Phase 4 (Days 7-8): Backend & LLM

- FastAPI backend with full RBAC
- Auth0/Cognito integration
- Private LLM deployment
- Audit logging infrastructure

Phase 5 (Days 9-10): Frontend & Testing

- Clinician web UI
- 50+ unit tests, 20+ integration tests
- Load testing and benchmarking
- Security and compliance audit
- Complete documentation
- Demo-ready system

Quality Assurance Throughout Sprint

Daily Checkpoints

- End of each day: Commit code to Git, run CI/CD pipeline
- All tests passing before code merge
- No merge conflicts or broken builds

Phase Boundaries

- End of each phase: Integration checkpoint
    - Components from previous phases still working
    - New components integrated successfully
    - Performance benchmarks verified
    - Documentation updated

Pre-Demo Validation (Day 10)

- [ ] Full system working end-to-end
- [ ] All security checks passing
- [ ] Demo rehearsed and timed
- [ ] Fallback demo video ready
- [ ] Documentation complete
- [ ] Code submitted to repository

Risk Mitigation Strategies

Risk	Probability	Impact	Mitigation
CyborgDB integration issues	Medium	High	Start early, build mock for parallel development
LLM inference too slow	Medium	High	Test multiple models, plan GPU optimization early
PHI masking incomplete	Low	Critical	Extensive testing, manual review of outputs
Authentication integration delays	Low	Medium	Use mock tokens initially, integrate later
Performance not meeting targets	Medium	Medium	Continuous benchmarking, optimization sprints
Team member unavailability	Low	Medium	Cross-train on critical components
Scope creep	High	Medium	Prioritize MVP features, defer polish for post-campaign

---

Success Metrics Summary

Metric	Target	Phase	Verification
PHI detection accuracy	≥95%	2	Validation testing
Embedding generation throughput	≥100 docs/hour	3	Benchmarking
Query latency	<5 seconds	4-5	Load testing
Code coverage	≥80%	5	pytest-cov report
RBAC enforcement	100%	4-5	Integration tests
Audit logging completeness	100% of events	4-5	Audit log review
Documentation completeness	≥90%	5	Checklist review
Security compliance	≥95% HIPAA items	5	Audit checklist
Demo readiness	All workflows functional	5	End-to-end testing

Conclusion

This detailed sprint plan provides a clear roadmap for building a production-quality, HIPAA-compliant encrypted medical chatbot within the hackathon timeframe. Each phase builds on previous work, with continuous testing and validation ensuring quality and security.

The key to success is:

1. **Prioritization**: Focus on MVP first (core workflow), defer nice-to-have features
2. **Integration**: Validate components work together at each phase boundary
3. **Testing**: Continuous testing from day 1, not just at the end
4. **Documentation**: Keep docs updated with code, not as afterthought
5. **Demo preparation**: Rehearse demo, prepare fallbacks, ensure smooth presentation

By following this structured approach, the team will deliver a compelling, technically sound, and business-valuable solution ready for hackathon evaluation.